Understanding Understanding: How Do We Reason About Computational Logic?

by

Hammad Ahmad

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Hammad Ahmad

hammada@umich.edu

ORCID iD: 0000-0002-0434-6194

ACKNOWLEDGEMENTS

My journey through graduate school has been a long and arduous one. There are many people without whom I would not have been able to finish my Ph.D. This brief dedication section will almost certainly fail to capture the true level of my appreciation for folks who helped me get here, but let me try.

Thank you to my advisor, Westley Weimer. Thank you for patiently guiding me these past five years to be a successful researcher and an effective educator. Thank you for making sure external factors do not get in the way of my completing this degree. Thank you for supporting my desire to balance research and teaching. Thank you for providing me with opportunities and resources to pursue my career goals. Thank you for your life advice on things that were bothering me. Thank you for always being available (even during your sabbatical!) to answer my questions or listen to me complain. Thank you for making sure I am well-prepared to talk to my students about a variety of CS problems (and yes, that includes NP-completeness). I know, I know... *the check is in the mail.* But seriously, thank you for everything.

Thank you to my committee members: Stephanie Forrest, Amir Kamil, and Taraz Lee. Thank you for your support and and advice on my research and career trajectories. Thank you for providing me feedback to make sure my research endeavors succeeded. Thank you for being patient with me as I navigated the graduation requirements, and for carving out time during busy semesters for our meetings.

Thank you to Jean-Baptiste Jeannin. Your patience and advice as I was figuring out graduate school helped make the transition easier, and for that I am really grateful. Thank you for introducing me to a variety of research projects and helping me explore the research area that most excites me.

Thank you to Sara Sprenkle. Thank you for all you have done to help me be who I am today. I still remember feeling lost as an undergraduate. I still remember feeling the *need* to major in Computer Science and *X*. You listened to my ramblings patiently, provided me with career advice, introduced me to research, guided me through the process of graduate school applications, and years later, served as a mentor again when I taught at W&L. Thank you for everything.

Thank you to my collaborators over the years for all you taught me during my time in graduate school: Kevin Angstadt, Stephanie Forrest, Yu Huang, Amir Kamil, Kimberly Diaz, Manos Kapritsos, Baris Kasikci, Taraz Lee, Karem Sakallah, and Zohreh Sharafi. Without your expertise, none of this work would have been possible.

Thank you to my teaching mentors who have helped me grow as an educator: Dave Paoletti, Marcus Darden, Héctor Garcia-Ramirez, and Sara Sprenkle, Taha Khan, Cody Watson, Amir Kamil, and others. Teaching was a major part of my graduate career, and I am incredibly grateful for your efforts helping me find my footing in the classroom.

Thank you to Taraz Lee, James Brissenden, and Quynh Nguyen for your time and effort helping me get comfortable with TMS, and for being patient with my ignorant questions about brain anatomy and stimulation protocols.

Thank you to Jasmin Stubblefield for helping me navigate the doctoral degree requirements, Melissa Vaquera for helping me with participant compensation for my human studies, Dina Salhani for helping me with fMRI and TMS scheduling (and for bearing with me as I figured out how fMRI sessions are conducted), and Karen Liska for helping me with my logistical and administrative enquiries. Completing this degree without your assistance would have been very difficult.

Thank you to my office mates over the years: Kevin Angstadt, Kevin Leach, Zohreh Sharafi, Yu Huang, Madeline Endres, Colton Holoday, Priscila Santiesteban, Zachary Karas, Annie Li, Kaia Newman, Emma Shedden, Wenxin He, Manasvi Parikh, Danniel Hu, and Yiannos Demetriou. I truly enjoyed our conversations about random things and appreciated your willingness to let me distract you when I felt like procrastinating. Special thank you to Priscila for keeping me company when I did not feel like going hiking (or generally being adventurous) during WRG retreats.

Thank you to all participants in my studies. Your willingness to contribute to my research helped me advance our understanding of the cognition behind computer science.

Thank you to my family and friends, without whom I would not have been able to finish this degree.

Thank you to Mom, Dad, Rida, Midha, Khala, Idrees Mamo, Yaqoob Mamo, and others. No words I write here can capture how much I appreciate your unconditional love and support. Thank you Mom and Dad for always believing in me, even when I doubted my ability and endurance to finish graduate school. Thank you Rida for bearing with the countless medical questions I hurled your way (and rolling your eyes at most of them). Thank you Midha for being... you (and for offering uncharacteristically sage advice and very characteristic amusement time and again). *Aap sab kay baghair ye paanch*, or actually, *aath saal guzarna namumkin hota. Ye sab aap ki waja say hai. Aap ki duaoun ki waja say. Aap* *kay pyaar ki waja say. Aap ki madad ki waja say.* Before it starts to sound too unlike me, *meray dil ki gehrayoun say aap sab ka bohat bohat bohat shukriya.*

Thank you to Madhav, Austin, Pepito, Jesse, Anu, Uma, and Brittany. Thank you for keeping me sane these past few years. Thank you for our game nights. Thank you for providing me with endless entertainment and distractions. Thank you (I think) for roasting me for my questionable life decisions, and for my obsession with FIFA. I promise it is not *exactly* the same game every year. Jokes aside, thank you for making graduate school tolerable.

Thank you Keerthana for introducing me to Detroit (after four years of my laziness to drive 30 minutes from Ann Arbor) and Ginger Deli, and for pushing me do "young people things". Thank you Stina for letting me play with Taro and Coco to de-stress, and for providing shelter during my never-ending series of power outages.

Thank you to Amur for a countless number of things. Thank you for teaching me how to use the Wolverine Access UI to register for my first courses, and for everything thereafter. Thank you for being willing to be a guinea pig for my talks and studies. Thank you for cheering me on across the finish line, and for making all of this easier.

Finally, thank you to the reader taking out time to go over my pontifications. I hope you find the research in here informative and useful.

If I have forgotten to explicitly thank you, please forgive my fatigued brain. I truly do appreciate everything people in my life have done for me to get to and across this finish line. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AMT	Active Motor Threshold
AOI	Area of Interest
APR	Automated Program Repair
AST	Abstract Syntax Tree
fMRI	Functional Magnetic Resonance Imaging
fNIRS	Functional Near-Infrared Spectroscopy
GP	Genetic Programming
HDL	Hardware Description Language
M1	Primary Motor Cortex
SMA	Supplementary Motor Area
TBS	Theta Burst Stimulation
TMS	Transcranial Magnetic Stimulation

ABSTRACT

Computers fundamentally do not reason like humans do, making it time-consuming and expensive for programmers to find and fix mistakes in, or maintain, hardware and software. Given that such maintenance activities can often comprise up to 90% of the total cost associated with hardware and software, interest in better preparing future programmers for the computational logic reasoning required for computer science industry and academia has grown. We argue that understanding how programmers reason about computational logic can guide tool development and training activities for more efficient logical reasoning. In this thesis, we use non-intrusive, objective measures to present three research components investigating cognition for computational logic reasoning tasks, with a focus on digital logic, mathematical logic, and programming logic. We further illuminate, and advocate the investigation of, promising cognitive interventions to help programmers reason about computers.

First, we present a first-of-its-kind automated algorithm to repair defects in hardware designs (i.e., digital logic) and a human study investigating its use as a debugging assistant for programmers. We also produce a publicly-available benchmark suite of defects for the evaluation of future automated repair techniques for hardware. We find that our approach, CirFix, is able to successfully repair 16 out of 32 defects, representing a repair rate similar to that of established software repair techniques. In our human study involving 41 participants, we further find a preference among programmers for CirFix as a debugging aid for defects spanning multiple lines, primarily in classroom contexts. Our study paves the path for exploration of automated repair of hardware designs to reduce maintenance costs in industry and help programmers in the classroom.

Second, we present a human study involving 34 participants tasked with finding mistakes in formal algorithmic proofs (i.e., mathematical logic). We use eye-tracking to investigate the effects of various problem-solving strategies on formalism comprehension task outcomes. Analyzing participant responses, we find that incoming preparation and programmer self-perceptions are not accurate predictors of task outcomes, and that differences in outcomes can be attributed to inductive and recursive logical reasoning. Analyzing eye-tracking data, we find that more-prepared programmers employ different visual strategies but achieve similar outcomes to those of less-prepared programmers, and that programmers with more successful outcomes frequently go back and forth between presented material. Our results advocate for pedagogical interventions in undergraduate computer science theory courses to better prepare future programmers for formal reasoning in industry and academia.

Third, we present a human study probing causality between spatial reasoning and program comprehension (i.e., programming logic) using transcranial magnetic stimulation (TMS), a non-invasive neurostimulation technique. Analyzing responses collected from 16 participants, we replicate a prior psychology study showing that TMS impacts spatial reasoning. More importantly, and contrary to previously-established correlations, we find no evidence of a simple causal relationship between activity in the primary motor cortex or the supplementary motor area and programming outcomes. Further, we find that TMS can affect response time for programming tasks. Ours is the first study to use TMS in a computer science context, and our results advocate for further exploration of the technique to investigate neural activity for programming.

This thesis shows that it is possible to use objective measures to obtain mathematical models describing programmer behavior and cognition for computational logic reasoning tasks, and these models can highlight prospective cognitive interventions for student training.

CHAPTER 1

Introduction

The global IT industry harbors a market share totaling \$5 trillion in 2024, with predicted annual growth rates of 6–8% [1]. The average user spends a significant portion of the day using electronic devices that comprise hardware and software components [2, 3, 4]. As such, it is critical for stakeholder companies to ensure correct functionality of their IT infrastructure. Indeed, a service downtime of mere minutes can be catastrophic for companies, often resulting in millions in lost revenue [5] and shrinking of consumer confidence and trust [6].

To ensure that their IT services (including both hardware and software) function as intended, companies often employ a multitude of engineers and software developers (collectively referred to as *programmers* in this thesis) who command high wages in the modern economy [7]. Such professionals end up spending most of their time on *maintenance activities* (e.g., applying fixes to hardware or software to rectify incorrect or unexpected behavior). Reports have suggested that programmers often spend around 60%, and up to 90%, of their paid hours on such activities [8, 9], making maintenance the most expensive stage of the hardware and software development process. As an example, a 2020 report estimated the cost of poor software quality in the US alone to be over \$2 trillion [10].

Many discrepancies between expected and actual behavior from hardware and software stem from an incorrect understanding of user requirements and human errors during the design process (see Section 2.1 for details on the requirements gathering and design processes). However, even when efforts are put in place to keep such programmer errors in check, one key issue makes addressing the discrepancies a challenging and time-consuming task: computers do not reason like human beings do [11, 12]. While there exist computational technologies that mimic a human brain for reasoning (e.g., neural networks [13]), fundamentally, computers operate on bits (i.e., zeros and ones) at incredibly fast speeds — often up to billions of operations in a given second. By contrast, human brains operate using signals transmitted through a vast and complex array of neurons. This reasoning divide frequently results in produced code that appears correct at the surface level, yet fails to produce the expected behavior, eventually requiring expensive programmer maintenance efforts [14, 15]. More formally, such *bugs* (also referred to as *defects*) correspond to incorrect code that, during runtime, produces a computed value different from a theoretically correct value, often as a result of flawed human reasoning [16]. Finding and fixing bugs requires programmers to *reason about the underlying logic* (e.g., by tracing through program code, reading and writing formal proofs of correctness, etc.; see Section 2.1), given an input and expected program behavior or user requirements.

A common way to help humans understand how computers "think" is through training and education, and this training predominantly takes place at the undergraduate education level [17]. As such, reasoning about computational logic not only constitutes a fundamental pillar of software engineering activities [18, 19, 20], but also forms a core component of undergraduate computer science curricula [21]. Indeed, most introductory computer science courses are structured around cultivating critical thinking and problem solving abilities using logical reasoning [22, 23]. Further, many companies have committed a significant amount of resources in retraining programmers for efficient computational logic reasoning and programming (e.g., [24]). However, achieving uniformly satisfactory outcomes for computational logic reasoning training is difficult, given the varying levels of incoming preparation programmers possess [25, 26].

Given its importance to the field of computer science (both education and industrial practice), we are interested in investigating how programmers reason about computational logic. A foundational understanding of the cognition behind logical reasoning can help shed light on ways to help programmers reason about logic more effectively.[1] Note that while there exist several interpretations of logic (e.g., the study of correct reasoning or deduction in Philosophy [27]), in this thesis, we consider three facets of logic related to computation: *digital logic* [28] (e.g., hardware designs), *mathematical logic* [29] (e.g., proving properties about algorithms), and *programming logic* [30] (e.g., coding, manipulating data structures).

In the realm of psychology and cognitive science, *cognition* refers to the mental processes involved in acquiring knowledge [31] and comprises several processes, including attention, language, learning, memory, perception, and reasoning [32]. Over the years, researchers have investigated cognitive processes in a variety of contexts, including automobile driving [33], education [34], language [35], marketing [36], medical decision-making [37], office work [38], and team-based sports [39], working memory [40], among others. Such studies have furthered our understanding of how the human brain addresses different tasks, and what factors are associated with task success.

[1]We restrict the scope of this dissertation to obtaining a better understanding of the cognition behind logical reasoning, rather than investigating suggested pedagogical interventions. We advocate for such investigations as future work.

Within a computer science context, several studies investigating cognition for programming have helped researchers and educators better understand how we program [41, 42, 43, 44, 45, 46]. Many cognition studies of programmers have also shown the potential to inform pedagogy and guide tool development and retraining (see Floyd *et al.* [43, Sec. II-D] for a summary). This potential has been recently realized by several studies demonstrating that *cognitive interventions* (or changes in behavioral activity based on cognitive processes) can improve programming outcomes in introductory computing [47, 48, 49, 50, 51], encouraging additional work on understanding the cognitive basis of programming.

Given the potential impact of cognitive interventions on effective and efficient reasoning, this thesis probes a better understanding of programmer behavior and associated cognitive processes underlying logical reasoning in computer science (i.e., *how programmers understand computational logic*).

1.1 Approach

To obtain a deeper understanding of logical cognition, we desire a solution that satisfies the follow criteria:

- **Non-intrusive Methodology.** While medical imaging approaches like *Functional Near-Infrared Spectroscopy* (fNIRS) and *Functional Magnetic Resonance Imaging* (fMRI) have been successfully used to investigate user cognition in Computer Science [41, 42, 43, 44, 45, 46], such approaches often carry ecological validity concerns (e.g., a programmer inside an fMRI machine or connected to fNIRS equipment is performing computer science tasks in a non-traditional environment, study stimuli may be limited to a 30 second duration, etc.) [45, Sec. VI]. We propose a solution that allows for user cognition to be investigated non-intrusively (i.e., in a more natural setting) with minimal interference to user attention.

- **Objective Measures.** Research from both psychology and computer science has cast doubt on the trustworthiness of subjective measures [52, 53] (e.g., self-reporting). As such, we primarily favor objective measures to understand cognition for computational logic in a generalizable way.

- **Context-specific Models.** We desire an understanding of a variety of comprehension tasks for computational logic. Recent research has shown that user cognition varies for different computer science activities (e.g., different regions of the brain are correlated with code comprehension [43, 54] and code writing [46]). Instead of producing one overly-generalized model for all logical cognition, we propose to investigate context- or task-specific models associated with computational logic.

- **Incoming Preparation.** Software engineers entering a workforce often have different levels of incoming preparation or expertise [25], and this disparity in preparation is also reflected in students at the undergraduate level [26]. Since we aim to illuminate promising pedagogical interventions for follow-on investigation, we desire a solution that accounts for incoming preparation in the produced mathematical model (i.e., applies to students regardless of their levels of incoming preparation).

Previous research investigating user cognition for computer science activities does not satisfy one or more of the aforementioned desired properties. For instance, several studies investigating the cognitive processes behind programming have administered study stimuli inside a narrow and loud fMRI machine or with an array of external fNIRS connections attached to the participants [41, 42, 43, 44, 45, 46], calling into question the ecological validity of such experiments. By contrast, we desire an approach that allows the participants to complete the study tasks in a more traditional environment. Other studies investigating the psychology of programming (e.g., user behavior while programming [55, 56]) in more ecologically-valid settings often rely on self-reporting data or subjective measures that may not be adequately trustworthy [57]. Critically, many studies on user cognition for computer science tasks also do not elucidate how factors such as incoming preparation or expertise, among others, affect decision making in such programming tasks [58, 59, 60]. Since we aim to probe pedagogy based on student cognition, we desire an approach that explicitly accounts for incoming preparation in our models, and as such, can be applied to a wider variety of student groups.

We combine several insights to present a systematic study of programmer behavior and cognition for several tasks involving computational logic. **First**, we can implement objective, non-intrusive measures in a computer science context to establish correlations between variables. In particular, for this thesis, we hypothesize that with the emergence of technologies like high-precision *eye-tracking*, coupled with novel state-of-the-art *bug-finding algorithms*, it is possible to study user cognition for computation logic in a more ecologically-valid setting. **Second**, we can use medical devices in a computer science context to investigate causal relationships. In this thesis, we adapt the scientific approach and methodology behind *transcranial magnetic stimulation* from psychology and medicinal research, and apply it in a programming context for the first time to probe neural causality. **Finally**, we can use more advanced statistical rigor in computer science to account for student background and context. For this thesis, we hypothesize that using *multi-level regression models* can help us draw nuanced conclusions, and that *accounting for incoming preparation or expertise* as an explicit independent variable in our produced models can allow suggested pedagogical interventions to be applied to more students with different levels of preparation.

Combining these insights allows us to present the following three research components investigating cognition for computational logic:

(1) ***Does pointing programmers to the potential source of defects in a hardware design help their debugging efficacy?*** In the first research component, we develop the first automated program repair (APR) algorithm for hardware designs (i.e., digital logic) and investigate its use as a debugging aid for novice and expert programmers. Our study participants are shown defective hardware designs with partial and full debugging hints (see Section 3.3.3), and we compare their debugging efficacy against a baseline of no debugging information. We demonstrate the utility of the implementation of the algorithm as a debugging assistant using statistical tests on behavioral data collected from our participants. We hypothesize that our algorithm can be useful in a classroom context to help improve the debugging efficacy of programmers.

(2) ***How exactly do programmers locate incorrect parts of an algorithmic proof?*** In the second research component, we use eye-tracking to gather insights into student problem-solving strategies for proof or *formalism* comprehension tasks (i.e., mathematical logic). We further investigate any differences in strategies (e.g., visual attention on different parts of the proof) used by students with different incoming preparation for formal methods. Our participants are shown a series of algorithmic proofs and asked to identify any mistakes in the proofs. We use statistical tests on the eye-tracking, behavioral, and outcome data to understand and compare the strategies employed by students with different levels of incoming preparation. We hypothesize that understanding the strategies used by different students and their outcome success rates can help elucidate how educators can better prepare students for formal reasoning about algorithms.

(3) ***If we temporarily excite or inhibit the brain regions associated with manipulating 3D objects, does it impact programmers' ability to reason about code?*** In the final research component, we investigate the relationship between *spatial reasoning* (i.e., the ability, with well-studied cognitive processes, to mentally visualize and manipulate 3-dimensional structures) and program comprehension tasks (i.e., programming logic) using transcranial magnetic stimulation (TMS). Previous studies have found neuroscientific correlations between spatial visualization and various program comprehension tasks [45, 47], but a causal relationship between the two cognitive processes is yet to be established. We use TMS to stimulate two brain regions associated with spatial reasoning tasks and one control region before presenting participants with program comprehension tasks. We use statistical tests on the collected behavioral data to determine the presence of a casual link between brain activity for spatial reasoning and that for program comprehension. We hypothesize that such a causal link at the neurological level could spur changes in introductory programming education (e.g., including spatial visualization training) to improve student outcomes.

The overarching thesis statement for this work is:

> *It is possible to use objective measures to obtain mathematical models describing programmer behavior and cognition for computational logic reasoning tasks (with a focus on digital logic, mathematical logic, and programming logic), and these models can highlight prospective cognitive interventions for student training.*

This thesis includes, to the best of our knowledge, the first automated program repair algorithm aimed at diagnosing and repairing hardware design defects. We also present the first TMS study of programming, demonstrating the applicability of the technique to computer science research. To support open and reproducible science, we have made publicly available the source code and datasets (including our statistical analysis scripts) for results associated with all three research components.

1.2 Summary and Organization

The main contributions of this thesis include:

- An automated program repair algorithm for hardware designs (i.e., digital logic) and a mathematical model of the effects of its use as a debugging assistant for programmers;
- A mathematical model of the cognitive processes for formalism comprehension (i.e., mathematical logic) tasks using eye-tracking;
- A mathematical model for the neural connection between program comprehension (i.e., programming logic) and spatial reasoning using transcranial magnetic stimulation.

The remainder of this thesis is structured as follows. In Chapter 2, we outline key concepts and techniques used in this thesis for a general computer science audience. In Chapter 3, we present a study developing an automated program repair algorithm for hardware designs (i.e., digital logic) and a mathematical model of the effects of its use as a debugging assistant for programmers. In Chapter 4, we present a study developing a mathematical model of the cognitive processes for formalism comprehension (i.e., mathematical logic) using eye-tracking. In Chapter 5, we present a study developing a mathematical model for the connection between program comprehension (i.e., programming logic) and spatial reasoning using transcranial magnetic stimulation. In Chapter 6, we summarize the work in this thesis and discuss future research directions.

CHAPTER 2

Background

Prior to investigating cognition for computational logic reasoning, we first introduce key concepts and techniques for a computer science audience. First, we motivate and introduce the notion of automatically repairing software in Section 2.1. Second, we introduce the notion of hardware designs and contrast such designs to traditional software programs in Section 2.2. Next, in Section 2.3, we provide background, methods, and metrics associated with eye-tracking. Finally, we explain the approach behind transcranial magnetic stimulation in Section 2.4. More specific related works for each research component in this thesis, along with discussions on novelty, are included in the associated chapters.

2.1 Automated Program Repair for Software

Software programs are all around us: many everyday hardware devices require a software counterpart to function. The construction of software can often by modelled as a process known as the *software development life cycle.* The traditional software life cycle includes the following stages [61]:

- **Planning.** During planning, the software development team analyzes costs, schedules, and resources while gathering stakeholder (i.e., consumer, user, etc.) requirements to create a software requirement specification document.

- **Designing.** During the design phase, programmers analyze requirements, select optimal *algorithms* and solutions, choose technologies and tools, and plan integration into the existing IT infrastructure.

- **Implementing.** During the implementation phase, the development team translates the requirements into code, breaking each requirement down into smaller coding tasks that eventually progress towards the final product.

- **Testing and validation.** During this phase, the development team uses both automated and manual checks to ensure software quality and compliance with customer requirements. The code is often evaluated as it is developed, resulting in an overlap between the implementing and testing phases. Programmers often develop extensive *test suites* (i.e., a series of inputs to the software with known expected outputs to check functionality) that can be automatically run to test a program. Additionally, programmers may manually engage in *formal or mathematical reasoning* to ensure that the implemented code matches the requirements specification.

- **Deploying.** The deployment phase involves making the newly-implemented and tested code available to the customer for use.

- **Maintaining.** During maintenance, the team handles *bug fixes* (often via *debugging activities*), customer issues, and software changes. This stage also involves monitoring system performance, security, and user experience for continuous improvement. Bug fixes primarily refer to changes made to software by the development team to fix incorrect behavior.

The software industry is a major driving force behind the $5 trillion worldwide IT market [1], and software maintenance is the most time-consuming (and hence, costly) stage in the software life cycle, with around 60% and often up to 90% of the total cost of software attributed to the maintenance stage [62, 63]. Further, finding and fixing software bugs after deployment is often estimated to be 100 times more expensive than fixing bugs during the design and testing phase [64].

To lower the costs associated with software maintenance, significant research effort has been devoted to repairing bugs (or errors) automatically over the last decade [65, 66, 67]. *Automated program repair* (APR) usually takes as input source code with a deterministic bug (i.e., a bug that can be replicated with a given sequence of steps) and a test suite with at least one failing test that reveals the bug, and aims to automatically generate fixes to the buggy code. Most APR techniques operate on a tree representation of the source code of a program known as an *abstract syntax tree* (AST) [68].

Such test suite based repair, where test cases are used to guide the search for a patch, can be further divided into generate-and-validate and semantics-driven approaches:

- Generate-and-validate techniques produce candidate patches for the buggy code and evaluate them against the test suite to check if all tests pass [69, 70, 71, 72]. Figure 2.1 shows an overview of generate-and-validate APR techniques. Within the APR loop (gray box in Figure 2.1), *fault localization* is responsible for automatically implicating

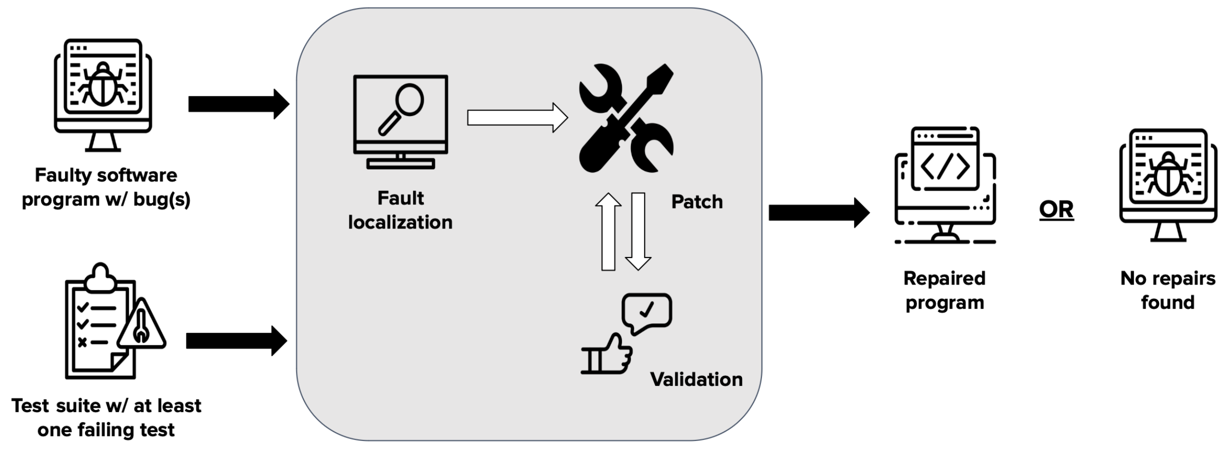

Faulty software program w/ bug(s)
Test suite w/ at least one failing test
Fault localization
Patch
Validation
Repaired program
OR
No repairs found

Figure 2.1: Overview of the generate-and-validate APR technique

lines of code likely responsible for the bug, *patch* corresponds to automatically applying an edit to the original program to repair the bug, and *validation* involves re-running test suites to assess program behavior.

- Semantics-driven approaches first extract constraints on a program based on test suite execution and then use these constraints to synthesize a patch [73, 74, 75, 76].

In this thesis, we focus on applying generate-and-validate software APR techniques to hardware designs, targeting the testing and maintenance phases of the design process.

2.2 Hardware Designs

As our reliance on electronic devices for everyday tasks increases, so does the number of computer chips around us. From microwaves and refrigerators in the kitchen to personal computing devices in the office to cars on the road, virtually every electronic item we use contains a tiny wafer of semiconducting material with an embedded electronic circuit [77], known as a *chip*. These silicon chips are typically manufactured in highly-controlled fabrication plants through a precise circuit etching process that transforms *digital hardware designs* into manufactured hardware.

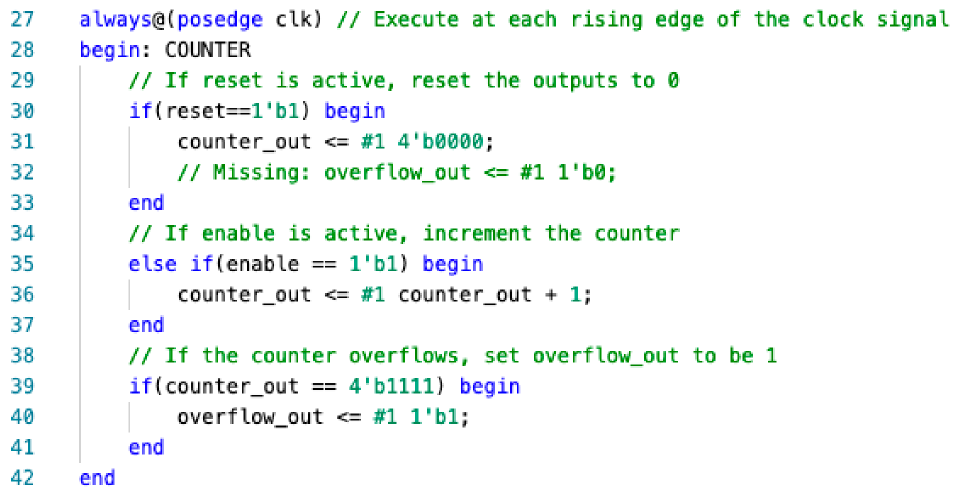
In modern engineering, the hardware design process typically includes producing such digital specifications (often using *hardware description languages*, or HDLs) for circuits that enable programmers to simulate and verify functionality of devices before the manufacturing process [78]. This design process is critical to get right, since errors are very difficult to correct once hardware is manufactured and shipped to consumers (e.g., [79, 80]). As such, there has been a significant amount of interest in ensuring defects in designs are caught and rectified before the fabrication process (see Section 3.6).

2.2.1 Properties of Hardware Designs

Hardware or HDL designs differ from software programs in two key ways. First, while software written in conventional languages such as C [81] and Java [82] is generally based around a serial execution model (where one line of code executes before the next, etc.), hardware designs are inherently *parallel* and often include non-sequential statements (e.g., since separate portions of hardware can operate simultaneously). Second, while software programs usually use test suites to evaluate functional correctness (see Section 2.1), where individual *test cases* (or units of tests) may pass or fail depending on the quality of the software, HDL designs use *testbenches* [78] — which are programs with documented and repeatable sets of stimuli — to simulate behaviors of a device (or design) under test.

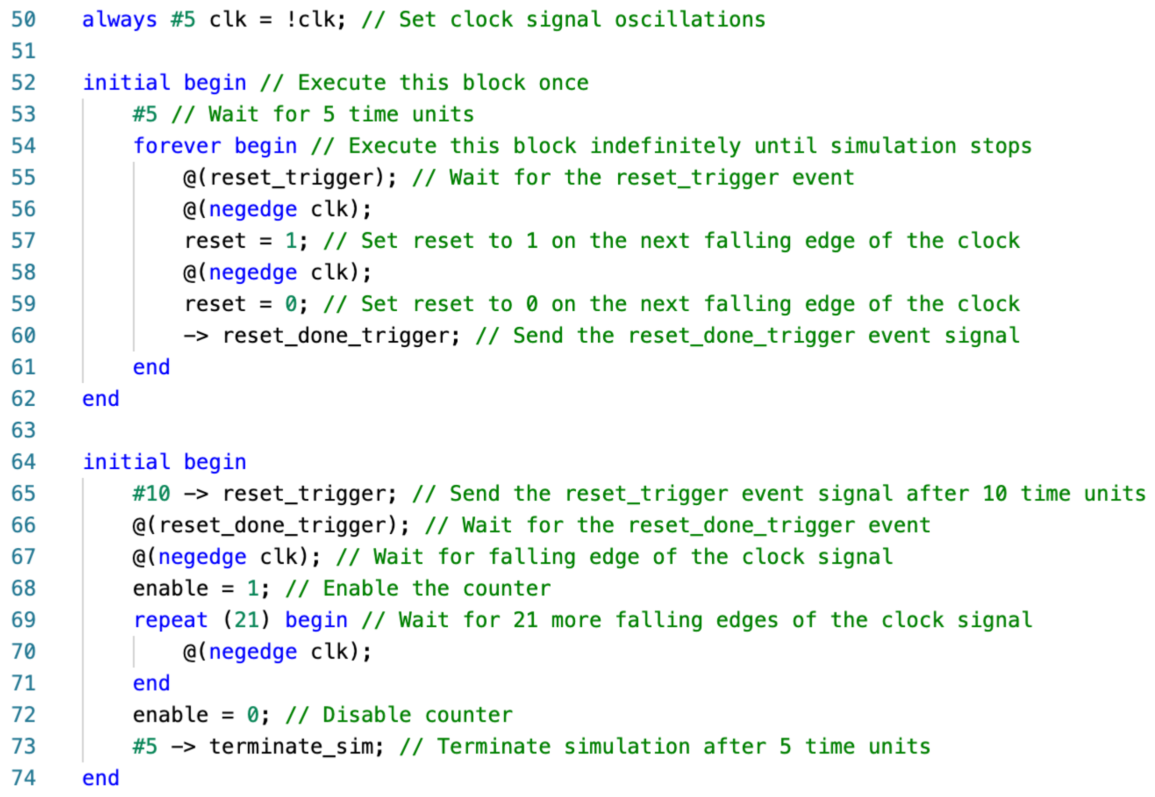
Consider the design for a faulty 4-bit counter with an overflow bit, implemented in Verilog shown in Figure 2.2. The design can keep track of an integer count using four bits, spanning a decimal value from 0 through 15. Once the count goes past 15 (`4'b1111` in binary), the counter should overflow and reset to 0 (`4'0000` in binary). The main block of the source code is shown in Figure 2.2a, with the corresponding testbench in Figure 2.2b. The circuit design uses variables `enable` and `reset` — representing *wires* in the circuit design — to increment (lines 35–37) and reset (lines 30–33) the counter respectively. Incrementing the counter when it has a binary value of `4'b1111` results in the overflow bit being set to true (lines 39–41). This implementation incorrectly manages the overflow bit: the if-statement at line 30 is missing an assignment that would reset `overflow_out`. Such defects can have serious consequences — integer overflow errors can be leveraged into significant security exploits (e.g., [83]).

The main block of the circuit design code shows an *always* block (line 27, Figure 2.2a) that executes repeatedly until the simulation stops. The execution of such blocks can only be triggered by changes to wires in the *sensitivity list* that follows the `always` keyword. Nearly every digital circuit design includes a *clock signal* (line 50, Figure 2.2b) that oscillates between a high and a low state (denoted by events `posedge` and `negedge` respectively); circuits rely on clock signals to know when and how to execute their programmed actions. A *clock cycle* is the period of time it takes for the clock signal to oscillate from high to low and back to a high state. For the 4-bit counter in Figure 2.2a, the wire `clk` (denoting the clock signal) is the only wire in the always block's sensitivity list (see line 27), and lines 28–42



```verilog
always@(posedge clk) // Execute at each rising edge of the clock signal
begin: COUNTER
    // If reset is active, reset the outputs to 0
    if(reset==1'b1) begin
        counter_out <= #1 4'b0000;
        // Missing: overflow_out <= #1 1'b0;
    end
    // If enable is active, increment the counter
    else if(enable == 1'b1) begin
        counter_out <= #1 counter_out + 1;
    end
    // If the counter overflows, set overflow_out to be 1
    if(counter_out == 4'b1111) begin
        overflow_out <= #1 1'b1;
    end
end
```

(a) Main block of the 4-bit counter with an overflow error



```verilog
always #5 clk = !clk; // Set clock signal oscillations

initial begin // Execute this block once
    #5 // Wait for 5 time units
    forever begin // Execute this block indefinitely until simulation stops
        @(reset_trigger); // Wait for the reset_trigger event
        @(negedge clk);
        reset = 1; // Set reset to 1 on the next falling edge of the clock
        @(negedge clk);
        reset = 0; // Set reset to 0 on the next falling edge of the clock
        -> reset_done_trigger; // Send the reset_done_trigger event signal
    end
end

initial begin
    #10 -> reset_trigger; // Send the reset_trigger event signal after 10 time units
    @(reset_done_trigger); // Wait for the reset_done_trigger event
    @(negedge clk); // Wait for falling edge of the clock signal
    enable = 1; // Enable the counter
    repeat (21) begin // Wait for 21 more falling edges of the clock signal
        @(negedge clk);
    end
    enable = 0; // Disable counter
    #5 -> terminate_sim; // Terminate simulation after 5 time units
end
```

(b) Main testing logic from the 4-bit counter testbench

Figure 2.2: A 4-bit counter with an overflow error in Verilog

are executed every time that wire reaches a high state. Note that there also exists a notion of asynchronous designs where the state of the system can change in response to changing inputs. However, given the increased complexity associated with asynchronous designs, most hardware designs tend to be synchronous in nature [84].

A key property of HDL designs not immediately apparent in Figure 2.2a is that parts of the design code typically execute in parallel. When a design is realized into actual hardware, individual components run all the time. Indeed, every statement in a Verilog design not inside an explicit sequential block of code exhibits concurrency. For instance, for the 4-bit counter in Figure 2.2a, an implementation managing the overflow bit correctly would include two assignments to `counter_out` and `overflow_out` (on lines 31 and 32 respectively) that logically happen at the same time when `reset` is true.

In this thesis, we exploit the traditional hardware design process and bridge the gap between software and hardware to introduce a new automated repair algorithm that works with parallel hardware designs. Our approach, outlined in detail in Chapter 3, can help designers catch and fix expensive mistakes before the design is manufactured into physical hardware and distributed to consumers.

2.3 Eye-tracking

There exist several ways to understand user cognition for reasoning tasks. One way to acquire such understanding in a controlled human study is to ask participants to self-report their subjective thoughts and experiences. Such self-reports, however, are often not trustworthy [52, 53]. A more trustworthy approach to probing cognition involves observing user behavior during task completion using objective measures (see Section 1.1). Unfortunately, such observational studies conducted in unconventional environments or via external equipment (e.g., wires, electrodes) connected to participants often raise ecological validity threats [85]. By contrast, when users work on given tasks, their eye movements can serve as an objective and non-interfering proxy for visual attention, and hence, cognition.

Eye-trackers are non-invasive, cost-effective, and easy-to-use devices that reliably measure visual attention and effort in variety of tasks [86], including human-computer interactions [87], software engineering [88, 89, 90], driving automobiles [91], marketing [92], and surgery [93].

Modern eye-tracking cameras measure and track a participant's eyes and use domain-specific algorithms to report *gaze data* that is then analyzed with respect to pre-defined *areas of interest* (or AOIs, corresponding to boundaries drawn around a visual feature or element to be investigated) in a stimulus. AOIs are typically manually defined by an experimenter

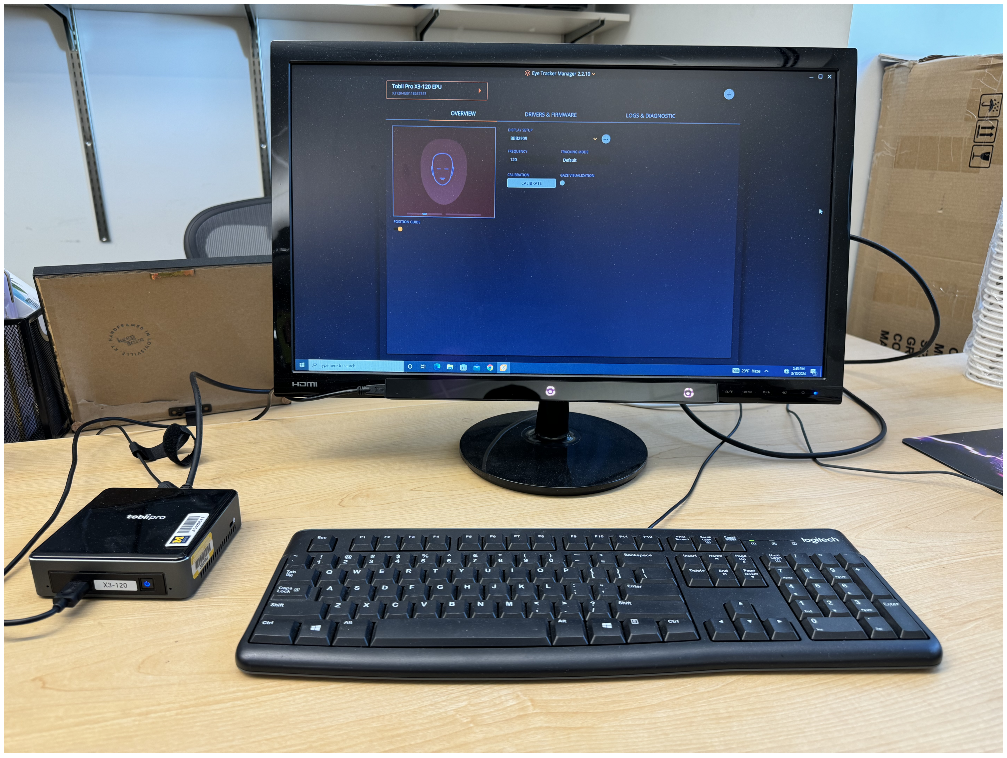


Figure 2.3: Tobii Pro™ eye-tracking setup with associated external processing unit

based on the nature of the study [94, 95].

2.3.1 How Eye-tracking Works

To collect gaze data, an eye-tracker uses high-definition cameras and an infrared light source to measure corneal reflection [96]. The infrared source casts a pattern of invisible infrared light on the user's eyes. A significant portion of the emitted light is reflected back, and this reflection is captured by the high-definition cameras. On the software front, image processing is applied to the captured camera data to determine the user's gaze point.

Screen-based eye-trackers are usually mounted at the top or bottom of a monitor, where they blend in with the monitor frame and do not interfere with the task being performed (see the bottom bezel of the monitor in Figure 2.3 for an indicative mounting location). Since most research-grade eye-trackers sample data at a high frequency, researchers may choose to use external processing units (center left on Figure 2.3) to offload the processing from the main computer and avoid performance degradation for study tasks (e.g., mouse or keyboard lag, applications freezing up, etc.).

For work in this thesis, we use the Tobii Pro™ X3-120 with an external processing unit to collect gaze data at a sample rate of 120Hz, and the Tobii Pro Lab [97] software to analyze collected data.

2.3.2 Eye-tracking Metrics and Key Terms

Two aspects of gaze data, based on ocular behavior, can clarify *cognitive load* (i.e., strain on working memory while processing information or completing a task) and task difficulty. A *fixation* is an eye gaze that lasts for approximately 200–300ms on a specific AOI and results in the focus of visual attention on the AOI. The majority of information processing for humans occurs during fixations [98, 96] and a small number of fixations usually suffices for a human to process a complex visual input [94, 99]. As such, fixation data is widely used to measure cognitive load for different tasks, with longer fixations and higher numbers of fixations indicating higher cognitive load [90, 89]. A *saccade* is a rapid eye gaze movement (40–50ms) that occurs between fixations on AOIs, and often does not correspond to cognitive processing [99, 98]. The *regression rate* is the ratio of backward or regressive saccades (e.g., leftward in left-to-right text reading) to the total number of saccades, and higher regression rates often indicate increased difficulty in performing and completing a task [100]. Finally, the *attention switching* metric depends on fixation counts and measures the total number of switches between AOIs, and can approximate the dynamics of visual attention during a task [90].

Modern eye-trackers can also report the *pupil diameter* of users. Pupil diameter has been used in the context of eye-tracking to approximate the cognitive load for users working on study tasks [101, 102, 103], with higher pupil diameters under controlled experimental conditions (e.g., manipulating cognitive load through task difficulty) indicating increased cognitive load [104].

In this thesis, we advocate for the use of eye-tracking to measure cognition for computational logic reasoning tasks. In particular, in Chapter 4, we investigate cognition for formalism (or proof) comprehension tasks (i.e., mathematical reasoning about algorithms) via eye-tracking.

2.4 Transcranial Magnetic Stimulation

In recent years, the software engineering community has increasingly used medical neuroimaging (e.g., fMRI, fNIRS) to non-invasively measure brain activity and understand the cognitive processes behind programming [41, 42, 43, 44, 45, 46]. Such studies have identified cognitive processes correlated with various programming tasks. For example, many of the neuroimaging studies in computer science have found connections between programming and reading [43] or spatial visualization [45, 105], two skills with well-understood cognitive structures. Note, however, that medical imaging studies often only provide confidence in correlations between variables, not causative links.

Transcranial magnetic stimulation (TMS) is a safe and noninvasive technique that is well-established for a variety of clinical and scientific use cases [106]. Clinically, TMS is used as a treatment for major depressive disorder [107], smoking cessation [108], and obsessive-compulsive disorder [109], among others. TMS is a well-studied research tool: during 2014–2024, the National Library of Medicine has recorded over 1000 academic papers published each year which investigate the use of TMS.

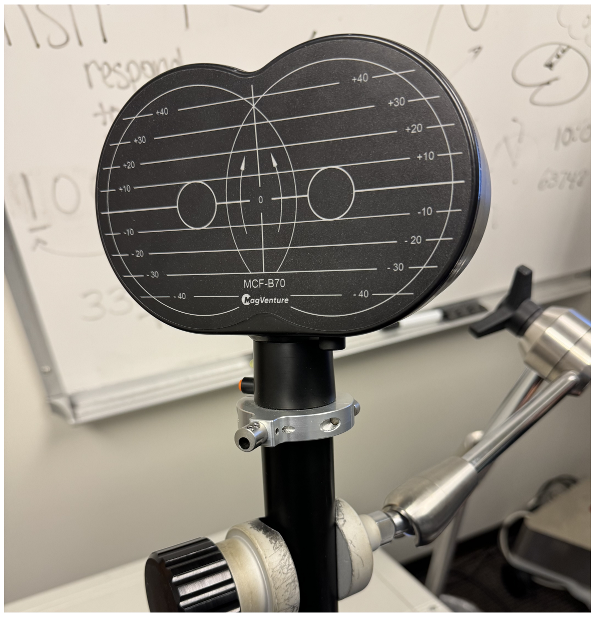
Compared to other methods, TMS is a time-efficient way to investigate the *causative* link between neural activity and programming ability. Other medical approaches that affect the brain in specific areas tend to be invasive, often requiring implanted electrodes, drug treatments, or neurological surgeries. By contrast, non-medical approaches such as transfer training or pedagogy are typically studied over a longer period of time (cf. [47]). Using TMS also removes variance and potential confounds which may be present in a study extending over a long period of time (e.g., changing physical or mental states of participants). By disrupting brain activity using this neurostimulation technique and then measuring behavioral outcomes (e.g., timing, accuracy, etc.) on programming tasks, we can observe a causal relationship between spatial reasoning and programming ability, should one exist.

2.4.1 How TMS Works

Administering TMS involves the use of a stimulator (Figure 2.4a) sending periodic phases (or *pulses*) of electric current through a coil (Figure 2.4b). This current induces highly concentrated magnetic fields around the coil. When the TMS coil is placed on a human subject's head (see Figure 5.3 for an indicative example), the TMS pulses produce small electric currents in the brain, temporarily altering neural activity in the targeted brain region [110, 106]. Two trackers — one mounted below the coil and another mounted on the participant via a headband — are tracked by the corresponding infrared camera (Figure 2.4c). The software for TMS equipment allows a researcher to define targets (i.e., brain regions) for stimulation and provides real-time feedback on the placement and orientation of the coil for accurate administration of TMS. In a research context, after stimulation (lasting 40 seconds for our study; see Section 5.2.3), any equipment attached to the participant is removed and the participant is asked to complete study tasks on a regular computer. Behavioral data collected from the participant is later analyzed to infer causal relationships.

For work in this thesis, we use the MagVenture MagPro™ X100 stimulator with an MCF-

(a) TMS stimulator


(b) TMS coil

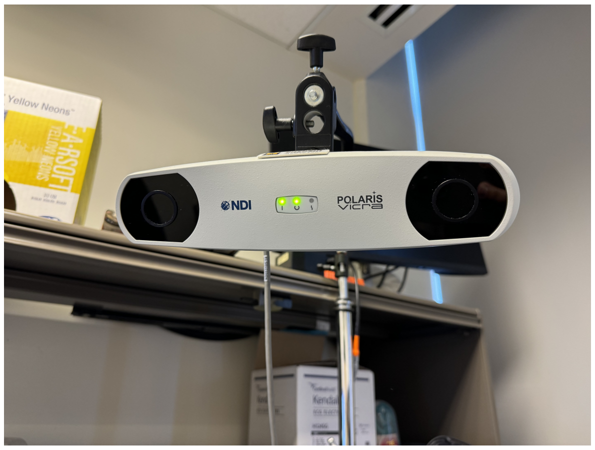

(c) TMS tracking camera

Figure 2.4: MagVenture MagPro™ TMS stimulator and coil with NDI Polaris Vicra™ tracking cameras

B70 butterfly coil and an NDI Polaris Vicra™ tracker. On the software front, we use the Brainsight® TMS neuronavigation developed by Rogue Research (Montréal, Canada).

2.4.2 TMS Metrics and Key Terms

A variety of pulse patterns and durations (or *protocols*) have been explored in the context of TMS [111]. For instance, *single-pulse TMS* can be used to explore brain functionality without lasting effects (e.g., stimulating a visual brain region to produce momentary sensations of light flashes), while *repetitive TMS* (rTMS) can be used to induce changes in brain activity outlasting the stimulation period (e.g., as a treatment for a neurological or psychiatric disorder, or as a research tool) [112]. *Theta burst stimulation* (TBS) is a type of rTMS that promises effects lasting up to 60 minutes and has been widely studied in the context of the human motor cortex (e.g., [113]).

To comply with established safety standards, prior to rTMS treatment, researchers often perform a *thresholding* step using single-pulse TMS to determine the appropriate stimulation intensity, or the *active motor threshold* (AMT), for each participant [114, 115] (see Section 5.2.3.2 for details on an established thresholding process).

In this thesis, we explore the first use of TMS in a programming context to investigate causal relationships at the neural level, demonstrating the applicability of the more ecologically-valid medical technique to computer science research. For our TMS application, we use well-studied protocols for participant thresholding (single-pulse TMS) and neurostimulation treatment (TBS).

Having introduced key concepts and techniques used in this thesis, in the next chapter, we present our first research component. In particular, we explore developing an APR algorithm for hardware designs (i.e., digital logic), and investigate its use as a debugging aid for programmers.

CHAPTER 3

Automated Program Repair for Hardware Designs

Increases in the complexity of hardware designs (i.e., digital logic) over the past few decades have challenged the ability of programmers to find and repair defects in circuit descriptions [116]. While significant effort has been devoted to efficiently verifying functional correctness in hardware design descriptions, relatively little work has been done in patching defects in such descriptions automatically. By and large, debugging and repairing hardware designs remains a very expensive and time-consuming task [8]. Indeed, recent functional and security vulnerabilities due to defects at the hardware design level have led to expensive consequences [80, 79]. To reduce the cost and improve the maintenance of hardware designs, a solution needs to not only precisely identify sources of defects in real-world hardware descriptions, but also automatically produce repairs implementing correct functionality of the circuit designs. These repairs can then be shown to programmers for validation before moving on to the synthesis phase. Additionally, we desire a solution that makes use of readily-available resources that are part of hardware design to validate proposed repairs.

Previous work has attempted to address this problem but may not satisfy these characteristics of a desired solution. For instance, some techniques automatically localize defects in design source code but suffer from high false positive rates [117, 118]. Other approaches for automatic error diagnosis and correction require formal specifications to conduct design verification [119], which usually do not scale to large designs. Furthermore, previous work does not operate on behavioral-level descriptions of hardware circuits (i.e., higher-level design without specifying internal structure in detail) [120, 121].

On the other hand, in the realm of software, significant research effort focuses on repairing bugs automatically [65, 66, 67]. Automated program repair (APR) algorithms fix defects in software by producing patches that pass all test cases while retaining required functionality. Traditional APR for software employs fault localization techniques (see Section 2.1) to implicate faulty code, and such techniques are often crucial to the success of program repair. Interest in applying software APR methods to hardware has been seen in the literature [122, 123, 124, 125], but to limited fruition.

While both software programs and hardware description languages (HDLs) share programming concepts like expressions, statements, and control structures, suggesting the possibility of repurposing software repair techniques to hardware designs, we highlight two key differences between the two domains: (1) HDL designs are inherently parallel and often include non-sequential statements, since separate portions of hardware can operate simultaneously, and (2) Software programs usually use test cases to evaluate functional correctness, where individual test cases may pass or fail depending on the quality of the software. HDL designs, on the other hand, use testbenches, which are programs with documented and repeatable sets of stimuli, to simulate behaviors of a device under test. In both academia and industry, the majority of digital hardware design is done using such HDLs.

In the first research component of this thesis, we aim to close the gap between software APR and the hardware design process. In particular, we develop an APR algorithm for hardware designs (i.e., digital logic) and investigate its use as a debugging aid for novice and expert programmers.

3.1 Overview of Experimental Design, Results, and Contributions

In this chapter, we present two key insights to bridge the gap between software repair techniques and hardware designs. We first hypothesize that while traditional *spectrum-based fault localization* approaches (that assign blame to faulty code based on the lines of code executed by passing and failing test cases; e.g., [126]) do not apply to hardware designs that feature a more parallel structure [127], certain *dataflow-based fault localization* approaches (that use information about the possible values and relationships of data in a program during execution; e.g., [128]) work well in this domain. Second, we hypothesize that a traditional hardware testbench can be instrumented to admit observations for candidate patches that guide the search for APR.

Leveraging these insights, we present CirFix, a framework for automatically repairing defects in hardware designs implemented in languages like Verilog, one of the most popular HDLs [129]. CirFix uses *genetic programming* (GP), an iterative stochastic search technique, to find candidate repairs for defects in hardware designs. CirFix also makes use of readily-available artifacts in the hardware design process (e.g., testbenches, simulation environments) to diagnose and repair defects in a circuit description. We propose an approach to guide the search for a repair by instrumenting hardware testbenches to record the values of output wires at specified time intervals during a simulation of the circuit design. Our novel fault localization utilizes the simulations to assign blame — or suspiciousness values — to incorrect *wires* and *registers* (analogous to program variables for software). CirFix then performs a bit-level comparison of output wires against information for expected behavior to assess functional correctness of candidate repairs. CirFix employs a fixed point analysis of assignments made to internal registers and output wires to implicate statements and reduce the search space, enabling our approach to scale to large circuit designs in industry.

We present a benchmark suite of 32 *defect scenarios* [69] based on three hardware experts — two from industry and one from academia — asked to transplant bugs they observed in real life into 11 different Verilog projects. CirFix can produce testbench-adequate repairs for 21 out of the 32 Verilog defect scenarios within reasonable resource bounds, of which 16 are deemed correct upon manual inspection.

Furthermore, we evaluate the usability of our novel fault localization algorithm independent of the automated repair context through a human study in which 41 programmers assess its quality and usefulness. We find a statistically-significant preference ($p \leq 0.003$) for CirFix fault localization as a debugging aid in fixing multi-line hardware defects, primarily in student applications ($p \leq 0.02$).

The main contributions of this chapter — separately published in the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems [130] and IEEE Transactions on Software Engineering [131] — are:

- CirFix, a repair algorithm for hardware designs.
- A novel dataflow-based fault localization approach for HDL descriptions to implicate faulty design code.
- A novel approach to guide the search for a hardware design repair that is compatible with the testbench-based hardware testing process.
- A new publicly available benchmark suite of 32 scenarios, based on proprietary bugs but available in 11 open projects.
- A systematic evaluation of CirFix on our benchmark suite. CirFix was able to correctly repair 16 out of the 32 Verilog defects under consideration.
- A human study ($n = 41$) using CirFix's fault localization algorithm as a debugging aid on real-world and student applications. We observe statistically significant preference using the support for multi-line defects ($p \leq 0.003$) in student applications ($p \leq 0.02$).

Algorithm 1 The high-level CirFix pseudocode.

```
Input:  Circuit design to be repaired, $C$.
Input:  Instrumented testbench for circuit, $TB$.
Input:  Expected output for circuit behavior, $O$.
Input:  Fitness function, $f$.
Input:  Parameters, $popSize, maxGens, rtThreshold, mutThreshold, eliteThreshold$.
Output: Repaired circuit description.
 1: $pop \leftarrow \mathsf{seed\_pop}(C, popSize)$
 2: repeat
 3:     $childPop \leftarrow \mathsf{elitism}(pop, eliteThreshold)$
 4:     while $|childPop| \leq popSize$ and $\forall$ $candidate \in childPop.$ $f(candidate, TB, O) < 1.0$
        do
 5:         $parent \leftarrow \mathsf{tournament\_selection}(pop, f)$
 6:         $fl\_set \leftarrow \mathsf{fault\_loc}(parent)$
 7:         if $\mathsf{probability}() \leq rtThreshold$ then
 8:             $child \leftarrow \mathsf{apply\_fix\_pattern}(parent, fl\_set)$
 9:             $childPop \leftarrow childPop \cup \{child\}$
10:         else                                                   ▷ Repair operators
11:             if $\mathsf{probability}() \leq mutThreshold$ then
12:                 $child \leftarrow \mathsf{mutate}(parent, fl\_set)$
13:                 $childPop \leftarrow childPop \cup \{child\}$
14:             else
15:                 $parent2 \leftarrow \mathsf{tournament\_selection}(pop, f)$
16:                 $\{c1, c2\} \leftarrow \mathsf{crossover}(parent, parent2)$
17:                 $childPop \leftarrow childPop \cup \{c1, c2\}$
18:     $pop \leftarrow childPop$
19: until resources exhausted or $\exists$ $candidate \in childPop.$ $f(candidate, TB, O) = 1.0$
20: if resources exhausted then return
    $candidate \in childPop.$ $\exists$ $other \in childPop \Rightarrow f(candidate, TB, O) \geq f(other, TB, O)$
21: return $\mathsf{minimize}(candidate, TB, O)$
```

3.2 Technical Approach

In this section, we present CirFix, an automated repair algorithm for defects in hardware design code. Our prototype implementation of CirFix operates on hardware descriptions written in Verilog, and thus supports HDL programming constructs such as sequential and parallel code, variable reassignment, and synchronized code blocks. Our prototype would require modifications to generalize to other hardware description languages (e.g., adding support for AST parsing or different simulation environments). The pseudocode is shown in Algorithm 1.

CirFix applies our two-pronged HDL-specific approach to implicate faulty design code and assess the correctness of circuit descriptions to produce repairs that can then be shown to programmers for review. Our *fault localization* approach simulates a faulty circuit and assigns blame to incorrect wire and register outputs (line 6 in Algorithm 1; see Section 3.2.1). Note that while traditional software-based APR techniques typically compute fault localization once at the start of the search for repairs, we choose to repeatedly re-localize to support multiple dependent edits made to the source code. Our *fitness function*, tailored to the hardware domain, scores each candidate patch to guide the search for repairs (lines 4 and 18 in Algorithm 1; see Section 3.2.2).

At a high level, CirFix uses genetic programming (GP) [132], an iterative stochastic search technique, to synthesize candidate repairs to buggy HDL programs. The framework takes as input the source code implementing a faulty circuit design, an instrumented testbench used to simulate the circuit for testing and verification purposes, the expected circuit behavior,[1] and the input parameters. The algorithm starts with the original source code and maintains a population of program variants, each stored as a *repair patch* [71] describing a sequence of abstract syntax tree edits parameterized by unique node numbers. Each program variant is obtained by applying a *repair operator* (lines 12 and 16 in Algorithm 1; see Section 3.2.3) or a *repair template* (line 8 in Algorithm 1; see Section 3.2.3) to a parent selected for reproduction. Candidate variants are selected for reproduction based on their *fitness* scores assigned by the CirFix fitness function (line 5 in Algorithm 1; see Section 3.2.4). Our *fix localization* identifies code to be inserted or replaced as part of mutation operations (see Section 3.2.5). The algorithm loops for several *generations*, each maintaining a population of program variants, until a *plausible repair* is found that produces output (as observed by the instrumented testbench) matching the expected circuit output, or allowed resources are exhausted (i.e., the algorithm reaches a timeout or a certain number of generations). If allowed resources are exhausted, the algorithm returns the highest fitness individual in the terminating generation (line 20 in Algorithm 1). For the final post processing step, CirFix *minimizes* [133] a candidate repair to remove extraneous operations not needed to obtain correct circuit output (line 19 in Algorithm 1; see Section 3.2.6). Candidate repairs are not deployed directly but are instead shown to programmers for validation before the design is ultimately synthesized, reducing maintenance costs [134, 135].

Algorithm 2 High-level algorithm for fault localization for HDL based on a fixed point analysis of assignments.

Input: Faulty circuit design code AST, *ast*.
Input: Simulation output, $S : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$.
Input: Expected output, $O : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$.
Output: Fault localization set, FL.

```
1:  $FL, mismatch \leftarrow \emptyset, \emptyset$
2:  $mismatch' \leftarrow \mathsf{get\_output\_mismatch}(O, S)$        ▷ Section 3.2.2
3:  while $mismatch \neq mismatch'$ do
4:      $mismatch \leftarrow mismatch \cup mismatch'$
5:      for $node$ in $ast$ do
6:          if $\mathsf{implicated}(node, mismatch)$ then
7:              $FL \leftarrow FL \cup \{node.id\}$
8:              for each $child$ of $node$ do
9:                  $FL \leftarrow FL \cup \{child.id\}$
10:                 if $\mathsf{type}(child) = \mathsf{Identifier}$ and $child \notin mismatch$ then
11:                     $mismatch' \leftarrow mismatch' \cup \{child\}$
12: return $FL$
```

3.2.1 Fault Localization

Fault localization is critical to the success and efficiency of APR [136]. Traditional APR for software often relies on spectrum-based fault localization [126] to narrow down defects to certain parts of a faulty program by sampling the program counter. Such fault localization approaches do not extend naturally to the parallel structure of hardware descriptions [127].

To overcome this challenge, we propose a novel dataflow-based fault localization approach to implicate faulty code in HDL descriptions. Previous work analyzing defects in large hardware projects reports that most defects in Verilog descriptions correspond to assignment statements and if-statements [137]. We present an algorithm that implements an analysis of assignments made to wires and registers in a circuit's design code to implicate faulty statements. Our proposed algorithm transitively captures data and control dependencies in a context-insensitive fixed point analysis. While traditional spectrum-based fault localization approaches for software return a ranked list of implicated statements [138, 139, 140], our approach returns a uniformly-ranked set: due to the parallel structure of HDL designs, a set of implicated assignments that are equally likely to contribute to the design defect suffices.

Algorithm 2 outlines the high-level pseudocode for our fault localization approach. The algorithm takes as input the AST of the faulty circuit design, the output from design simu-

[1]CirFix does not require perfect information for expected behavior for every timestep: the programmer can choose to only provide information at certain intervals. See prior work RQ4 [130] for an evaluation of the trade-off between the level of detail of expected output and repair success.

lation, and the expected circuit behavior (see Section 3.2.2 for the simulated and expected outputs). It then compares the simulation output against the expected behavior to produce a set of *identifiers* (i.e., variable names) for output wires and registers with mismatched values. Using this mismatch set as a starting point, for every node in the AST, the algorithm checks if the node is implicated by output mismatch[2]. Implication for a node in the AST occurs when

- (Impl-Data): either the node corresponds to an assignment statement and the left child of the node corresponds to an identifier in the mismatch set (cf. data dependency analysis),
- (Impl-Ctrl): or the node corresponds to a conditional statement and an identifier in the conditional statement belongs to the mismatch set (cf. control dependency analysis).

Any implicated node and all of the node's children are added to the fault localization set. Additionally, if any child of an implicated node is itself an identifier not part of the mismatch set, the name of the identifier is added to the mismatch set (Add-Child). For example, for the 4-bit counter introduced in Section 2.2.1, recall that the `overflow_out` wire had incorrect output from the circuit simulation. This causes the wire to be added to the mismatch set. The CirFix fault localization implicates the only assignment to `overflow_out` (line 40, Figure 2.2a) by rule (Impl-Data) in the first iteration of the algorithm. Indeed, the entire if-statement wrapping this assignment (line 39, Figure 2.2a) becomes implicated by (Impl-Ctrl), bringing in the new identifier `counter_out` to the mismatch set by (Add-Child). The process is repeated until no new identifiers are added to the mismatch set.

This novel approach to fault localization for hardware is a good fit for automatically repairing HDL designs: it returns a precise set of implicated AST nodes in a faulty circuit design, is context-insensitive and therefore inexpensive to compute, and applies directly to node types associated with ASTs for languages like Verilog. Note that while we demonstrate the scalabaility of our approach on a variety of hardware designs of different sizes (see Table 3.2), our approach may require additional programmer effort to generalize to very complex designs (e.g., a microprocessor) with millions of wires, gates, and registers.

3.2.2 Fitness Evaluation

The *fitness function* evaluates the acceptability of a program variant by assigning a value ranging continuously between 0 and 1 to the variant, with 1 indicating a *plausible* [72] (i.e.,

[2]In a focused investigation of our our three largest benchmarks, both control flow complexity and also the number of wires/registers were found to contribute equally (40–50% each) to the final fault localization size, and thus the scalability of our algorithm.

testbench-adequate) repair ready to be shown to programmers. Fitness provides a termination criterion for CirFix and guides the search for a repair. As mentioned in Section 2.1, traditional APR for software uses test-case based evaluation strategies to assess candidate repairs. Hardware designs, by contrast, use testbenches to verify functional correctness. We present a novel fitness function tailored to hardware to guide the search for repairs to HDL designs. Our fitness function uses two key insights: *visibility* and *comparison*.

Many traditional hardware testbenches monitor the values of output wires during simulation and assess correctness based on the final output values. For instance, the testbench for the 4-bit counter introduced earlier (Figure 2.2b) may report that the final value of the counter is 5 and the overflow bit is 1 when the simulation terminates. Some off-the-shelf hardware testbenches, especially those for large projects, may not even report the exact incorrect value, reporting instead merely the presence or absence of an error during simulation. We want our fitness function to assess a candidate repair based on intermediary as well as final output values, and assign fitness values to the repair based on its overall closeness to the correct circuit design [141]. To do so, given a testbench for a faulty HDL description, we instrument the testbench to record the values of output wires and registers for specified time intervals. This instrumentation is easily automatable: every hardware testbench must instantiate a device under test and connect wires to the module being instantiated (cf. unit tests in software instantiating the object being tested); each module in turn specifies input and output wires, and a static analysis of the instantiation of the device under test can provide the information needed to instrument a testbench automatically.

Once the testbench is instrumented, we simulate the circuit design and compare the results against the expected output (see Section 3.3.1.2 for a discussion on obtaining correct circuit behavior) to assess functional correctness of the HDL description. We desire a fitness function that assigns high values to candidate repairs that display behavior similar to expected behavior. To do so, we need to determine the relative contribution of each bit to the fitness of a proposed repair. Given a set of time steps $Time$, a set of output wires and registers Var, a simulation result $S : \mathit{Time} \rightarrow \mathit{Var} \rightarrow \{0, 1, x, z\}$, and expected output $O : \mathit{Time} \rightarrow \mathit{Var} \rightarrow \{0, 1, x, z\}$, where x or z correspond to unknown logic value and high impedance respectively, for timestamp $c_i \in \mathit{Time}$, we sum over the $n = |S(c_i)|$ output bits of the circuit. We compare the expected value for wire b from clock cycle c_i, $O_{c_i,b} = O(c_i(b))$, against the actual value from the simulation result, $S_{c_i,b} = S(c_i(b))$. If the bits match, we add to the fitness sum of the circuit; if the bits differ, we subtract from the fitness. An additional penalty weight φ is assigned to bits with values of x (uninitialized) or z (high impedance).

The fitness sum, $\mathit{sum}(S, O)$, and total possible fitness, $\mathit{total}(S, O)$, are defined as follows, where _ represents a bit value of 0 or 1:

$$sum(S,O) = \sum_{c_i=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{c_i,b}, S_{c_i,b}) \in \{(0,0),(1,1)\} \\ \varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(x,x),(z,z)\} \\ -1 & (O_{c_i,b}, S_{c_i,b}) \in \{(1,0),(0,1)\} \\ -\varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(\_,x),(x,\_),(z,\_),(\_,z)\} \end{cases}$$

$$total(S,O) = \sum_{c_i=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{c_i,b}, S_{c_i,b}) \in \{(0,0),(1,1),(1,0),(0,1)\} \\ \varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(\_,x),(x,\_),(x,x), \\ & \qquad (z,\_),(\_,z),(z,z)\} \end{cases}$$

The normalized fitness of the circuit is then defined as:

$$fitness(S,O) = \begin{cases} 0 & sum(S,O) < 0 \\ \frac{sum(S,O)}{total(S,O)} & sum(S,O) \geq 0 \end{cases}$$

This novel approach to calculating normalized fitness is effective at capturing whether or not a candidate design is close to the correct implementation of the circuit, and at guiding the search for a repair.

3.2.3 Repair Templates & Repair Operators

A *repair template* for a defect in code is defined as a pre-identified pattern that can be applied to some aspect of the code to fix the defect. The idea of using templates for APR is well-studied for software [142, 143, 144]. We apply repair templates to aid CirFix in its search for repairs. We propose nine repair templates corresponding to four defect categories for HDL designs. Of the four defect categories we consider, three are suggested in previous work by Sudakrishnan *et al.* [137] that analyzes the bug fix history of four hardware projects written in Verilog and presents several commonly-occurring fixes for HDL descriptions; we propose the remaining defect category based on our experience with defects in hardware designs. The repair templates in CirFix are presented in Table 3.1. Incorrect conditionals, sensitivity lists, and assignments correspond to the three most commonly occurring defects in the four hardware projects analyzed in previous work [137, Tab. 2]. Note that our repair templates focus on correct behavior from circuit designs during simulation (cf. rules targeting synthesizability [145]). For an incorrect conditional for a program branch (e.g., the condition for a while-loop or an if-statement), our repair templates can negate the conditional.

CirFix uses two standard repair operators from well-known software repair approaches [69, 70, 146], mutation and crossover, to search the nearby space of circuit designs to produce a repair and to avoid local optima. The input parameter *mutThreshold* (line 11, Algorithm 1)

Table 3.1: Repair templates in CirFix

Defect Category	**Pattern Description**
Conditionals	Negate the conditional of a code block (e.g., if-statement, while-loop)
Sensitivity Lists	Trigger an always block on a signal's falling edge Trigger an always block on a signal's rising edge Trigger an always block on any change to a variable within the block Trigger an always block when a signal is level
Assignments	Change a blocking assignment to non-blocking Change a non-blocking assignment to blocking
Numeric	Increment the value of an identifier by 1 Decrement the value of an identifier by 1

tunes the relative application of mutation and crossover.

As in common software APR approaches (e.g., [69, Sec. III-F]), the mutation operator itself can be characterized into three subtypes: *replace*, *insert*, and *delete*. The `mutate` function of the CirFix framework generates a random probability value and employs the user-provided replace, insert, and delete thresholds to choose a mutation sub-type. The replace operator picks a random node from the fault localization space and replaces the node with another randomly chosen node from the corresponding fix localization (see Section 3.2.5) space. The insert operator picks a random node from the fix localization space and inserts it after another randomly picked node within a code block. The delete operator picks a random node from the fault localization and replaces it with an empty node — this operation is equivalent to deleting certain statements from the program variant under consideration.

CirFix uses the standard single-point crossover [147], which picks a *crossover point* for each of the two parents. Edit operations to the right of that point are swapped between the two parents. This results in two children program variants, each carrying some information from both parents. The crossover operator plays a key role in avoiding local optima when searching for high-fitness patches.

3.2.4 Selection

Automated program repair techniques based on GP use *selection* to choose parent variants from a population based on fitness. *Tournament selection* [148], a selection approach that selects a random pool of t program variants in a population and selects the fittest member of this pool as the parent, has been used widely for software-based APR [149, 69, 150, 70]. CirFix uses tournament selection to select a parent variant to transfer genetic information to the next generation as a child variant. The top $e\%$ fittest program variants from the previous generation are automatically included in the next generation (line 3 in Algorithm 1), a process known as *elitism* [151, 152].

3.2.5 Fix Localization

Given that fault localization has identified faulty design code to be changed, our *fix localization* provides some guidelines on how to perform the changes. We use fix localization to restrict the scope of the insert and replace operators to reduce the number of syntactically-invalid mutants (cf. [153]).

For the insert operator, we propose to only use statements types (e.g., conditional statements, assignments, etc. — see Annex A.6.4 in the IEEE Standard for Verilog [154] for the full Backus-Naur form definition of statement types) as the sources for insertion code. We further allow such statements to be inserted only into `initial` or `always` blocks, since such statements inserted elsewhere violate the syntax of Verilog [154, Annex A.6.2]. For the replace operator, we design CirFix such that an item in a Verilog module [154, Annex A.1.4] can be replaced either by another item of the same type, or by an item sharing the same immediate parent type (as specified in the formal syntax definition of Verilog [154, Annex A]).

Our fix localization approach reduces the average number of mutants producing compilation errors in our prototype from 35% to 10%. This reduction is comparable to that of fix localization techniques in software (e.g., [69]).

3.2.6 Repair Minimization

During the search for a repair, CirFix might produce edits to the code that do not contribute to the repair (e.g., repeated assignment statements within an `always` block). Such edits do not increase the fitness of the candidate repair, but they could introduce inefficiencies in the final circuit design or affect the design's readability [155].

CirFix removes such extraneous edits in a postprocessing *minimization* step by finding a subset of the edits in a repair patch from which no further elements can be dropped without causing a reduction in the fitness of the patch. As in APR for software (e.g., [69]), we use the delta debugging algorithm [133] to efficiently (i.e., in polynomial time) compute this *one-minimal* subset of the repair patch. The minimized set of repairs is then converted back into HDL code implementing the hardware design correctly.

3.3 Experimental Setup

This section describes the experimental setup for our evaluation of CirFix, including the construction of our new benchmark suite, our choice of experimental parameters, and our human study on evaluating the usability of CirFix's novel fault localization.

For our prototype implementation of CirFix, we use the open-source PyVerilog toolkit [156] (version 1.2.1, modified to support numbering for each node type) to parse a Verilog description of a circuit and produce an AST representing the circuit design code. We use Synopsys VCS [157], the primary hardware verification tool used by a majority of the world's top-twenty semi-conductor companies [158], to simulate the code using a manually instrumented testbench to assess functional correctness of the circuit design. Our prototype for CirFix is implemented using Python 3.6.8 and is made publicly available on GitHub (https://github.com/hammad-a/verilog_repair).

3.3.1 Benchmark Suite for Hardware Defects

For our evaluation of CirFix, we desire a benchmark suite consisting of faulty hardware designs that are indicative of defects in industry, comprise a wide range in terms of project size, and correspond to a variety of components found in real-world designs. To the best of our knowledge, there are no publicly available benchmarks that satisfy our requirements. Additionally, there is limited open source community support for industrial hardware designs, since such designs are often considered Intellectual Property (IP) of the stakeholder companies. As such, we propose to adapt the defect-seeding approach common in software [73, 159, 160] and present a benchmark suite of *defects scenarios* [136, 69] — each consisting of a circuit design, an instrumented testbench for the design, information for correct circuit behavior, and an expert-transplanted defect from real-life experience — to be used for the evaluation of automated repair techniques for hardware.

3.3.1.1 Selecting Hardware Projects

Every defect scenario includes a base circuit design and a testbench, as introduced in Figure 2.2. We required circuit designs with an available testbench and that admit simulation using the Synopsys VCS tool without any changes to the design code. This is a common requirement comparable to the benchmarks suites for APR in software [69, Sec. IV-A] [161, Sec. 3.1]. The hardware projects for our benchmark suite are presented in Table 3.2. For

Table 3.2: Benchmark hardware projects in our experiments. Project and testbench sizes are measured by source lines of code as reported by the Unix `wc` command.

Project	**Description**	**Project LOC**	**Testbench LOC**
decoder_3_to_8	3-to-8 decoder	25	56
counter	4-bit counter with overflow	56	135
flip_flop	T-flip flop	16	39
fsm_full	Finite state machine	115	66
lshift_reg	8-bit left shift register	30	44
mux_4_1	4-to-1 multiplexer	19	51
i2c	Two-wire, bidirectional serial bus for data exchange between devices	2018	482
sha3	Cryptographic hash function	499	824
tate_pairing	Core for the Tate bilinear pairing algorithm for elliptic curves	2206	983
reed_solomon_decoder	Core for Reed-Solomon error correction	4366	148
sdram_controller	Synchronous DRAM memory controller	420	95
Total		**9770**	**2923**

each hardware project, we need an instrumented testbench to record output values for our fitness function. While the instrumentation process is automatable (see Section 3.2.2), we manually instrument the testbenches for our prototype. Each testbench instrumentation required under 10 lines of Verilog code, took at most 5 minutes of programmer time, and did not require any circuit-specific knowledge beyond that available in the testbench (i.e., identifier names of output wires and registers, and the clock cycle duration).

We choose six projects from undergraduate VLSI courses to be indicative of repairing a small component in hardware design. We augment this by choosing the remaining five projects from OpenCores (a popular website for open-source HDL designs) and GitHub collectively to be indicative of repairing the entirety of a large circuit design. Unlike some previous works that only use toy benchmarks for evaluation (e.g., [162, 119]), our benchmarks include a range of project sizes (in terms of source lines of code), and all projects — including those from courses taught at the undergraduate level — correspond to components found in real-world hardware designs. To satisfy our variety requirement, we include a project from each of the key cores listed on the OpenCores website for certified projects (i.e., arithmetic, communication, crypto, error correction, and memory).

3.3.1.2 Obtaining Information for Correct Circuit Behavior

CirFix requires information about expected behavior for a circuit design to assign fitness values to candidate repairs. In APR for software, guidelines for correct behavior often take the form of passing and failing test cases [67]. More generally, however, such information can be induced from a previous version of the design known to be functional [163, 164, 165, 166, 167, 168] or a combination of data mining and static analyses of the design [169, 170, 171, 172], or manually provided by the programmer [173, 174, 175, 176].

This so-called "oracle problem" [177] remains a challenging issue in general for hardware testing and automated repair: implicit, high-level test oracles (e.g., "the program does not divide by zero") used by APR tools for software do not typically carry over to hardware. Given that circuit designs exhibit parallelism and require synchronization against a clock signal [178], how a circuit design reaches a certain output is often equally important as the actual final output produced. As such, any hardware test oracles need detailed information about the intermediate values from design simulation, and it does not suffice to only use the output values from the simulation as correctness information for an approach like CirFix.

For our benchmark suite, we follow an established approach in APR for software [179, 65] and employ a previously-functioning version of the circuit design to record the expected behavior information for circuits in our benchmark suite. We acknowledge that such a previously-functioning version might not always be available, or the circuit specification may have changed. In that case, a programmer can use a partially correct or most up-to-date version of the circuit as a starting point, and manually annotate the missing or incorrect bits based on knowledge of the circuit design. This process is analogous to test suite evolution in software [180]. Ultimately, however, if manual programmer effort and previous designs are both unavailable, CirFix cannot be applied to repair defects in a circuit.

While we recognize that the process of manually annotating the correctness information may take longer than manually fixing a single defect, this information is a one-time cost as long as the high-level circuit specification (i.e., I/O wires and registers, expected behavior) does not change. Given the number of bugs that may arise during the development and maintenance of a circuit design, we believe that it would still be more cost effective to invest programmer effort in the correctness information, which can then be used by CirFix during inexpensive machine idle time (see discussion in Section 3.4.1).

3.3.1.3 Transplanting Hardware Defects

Since actual industrial defects are not made publicly available, we propose an approach based on defect *transplantation* by experts. Previous works have used either randomly-seeded or self-seeded defects for evaluation, potentially admitting bias (e.g., [120]). To combat this, we recruited three hardware experts — two of whom work in industry and one who works in academia, with 19 years of experience with hardware design collectively — to transplant (proprietary or non-public) defects from their real-world experience into otherwise-correct open source implementations of the hardware projects in our benchmark suite. We desire defects in our benchmark suite corresponding to a variety of complexities, both in terms of finding and fixing the defect. As such, we define two defect categories for this process:

- *Category 1:* A Category 1 (i.e., "easy") defect denotes mistakes pertaining to simpler, higher-level aspects of circuit design.
- *Category 2:* A Category 2 (i.e., "hard") defect denotes more intricate errors that usually require more effort to diagnose, understand, and/or fix.

To get the benefits of real-world defects in our benchmark suite, we instructed our recruited experts to transplant and categorize real defects they have previously encountered to the open-source circuits in our benchmark. We also asked our experts for "... variety in how the defects appear and would be fixed, as long as that variety aligns with how often [they] observe these bugs or mistakes in real life". We further required that any transplanted defects should compile successfully and change the externally-visible behavior of the circuit with respect to the instrumented testbench, and correspond to approximately the same level of complexity as that of real-world defects.

Table 3.3 lists the transplanted defects from our experts that met these criteria. In total, our experimental setup includes 32 different defect scenarios spanning across 11 hardware projects, with 19 Category 1 (i.e., "easy") and 13 Category 2 (i.e., "hard") defects. This benchmark suite is 1.5–10× as large as benchmark suites used in the hardware diagnosis literature [120, 117, 118, 162, 119, 137].

3.3.2 Algorithm Parameters

We refer to each execution of CirFix as a *trial*. Each trial is initialized with a distinct random seed for reproducibility of our results, and conducted on a quad-core 3.4GHz machine with hyperthreading and 16GB of memory. We ran 5 independent CirFix trials for each defect scenario, stopping when an acceptable repair was found. Each individual trial was terminated after 8 generations of evolution or 12 hours of wall-clock time (whichever came first).

For the GP parameters, we use population size $popSize = 5000$, repair template threshold $rtThreshold = 0.2$, $mutThreshold = 0.7$. In line with established practices from APR for software [149, 70, 69], we use deletion, insertion, and replacement thresholds of $0.3, 0.3$

and 0.4 respectively. For parent selection, we use a tournament size $t = 5$ to increase the selection pressure on candidate variants [181]. For elitism, we propagate the top $e = 5\%$ of each generation to the next without any modifications.

For fitness evaluations, we use $\varphi = 2$ as additional weight assigned to bits with values of x or z. This makes incorrect comparisons between ill-defined wires twice as detrimental to the fitness score of a candidate repair as binary bit mismatches. We found that a weight $\varphi = 1$ did not penalize such incorrect comparisons enough (resulting in longer times to find a repair), while $\varphi = 3$ caused too significant a drop in fitness for candidate variants (negatively impacting the exploration of the search space for a repair).

We evaluated other values suggested by literature (e.g., smaller population sizes [182, 179]), and found no significant differences in CirFix's performance.

3.3.3 Human Study Protocol

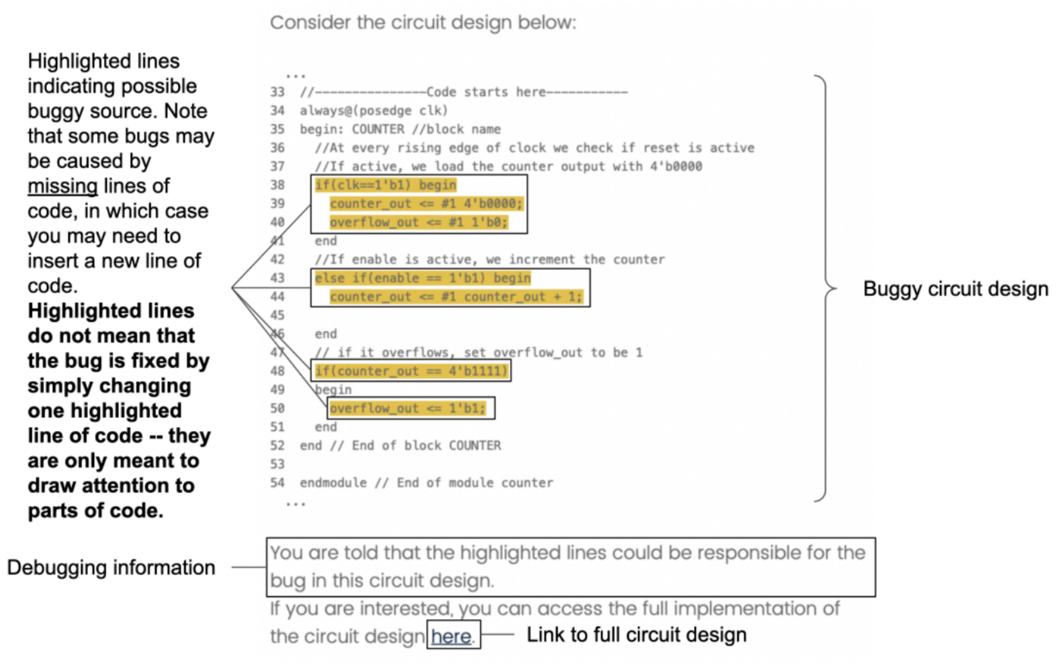
We aim to investigate our novel fault localization algorithm (see Section 3.2.1) as a debugging assistant, independent of the automated repair context. We asked programmers, rather than CirFix, to assess the quality and usefulness of the fault localization algorithm. To investigate the incremental benefit of our fault localization, we consider three scenarios: the full output of the algorithm (see Section 3.2.1), only initially implicated statements of the algorithm (without any transitive information), and no fault localization annotations.

3.3.3.1 Participant Recruitment

Under UM IRB-HUM00199335, we recruited a combination of undergraduate and graduate computer science students ($n = 41$). One student reported having less than a month of experience with HDL designs, ten students reported having 1 to 4 months experience, seven students reported having 4 months to 1 year of experience, nine reported having 1 to 2 years of experience, and the remaining six reported having 2 or more years of experience. We drew students from five undergraduate courses, a graduate course, and a computer engineering lab mailing list at the University of Michigan. At the beginning of the survey, participants' background in Verilog was collected (e.g., any courses they have taken). Participant data was anonymized, but they could optionally request a $25 USD gift card as compensation.

3.3.3.2 Debugging Scenarios

We sampled (uniformly at random) 10 defect scenarios each from student and OpenCores projects, with roughly equal numbers of Category 1 and 2 defects. To favor readability and comprehension within a time-constrained human study (e.g., [183, 184]), we filtered out


Highlighted lines indicating possible buggy source. Note that some bugs may be caused by missing lines of code, in which case you may need to insert a new line of code. **Highlighted lines do not mean that the bug is fixed by simply changing one highlighted line of code -- they are only meant to draw attention to parts of code.**
Buggy circuit design
Debugging information
Link to full circuit design

Consider the circuit design below:

```
...
//---------------Code starts here-----------
always@(posedge clk)
begin: COUNTER //block name
  //At every rising edge of clock we check if reset is active
  //If active, we load the counter output with 4'b0000
  if(clk==1'b1) begin
    counter_out <= #1 4'b0000;
    overflow_out <= #1 1'b0;
  end
  //If enable is active, we increment the counter
  else if(enable == 1'b1) begin
    counter_out <= #1 counter_out + 1;

  end
  // if it overflows, set overflow_out to be 1
  if(counter_out == 4'b1111)
  begin
    overflow_out <= 1'b1;
  end
end // End of block COUNTER

endmodule // End of module counter
...
```

You are told that the highlighted lines could be responsible for the bug in this circuit design.

If you are interested, you can access the full implementation of the circuit design here.

Figure 3.1: Example CirFix defect scenario presented to participants

defects that resulted in more than 100 lines of code implicated by fault localization. This resulted in 12 snippets from the programs in Table 3.2: eight from student projects and four from OpenCore projects. Each debugging scenario included information on the parent hardware design and documentation on the desired properties and output.

3.3.3.3 Debugging Task

Each participant was sequentially presented with 6 distinct randomly-chosen debugging scenarios. Each scenario was paired with a debugging hint: textual highlighting of implicated code, as shown in Figure 3.1.

Participants were asked to: (1) identify faulty lines (or bugs) in the snippet, (2) indicate which lines they would alter to fix the defect, (3) propose how they would alter the lines to fix the defect if they could patch it. If the snippet version presented to the participant contained fault localization hints, the participant also rated the usefulness and accuracy of those hints on a 1–5 scale.

3.4 Results

We analyzed the following research questions:

RQ 3.1 What fraction of defect scenarios can CirFix repair, and how sensitive is our fault localization approach?

RQ 3.2 How effective is the CirFix fitness function at guiding the search for a repair to a circuit description?

RQ 3.3 Does CirFix's fault localization algorithm improve programmers' objective performances?

RQ 3.4 In what contexts do programmers find CirFix's fault localization algorithm helpful?

3.4.1 RQ 3.1: Repair Rate, Quality, and Sensitivity for CirFix

Repair Rate. Table 3.3 presents the repair results for each defect scenario. CirFix produced *plausible* (i.e., testbench-adequate) repairs for 21 of the 32 (65.6%) defects. Of the 11 defects that were not repaired, 4 exhausted resource limits while 7 required edits not supported by CirFix operators and repair templates. While a direct comparison between CirFix and APR for software is not possible, we observe that the repair rate of CirFix comparable to the reported repair rates of well-known software repair approaches, e.g., GenProg (52.4%) [69], Angelix (34.1%) [74], and TBar (53.1%) [143]. When comparing CirFix to a more straightforward search algorithm applying edits at uniform to a circuit design, we found that the brute force algorithm did not scale to the complexity of defects in our benchmark suite and reported no repairs within the 12 hour resource bounds. Though not part of a comprehensive scientific evaluation, when tested on simple single-edit defects (not part of our benchmark suite) in smaller projects from undergraduate courses, the brute-force algorithm still took hours to find repairs that CirFix found in seconds to minutes, highlighting CirFix's efficient pruning of the search space. We leave a full investigation of CirFix against more straightforward search as future work. Note that we can not compare CirFix to other baselines for hardware repair, since at the time of writing, there are no baselines that operate on source code level Verilog descriptions to automatically repair defects; indeed, that is precisely the improvement CirFix brings over the state-of-the-art.

The average wall-clock time for a trial to find a repair was 2.03 hours, of which an average of over 90% was spent on fitness evaluations (i.e., design simulations). Most non-repairs timed out after 12 hours, though defects for some projects with smaller search spaces hit the 8 generation maximum first. These results are in line with previously-reported patterns of behavior for APR for software, supporting our hypothesis that the CirFix algorithm is capable of performing as well on hardware design defects as established APR approaches do on software.

Table 3.3: Repair results for CirFix. "Cat" indicates the category for the defect, "Repair Time" shows the time for repair (in seconds), and a missing time for repair indicates no repair was found in 5 independent trials. CirFix produced plausible repairs to 21 of the 32 defect scenarios in our benchmark suite, of which 16 were correct upon manual inspection (denoted with a ✓) and 14 were deemed correct along a different criteria by an independent expert team (denoted with a †).

Project	**Defect Description**	**Cat.**	**Repair Time (s)**
decoder_3_to_8	Two separate numeric errors	1	✓ 13984.3
	Incorrect assignment	2	—
counter	Incorrect sensitivity list	1	✓† 19.8
	Incorrect reset	1	✓† 32239.2
	Incorrect increment of counter	1	✓† 27781.3
flip_flop	Incorrect conditional	1	✓† 7.8
	Branches of if-statement swapped	1	✓† 923.5
fsm_full	Incorrect case statement	1	—
	Incorrectly blocking assignments	1	4282.2
	Assignment to next state and default in case statement omitted	2	1536.4
	Assignment to next state omitted, incorrect sensitivity list	2	✓† 37.0
lshift_reg	Incorrect blocking assignment	1	✓† 14.6
	Incorrect conditional	1	✓† 33.74
	Incorrect sensitivity list	1	✓† 7.8
mux_4_1	1 bit instead of 4 bit output	1	—
	Hex instead of binary constants	1	10315.4
	Three separate numeric errors	2	15387.9
i2c	Incorrect sensitivity list	2	✓† 183
	Incorrect address assignment	2	57.9
	No command acknowledgement	2	✓† 1560.5
sha3	Off-by-one error in loop	1	✓† 50.4
	Incorrect bitwise negation	1	—
	Incorrect assignment to wires	2	—
	Skipped buffer overflow check	2	✓† 50.0
tate_pairing	Incorrect logic for bitshifting	1	—
	Incorrect operator for bitshifting	1	—
	Incorrect instantiation of modules	2	—
reed_solomon_decoder	Insufficient register size for values	1	—
	Incorrect sensitivity list for reset	2	✓ 28547.8
sdram_controller	Numeric error in definitions	1	—
	Incorrect case statement	2	—
	Incorrect assignments to registers during synchronous reset	2	✓† 16607.6

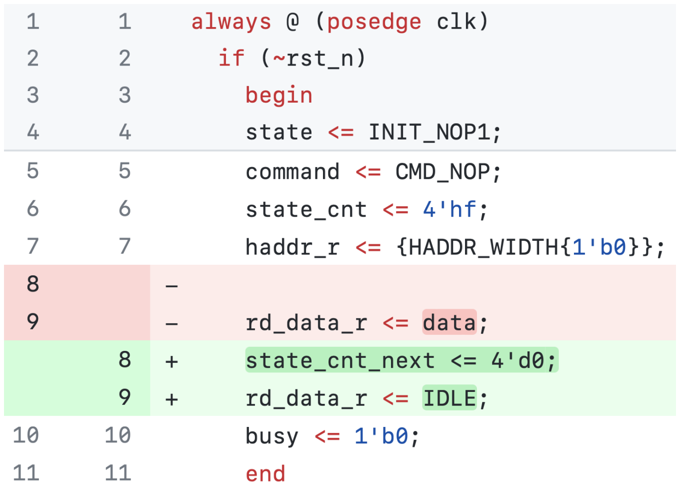
We acknowledge that wall-clock runtime for CirFix on a given defect can be longer than that of an expert human manually fixing the defect. However, CirFix was designed to favor situations in which programmer time is significantly more expensive than machine time: it is often more cost-effective to run tools like CirFix using inexpensive machine idle time and then to employ expensive programmer time to ensure the repairs are correct before being synthesized [134]. As such, we see CirFix as being cost-effective in terms of reducing the burden on programmers.

Repair Quality. We follow the approach taken by Long and Rinard [146] for patch assessment since it follows best practices in the APR literature [185, 72]. We manually analyze the 21 repairs produced by CirFix. We found 16 of the generated repairs to exhibit correct behavior, with the final 5 to be correct only with respect to the testbench (i.e., overfitting).[3] While room for improvement remains, software industrial deployments with similar rates have proved useful: for example, Bloomberg reported that a 48% correct patch rate was associated with "very positive" feedback and a general "helpful" opinion [187, p. 5].

We augment this analysis with an independent assessment from Yang *et al.*, an established expert team in APR [188, 189, 190, 191, 192], who analyzed the semantics of the produced repairs against the human-written patches and found 14 of the produced repairs to be semantically identical to the human patches (see Table 3.3). While APR expertise is not equivalent to domain expertise, APR experts tend to be more suited to assessing the patches produced by these methods due to "creative" (or adversarial or potentially-overfitting) nature of such patches [54, 193, 194], and evidence suggests that domain-experts may not be a strong gold standard [195]. We acknowledge that this assessment is not a substitute for a full human study on patch correctness; however, having two independent teams find converging results adds confidence that a majority of the plausible repairs do not overfit to the testbench (a common problem in APR for software [196, 197, 146]), since we inspect intermediate wire values when assigning fitness scores. We do note that correctness is critical in hardware designs (e.g., since manufactured chips cannot be easily updated once deployed), and our use case does not involve deploying patches directly but instead showing plausible patches to programmers to reduce maintenance costs [134, 135].

We observed that 7 out of the 21 minimized repairs were multi-edit repairs, highlighting CirFix's ability to produce repairs to defects that require more than one change to the circuit design. By comparison, common APR approaches for software usually only produce single-edit repairs [65], and only recently have there been works investigating multi-edit

[3]We focus on correctness of a patch against the specification of the circuit (e.g., ensuring the absence of clock- or reset-domain issues) during our manual inspections. The synthesizability of the design is left to be evaluated by the programmer during the validation phase of the hardware design process [186].



```
 1    1      always @ (posedge clk)
 2    2        if (~rst_n)
 3    3          begin
 4    4          state <= INIT_NOP1;
 5    5          command <= CMD_NOP;
 6    6          state_cnt <= 4'hf;
 7    7          haddr_r <= {HADDR_WIDTH{1'b0}};
 8        -
 9        -      rd_data_r <= data;
      8   +      state_cnt_next <= 4'd0;
      9   +      rd_data_r <= IDLE;
10   10          busy <= 1'b0;
11   11          end
```

Figure 3.2: A representative multi-edit repair by CirFix for a defect in the `sdram_controller` benchmark. The original defect, with a missing and an incorrect assignment, is shown in red; the repaired code is shown in green. Edits on lines 8 and 9 correspond to insert and replace operations respectively.

repairs [74, 198].

For instance, in a faulty version of the `sdram_controller` benchmark, one of our experts changed assignments to two wires to transplant a Category 2 defect, causing incorrect functionality in the host interface. CirFix assigned this faulty design code a fitness value of 0.818 based on output mismatch. CirFix repaired this defect scenario in 4.6 hours by inserting a new assignment and modifying an existing assignment. The original defect and the repaired code are shown in Figure 3.2. This is an indicative instance of CirFix repairing Category 2 (i.e., “hard”) defects with multiple edits to the faulty circuit design. We return to multi-edit repairs in the human study results (Section 3.4.4).

Fault Localization Sensitivity. To assess repair performance as fault localization quality decreases, we conducted a targeted experiment reducing the quality of the initial fault location available to CirFix in a controlled manner. This sort of investigation, in which the sensitivity of the algorithm with respect to fault localization is assessed, is important in software APR [199, 200, 201, 202].

When simulation outputs are compared against expected behavior to produce the initial set of wires and registers with mismatched values (see Section 3.2.1), we also randomly include some correct wires and registers (with probability 25%, 50%, or 75%) as “noise”. Because our fault localization is a transitive fixed point calculation, additional initial elements may result in larger fault localization sets (e.g., informally, the traditional scalability problem with fault localization is that almost everything may end up implicated).

We focus on defect scenarios CirFix successfully repaired. Table 3.4 presents normalized

Table 3.4: Repair results for CirFix with added noise to initial mismatch set for our fault localization algorithm. "Defect Cat." indicates the category for the defect, "Normalized Repair Time" shows the normalized time for repair (in seconds) when compared to the original repair, and a '—' indicates no repair was found in 5 independent trials. "Noise" indicates the percent of disturbance placed on the fault localization. The ordering of the benchmarks follows Table 3.3.

Project	**Defect**	**Normalized Repair Time**		
	Cat.	**25% Noise**	**50% Noise**	**75% Noise**
decoder_3_to_8	1	1.11×	—	—
counter	1	0.49×	0.45×	0.05×
	1	0.48×	0.58×	0.86×
	1	0.06×	0.98×	0.98×
flip_flop	1	0.99×	0.38×	1.86×
	1	0.87×	1.18×	0.35×
fsm_full	1	0.77×	0.08×	0.58×
	2	0.58×	0.57×	0.81×
	2	3.21×	3.24×	1.76×
lshift_reg	1	1.07×	0.11×	0.11×
	1	0.18×	0.49×	0.21×
	1	1.01×	0.32×	0.60×
mux_4_1	1	0.27×	0.35×	0.61×
	2	1.19×	1.27×	1.24×
i2c	2	0.93×	0.39×	0.34×
	2	0.04×	0.13×	0.13×
	2	18.57×	—	15.88×
sha3	1	1.44×	2.80×	3.60×
	2	0.67×	0.33×	0.73×
reed_solomon_decoder	2	1.39×	0.52×	1.29×
sdram_controller	2	0.11×	1.22×	0.55×

results of five trials at each noise level. Of the 21 defect scenarios CirFix originally plausibly repaired, CirFix also found plausible repairs for all 21 when subjected to 25% noise, 19 at 50% noise, and 20 at 75% noise. Execution times with lower-quality fault localization are not statistically different to those found without fault localization noise ($p = 0.7$, $p = 0.6$, $p = 0.9$, unpaired Student t-test), suggesting that CirFix performs similarly even if the design or testbench does not admit precise fault localization. Any difference in execution times can be attributed to the randomness of the search for repairs (a larger fault localization set may result in new candidate repairs or repairs being considered in a different order). An investigation of this outcome reveals that many of the same registers and wires were transitively implicated in both cases (i.e., with and without noise). For example, in the largest benchmark (reed_solomon_decoder), there are 10 (out of 11 maximum) elements in the initial mismatch set and 114 in the final fault localization set. With 75% noise, there are 11 elements in the initial set but 124 in the final fault localization set. This small increase suggests that many of the potential wires and registers were already transitively implicated without the added noise. Our targeted experiment furthers confidence that CirFix's novel fault localization approach scales to larger designs or those with more complicated or less precise testbenches that do not admit accurate initial fault localization.

CirFix produced plausible repairs to 21 out of 32 (65.6%) defect scenarios in our benchmark suite, of which 16 repairs were fully correct and 5 were correct only with respect to the testbench. The CirFix repair rate is comparable to strong results from APR for software, suggesting that our approach brings the benefits of APR to hardware designs. Lastly, our sensitivity investigation gives confidence that CirFix's fault localization approach scales to larger designs.

3.4.2 RQ 3.2: Quality of Fitness Function

CirFix's high repair rate suggests that our fitness function, coupled with our testbench instrumentation approach, is highly effective at guiding the search for repairs to faulty circuit designs. We observe that for each change to design code that brings a candidate repair closer to a correct repair, our fitness function shows a corresponding increase in the candidate repair's fitness (i.e., our fitness function has a strong *fitness distance correlation*, a trait that makes genetic algorithms thrive [141]). This is best observed in transplanted defects that require multiple edits to the design code to be corrected. For instance, one of our experts transplanted a defect in the `counter` project that required three edits to the design be repaired. The triple-edit repair produced by CirFix for this defect scenario incrementally raised the fitness of the best candidate patch first from 0 to 0.58, then to 0.77, and finally to 1.0 to produce a correct repair. Similar behavior is seen for every other multi-edit repair produced by CirFix, indicating that our fitness function is effective at capturing incremental changes to a circuit design during the search for a repair.

We also observe instances where CirFix produces a repair deemed unfit by our fitness function and instrumented testbench but considered correct by the original, unannotated testbench. We examine one such case in detail, related to the `out_stage` module in the error correction core `reed_solomon_decoder`. This module is responsible for generating output bytes from pipelining input memories. A faulty version of this circuit obtained from one of our experts removed the `reset` wire from the sensitivity list of an `always` block. This caused incorrect resetting of output wires by the circuit. Our fitness function assigns the incorrect design code a non-perfect fitness value of 0.999. The original testbench, however, reports no errors in the incorrect code. The final repair produced by CirFix fixes this defect and passes all checks by the original testbench and our instrumented testbench. This suggests that our fitness function and testbench instrumentation can catch errors beyond the capabilities of the original testbench without adding any additional testing logic.

The CirFix fitness function is highly effective at capturing incremental changes to a circuit's design code to guide the search for a repair, and has the potential to increase testing prowess without any added testing logic to a bench.

3.4.3 RQ 3.3: Fault Localization and Human Performance

We assessed programmer performance by evaluating (1) F-scores (F_1) of correctly-identified faults for each debugging task by each participant and (2) total time taken to complete a debugging task within no specific time limit (see Section 3.3.3.3). A participant is said to correctly identify faults for a given defect scenario if they identified program line(s) that contain a bug or missing line(s). F-scores were evaluated by calculating the harmonic mean of recall and precision.

To evaluate the statistical significance of participants utilizing the fault localization as a debugging aid as opposed to none, we used the unpaired Student t-test. We did not observe a statistically-significant difference in time taken to localize faults with full or no annotations from our fault localization ($p = 0.41$). On average, participants spent 299.6 seconds with full annotations as opposed to 239.0 seconds with no annotations. A participant with an F-score of 1 correctly identified faulty program line(s) or missing line(s) in the defect scenario, while a F-score of 0 meant no faulty program line(s) or missing line(s) were correctly identified. We did find that the objective F-score for participants given full localization was higher ($F_1 = 0.67$) than the objective F-score for participants who had half fault localization

($F_1 = 0.33$), which in turn was higher than those without fault localization ($F_1 = 0.29$). However, while this trend did not fully rise to the level of statistical significance ($p = 0.12$), our results indicate CirFix data-flow based notion of fault localization could be a useful tool for manual debugging, warranting further exploration.

In addition, we found statistically-significant differences in the F-scores between experts ($F_1 = 0.37$) and novices ($F_1 = 0.17$) when they had CirFix's fault localization with a large effect size ($p = 0.04$, $d = 0.54$). This statistic did not survive correcting for multiple comparisons. However, all other significant values reported survive correcting for multiple comparisons ($q = 0.05$) to avoid false discovery. We used Cohen's d due to similarities in standard deviations in the groups.

We observe a trend suggesting that CirFix fault localization may improve programmers' objective performances, but this trend does not fully rise to the level of statistical significance ($p = 0.12$).

3.4.4 RQ 3.4: Subjective Judgment of Fault Localization

We assessed participant subjective judgements of CirFix's fault localization support in various contexts, including debugging multi-line defects and different circuit designs (see Section 3.3.1).

For each presented stimulus with a debugging aid, participants were asked to rate, on a Likert scale, the usefulness and accuracy of the algorithm in helping them localize the circuit defects as seen on Figure 3.3. Differences in the number of responses per rating arise because not all participants answered all questions.

Participants rated full fault localization support on student-developed designs to be significantly more useful and accurate than full support for open source projects ($p = 0.01$, $d = 0.7$; $p = 0.002$, $d = 1.05$, a large effect size). These results suggest our algorithm would be more beneficial for debugging in pedagogical environments.

Most interestingly, we find that participants rated CirFix's fault localization support to be significantly more useful and accurate for debugging multi-line defects than single-line defects with a large effect size ($p = 0.002$, $d = 1.04$; $p = 0.003$, $d = 0.86$). Given that support for multi-line software repairs is limited [203, 204], with most algorithms only supporting single-line repairs, our results, by contrast, are promising for reducing maintenance costs associated with more complex defects in the hardware domain.

The statistically significant results on the subjective judgment of CirFix's fault localization may prove to be more beneficial for pedagogy. In our qualitative analysis of optional questions given to participants at the end of the study, we found that participants, particularly novices, who self-reported to be less effective at tasks related to debugging hardware

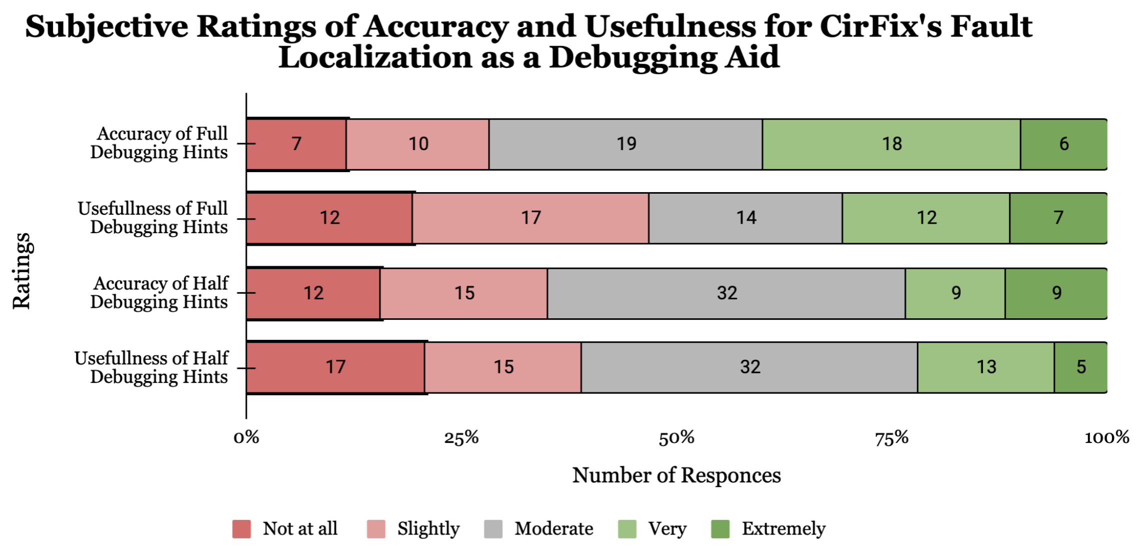


Figure 3.3: Subjective ratings of CirFix's fault localization when used as a debugging aid. Subjects rated the algorithm as a debugging aid based on accuracy and usefulness on a scale of 1–5, where 1 represents not at all accurate or useful and 5 represents extremely accurate or useful.

designs, rated the debugging features (e.g., highlighting of implicated statements and naming of implicated wires or registers) to be significantly useful ($p = 0.02$, $d = 0.97$; $p = 0.001$, $d = 1.35$). This suggests that debugging aids with supplemental supportive features, such as our fault localization algorithm, could help novices navigate these tasks. Despite advances in hardware development platforms, novices still report intimidation by circuitry [205]. The self-efficacy of students can be improved by providing them with support they find useful, such as our fault localization algorithm.

CirFix fault localization may be helpful for multi-line defects ($p \leq 0.003$) in classroom contexts ($p \leq 0.02$).

3.5 Threats to Validity

In this section, we summarize threats to validity of our experimental results.

Choice of algorithm parameters. The parameters for the prototype implementation of CirFix are chosen based on empirical performance and may not be optimal. We do note, however, that the repair operators, fault and fix localization approaches, and representation choice for repairs matter more than the actual values of the GP parameters for APR [206].

Benchmark suite construction. Our benchmark defects may not be indicative of defects in real-world hardware projects, posing a potential threat to external validity. To mitigate this threat, we evaluated CirFix on a variety of hardware projects taken from different sources, and had expert hardware programmers transplant defects from their real-life experience with HDL designs covering a variety of defect types (see Section 3.3.1.3).

Performance scalability. While our results on the scalability of CirFix's repairs gives us confidence that our implementation scales to larger benchmarks than those we tested, additional programmer effort may be needed to apply CirFix to very large designs, such as modularizing the design and testbench (cf. functions and unit tests in software). We leave further optimizations to the CirFix fault localization (e.g., more efficient pruning of the search space) as future work.

Participant recruitment. Finally, our human study participants are students. While they may represent new hires joining the workforce, they are not indicative of experienced hardware programmers.

3.6 Related Work

In this section, we discuss related work on automatic error correction on hardware designs and automated program repair for software.

3.6.1 Automatic Error Diagnosis and Correction in Hardware Designs

While a significant amount of work has been done in automatic error diagnosis of hardware designs, the correction of such errors automatically has not been well-explored to the best of our knowledge. Techniques in the works of Jiang *et al.* [117] and Ran *et al.* [118] employ software analysis approaches to identify statements in design code responsible for defects, but suffer from high false positive rates.

Bloem and Wotawa [119] use formal analysis of circuit descriptions to identify defects, and Peischl and Wotawa [122] use a model-based diagnosis paradigm that supports source-level debugging of large VHDL designs at the statement and expression level. This use of formal methods for error diagnoses is orthogonal to our work, but could be applied to reduce the search space for approaches like CirFix.

Staber *et al.* [162] use state-transition analysis to diagnose and correct hardware designs automatically, but their techniques similarly do not scale to real-world circuits with large state spaces. Our approach, by contrast, is more scalable to larger, real-world hardware descriptions. Chang *et al.* [120] explicitly insert multiplexers to automatically diagnose faults in hardware designs and suggest repairs; Madre *et al.* [121] use Boolean equation solving to diagnose and rectify gate-level design errors. By contrast, our technique also applies to behavioral (higher-level) aspects of a circuit design.

3.6.2 Automated Program Repair for Software

In the realm of software, significant research effort has been devoted to repairing bugs automatically over the last 15 years [65, 66, 67]. Automated program repair usually takes as input source code with a deterministic bug and a test suite with at least one failing test that reveals the bug, and aims to automatically generate fixes to the buggy code. Test suite based repair, where test cases are used to guide the search for a patch, can be further divided into generate-and-validate and semantics-driven approaches. Generate-and-validate techniques produce candidate patches for the buggy code and evaluate them against the test suite to check if all tests pass [69, 70, 71, 72]. Semantics-driven approaches first extract constraints on a program based on test suite execution and then use these constraints to synthesize a patch [73, 74, 75, 76]. While software approaches to APR make use of test suites to evaluate candidate repairs, CirFix uses instrumented hardware testbenches to make visible the internal and external behavior of a simulated circuit for fitness evaluation. Additionally, APR for software usually uses spectrum-based fault localization to implicate faulty code, whereas CirFix uses our novel fault localization approach supporting parallel hardware descriptions.

3.7 Chapter Summary

This chapter presents CirFix, a framework for automatically repairing defects in hardware designs (i.e., digital logic) implemented in languages like Verilog. CirFix makes use of readily-available artifacts included in the hardware design process (e.g., testbenches) to diagnose and repair defects in the circuit description. These repairs can then be shown to programmers for validation before the synthesis phase, reducing maintenance costs. The testbench-based evaluation and the parallel structure of hardware designs pose challenges that render traditional APR approaches from software inapplicable to the hardware domain.

We present two key insights to bridge this gap. First, we propose a method to instrument hardware testbenches to make a circuit's behavior externally available to guide the search for repairs. We present a novel fitness function that performs a bit-level comparison of the made-visible output wire values against expected behavior to assess functional correctness of candidate repairs. Second, we present a novel fault localization approach based on a fixed point analysis of assignments made to registers and output wires to implicate statements for defects, since spectrum-based approaches commonly used in APR do not apply to hardware designs.

Our systematic evaluation of CirFix presents a new benchmark suite of 32 defect scenarios transplanted by three hardware experts across 11 different Verilog projects. CirFix produces testbench-adequate repairs for 21 out of 32 and fully correct repairs for 16 out of 32 of the Verilog defects within reasonable resource bounds. Lastly, we evaluated the relative utility of our novel fault localization algorithm independent of our automated repair context via a human study involving 41 participants. We observed a trend suggesting that CirFix fault localization may improve programmers' objective performance, and found a statistically significant preference ($p \leq 0.003$) for CirFix fault localization as a debugging aid in fixing multi-line defects, primarily in classroom contexts ($p \leq 0.01$). While outside the immediate scope of this thesis, the investigation of a hardware debugging tool in a classroom setting is merited in light of our results, and we see this investigation as promising future work (see Section 6.1.1).

In the next chapter, we explore the use of eye-tracking to investigate cognition for computer science formalisms (i.e., mathematical logic).

CHAPTER 4

Cognition for Formalisms

Formal methods (i.e., mathematical logic) have long been used to provide rigorous guarantees for software engineering [207] (e.g., ensuring that executions of a program never reach an invalid state), and have been incorporated into various core stages of the software process. Successful applications of formal methods to software include requirements elicitation [208], software specification (e.g., the *abstract state machine* method [209], *design by contract* [210]), software design (e.g., *Vienna Development Method* [211]), software verification [212, 213], testing [214], and maintenance (e.g., legacy code at Microsoft makes use of assertions [215]). Unfortunately, many formal methods require advanced mathematical training and theorem proving skills that practitioners typically lack [216].

Given the extensive use of, and increased opportunities for, formal methods in software engineering [217], educators are placing an increased focus on formalisms (e.g., proofs of algorithmic properties, runtime complexity analyses, etc.) in undergraduate computer science curricula to prepare future programmers for logical algorithmic reasoning [218]. Unfortunately, despite the emphasis placed on formalisms in undergraduate computer science theory courses, students have historically struggled with course outcomes (e.g., in terms of final grades, mastery and retention of material, etc.). For instance, survey data from a large public university in the US highlight a trend of dissatisfaction and low outcomes from core theory courses focusing on formalisms [219, 220]. Given the difficulty associated with having engineers integrate formal methods into the software process [221], it remains important for educators to ensure that future practitioners are trained in logical reasoning skills, especially as they relate to code.

Previous work has used methodologies like eye tracking, occasionally coupled with fMRI and fNIRS [44, 222], to investigate student cognition for computer science tasks relevant to software engineering, including reading [183] and writing code [46], manipulating data structures [45], and reviewing code [54]. Researchers have also examined the cognitive models associated with higher-level math tasks, including number processing and arithmetic [223].

However, since formal methods fundamentally differ from other software development processes in their focus on mathematical reasoning instead of coding, lessons learned from cognition for coding tasks may not clarify how students comprehend formalisms, what distinguishes an expert in formal reasoning from a novice, and how educators can better prepare students for formal reasoning for computing.

In this chapter, we propose to use eye-tracking to gather insights into the problem-solving strategies for formalism comprehension (i.e., mathematical logic) tasks employed by students with different levels of familiarity with, or incoming preparation for, formal methods. We also investigate reasoning strategies that are correlated with student success, and shed light on interventions that could help educators better prepare struggling students for the rigorous logical reasoning required by high-assurance software engineering.

4.1 Overview of Experimental Design, Results, and Contributions

We argue that understanding how people less familiar with formalisms think about formal methods and proofs of algorithmic properties could be critical to how educators should teach formalisms. For instance, since the most vulnerable population groups with non-traditional backgrounds are also the ones most likely to drop the computer science major [219], educators need to make sure the needs of such groups are not overlooked. This understanding can also indirectly impact how high-assurance software engineering firms might train new workers. We focus on acquiring this understanding through the investigation of the cognition (e.g., problem-solving strategies, difficulty completing a task, visual attention, etc.) of computer science students while performing formalism comprehension tasks (see Section 2.3).

We present a controlled experiment investigating how students read and assess formal proofs about algorithms for correctness. We recruited 34 participants with varying levels of preparation for formalisms to perform these comprehension tasks. Participants were shown pseudocode algorithms from a widely used undergraduate textbook [224], a theorem about the algorithm with an accompanying formal proof, and a graphical illustration of the algorithm or proof. Participants were asked to evaluate the presented proofs for correctness.

We consider both prior coursework and current performance in our definition of incoming preparation for our participants. We first asked participants to outline the number of computer science theory courses covering formalisms (e.g., derivations and proofs) they had either completed with passing grades or were currently taking. We then tested participants to identify the mistake in a proof taken from an undergraduate textbook. We partitioned our sample based on whether a participant had taken more than the median number of courses in our sample *and* passed a screening test (see Section 4.4.1 for a more in-depth discussion on incoming preparation).

Contrary to conventional wisdom suggesting that students with greater incoming preparation achieve better outcomes for STEM courses [26, 225, 226, 227], we found no evidence that students with higher incoming preparation perform better at these formalism tasks ($p = 0.96$, Cohen's $d = 0.007$). We further find no evidence that student experience reports are accurate predictors of outcomes for formalism comprehension tasks ($\tau = 0.21, p = 0.15$), or that students are able to correctly identify the parts of a formalism presentation most pivotal to understanding a proof. Our results also indicate more-prepared students employ different problem solving strategies, with an increased visual attention on proof prose text ($p = 0.005$) and correct ($p = 0.03$) and distractor ($p = 0.038$) answer choices, but this ultimately is not correlated with task outcomes.

We do find, however, that higher-outcome students demonstrate significantly more attention switching behaviors (i.e., frequently going back and forth between presented materials) ($p = 0.002$), and are more likely to perform better at proofs by induction ($p = 0.01$) and recursive algorithms ($p = 0.006$) compared to lower outcome students. Our results argue for the need for pedagogical intervention in theory courses to ensure student outcomes are better aligned with the objectives of preparing future software engineers for formal methods.

The main contributions of this chapter — separately published in the 45th ACM/IEEE International Conference on Software Engineering [228] — are:

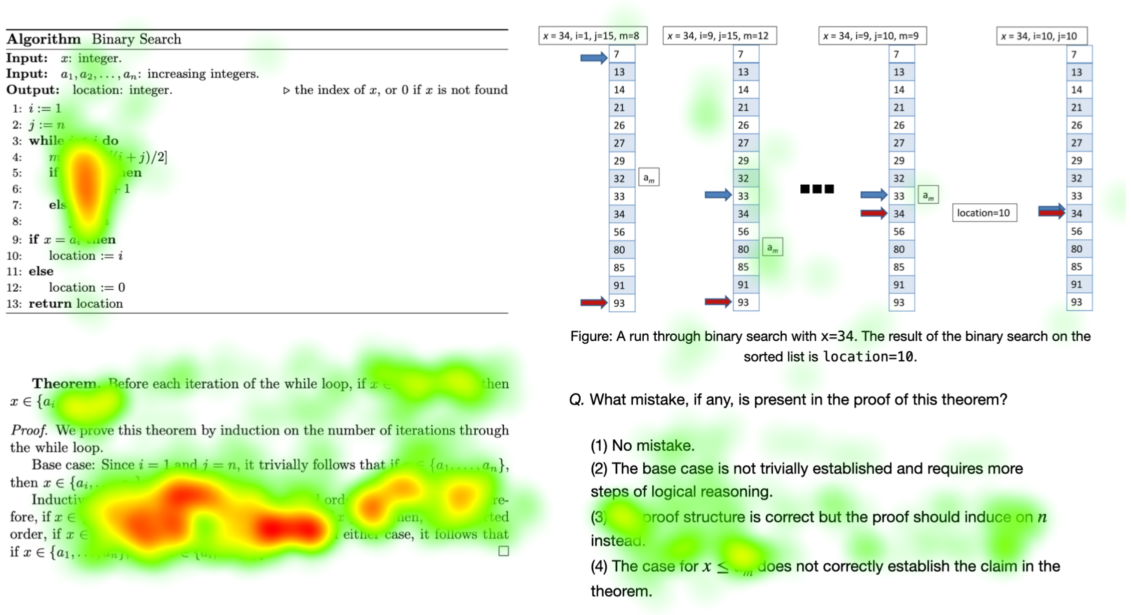
- A controlled experimental study ($n = 34$) investigating student cognition for computer science formalisms.
- Experimental evidence that suggests incoming preparation does not predict outcomes for formalism comprehension tasks ($p = 0.96$), and that students with higher outcomes employ different problem-solving strategies ($p = 0.002$) and exhibit better performance for certain types of proofs and algorithms ($p \leq 0.01$).
- Recommendations for educators to further investigate pedagogy for mathematical logic reasoning, including designing teaching materials to facilitate going back and forth between the presented content with ease, and emphasizing inductive and recursive problem-solving.
- A publicly available dataset, along with relevant analysis scripts, for future studies investigating cognition for computer science formalisms.

4.1.1 Motivating Example

We desire a deeper understanding of the problem-solving strategies for formalism comprehension tasks employed by students with different levels of incoming preparation. Traditional metrics, such as evaluating student transcripts or self-reports, are not as effective at teasing apart the problem-solving strategies employed by higher performing students (e.g., [229]). We hypothesize that eye-tracking can serve as a cost-effective, insightful methodology to investigate the factors correlated with better outcomes for formalism comprehension tasks. For instance, struggling students may demonstrate *higher regression rate* for (i.e., re-read text and figures more frequently), or *increased visual attention* to, certain aspects of a formalism presentation, suggesting greater difficulty completing the task.

As an indicative example, we present a snapshot of the strategies (in terms of visual attention) employed by two students with different incoming preparation for undergraduate theory courses (Figure 4.1). Figure 4.1a shows the *visual heatmap*, constructed from gaze data collected by an eye-tracker, for a participant with higher incoming preparation for theory courses. The heatmap indicates a significant proportion of visual attention to the proof text and answer choices (lower left and right quadrants respectively). By contrast, the visual heatmap for a participant with lower incoming preparation (Figure 4.1b) suggests a comparatively increased emphasis on the algorithmic pseudocode and figures (upper left and right quadrants respectively). While the increased focus by a less-prepared participant on the algorithm and figures aligns with educator expectations, one would also expect a more-prepared participant focusing attention on the proof text to achieve better response accuracy. Quite surprisingly, we find that both participants fail to correctly identify the presence of mistakes in the proof, and that this trend of no correlation between traditional measures of incoming preparation and task outcomes extends to the entirety of our participants (see Section 4.4.1 for a discussion on preparation and outcomes). Note that our use of heatmaps is intended to present a snapshot of what participants focus on, and is not the sole point of comparison between participants with different outcomes. Our results in Section 4.4 incorporate various eye-tracking metrics better suited to understanding the complete picture.

Given that neither incoming preparation nor the set strategies employed by more-prepared students is sufficient at teasing apart factors that result in better outcomes for such formalism comprehension tasks, we turn to eye-tracking to clarify factors correlated with student success. Our results help us better understand what strategies are correlated with student success, and in turn, how educators can better prepare future engineers to reason about formal methods.



```
Algorithm Binary Search
Input:  x: integer.
Input:  a_1, a_2, ..., a_n: increasing integers.
Output:  location: integer.          ▷ the index of x, or 0 if x is not found
 1: i := 1
 2: j := n
 3: while [illegible] do
 4:     m [illegible](i + j)/2]
 5:     if [illegible] then
 6:         [illegible] + 1
 7:     els[illegible]
 8:         [illegible]
 9: if x = a[illegible] then
10:     location := i
11: else
12:     location := 0
13: return location
```

Theorem. Before each iteration of the while loop, if $x \in$ [illegible] then $x \in \{a$[illegible]

Proof. We prove this theorem by induction on the number of iterations through the while loop.

Base case: Since $i = 1$ and $j = n$, it trivially follows that if [illegible] $\{a_1, \ldots, a_n\}$, then $x \in \{a_i,$ [illegible]

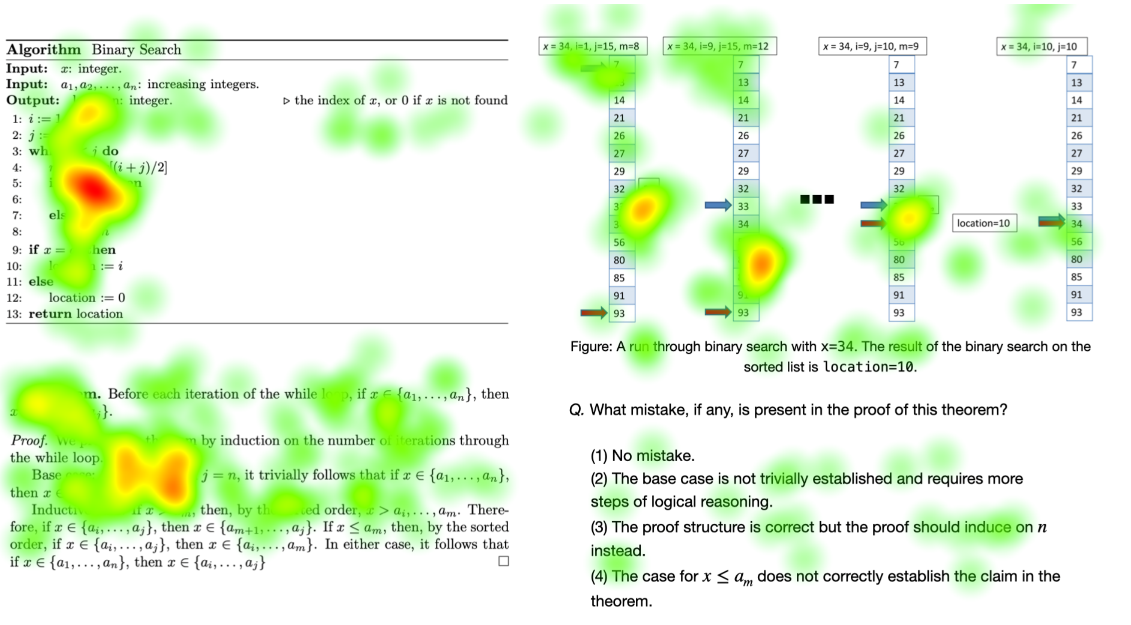
Inducti[illegible] ord[illegible] re-fore, if $x \in$ [illegible] then, [illegible] ted order, if $x \in$ [illegible] either case, it follows that if $x \in \{a_1, \ldots$ [illegible] □

Figure: A run through binary search with x=34. The result of the binary search on the sorted list is `location=10`.

Q. What mistake, if any, is present in the proof of this theorem?

(1) No mistake.
(2) The base case is not trivially established and requires more steps of logical reasoning.
(3) [illegible] proof structure is correct but the proof should induce on n instead.
(4) The case for $x \leq$ [illegible] does not correctly establish the claim in the theorem.

(a) Higher incoming preparation participant heatmap



```
Algorithm Binary Search
Input:  x: integer.
Input:  a_1, a_2, ..., a_n: increasing integers.
Output:  [illegible]: integer.          ▷ the index of x, or 0 if x is not found
 1: i := 1
 2: j :=
 3: wh[illegible] do
 4:     [illegible](i + j)/2]
 5:     i[illegible]
 6:
 7:     els
 8:
 9: if x = [illegible] then
10:     [illegible] := i
11: else
12:     location := 0
13: return location
```

[illegible]**m.** Before each iteration of the while [illegible], if $x \in \{a_1, \ldots, a_n\}$, then x [illegible]$\}$.

Proof. We [illegible] by induction on the number of iterations through the while loop.

Base [illegible] $j = n$, it trivially follows that if $x \in \{a_1, \ldots, a_n\}$, then x [illegible]

Inducti[illegible] if $x >$ [illegible], then, by th[illegible]ed order, $x > a_i, \ldots, a_m$. Therefore, if $x \in \{a_i, \ldots, a_j\}$, then $x \in \{a_{m+1}, \ldots, a_j\}$. If $x \leq a_m$, then, by the sorted order, if $x \in \{a_i, \ldots, a_j\}$, then $x \in \{a_i, \ldots, a_m\}$. In either case, it follows that if $x \in \{a_1, \ldots, a_n\}$, then $x \in \{a_i, \ldots, a_j\}$ □

Figure: A run through binary search with x=34. The result of the binary search on the sorted list is `location=10`.

Q. What mistake, if any, is present in the proof of this theorem?

(1) No mistake.
(2) The base case is not trivially established and requires more steps of logical reasoning.
(3) The proof structure is correct but the proof should induce on n instead.
(4) The case for $x \leq a_m$ does not correctly establish the claim in the theorem.

(b) Lower incoming preparation participant heatmap

Figure 4.1: Visual eye-gaze heatmaps for a stimulus shown to two participants with different incoming preparation. The more-prepared participant (top) focuses visual attention primarily on the proof and answer choices, while the less-prepared participant focuses on the algorithm and figure. Both participants choose the wrong answer.

4.2 Experimental Setup and Methods

Our experiment centers on a human study in which participants answer questions about computing formalisms (algorithms, theorems, proofs, and figures) while subjected to eye-tracking. We make the replication materials for our study (including the pre- and post-questionnaires, stimuli, and de-identified raw data) publicly available at https://doi.org/10.5281/zenodo.7626901.

4.2.1 Participant Demographics and Recruitment

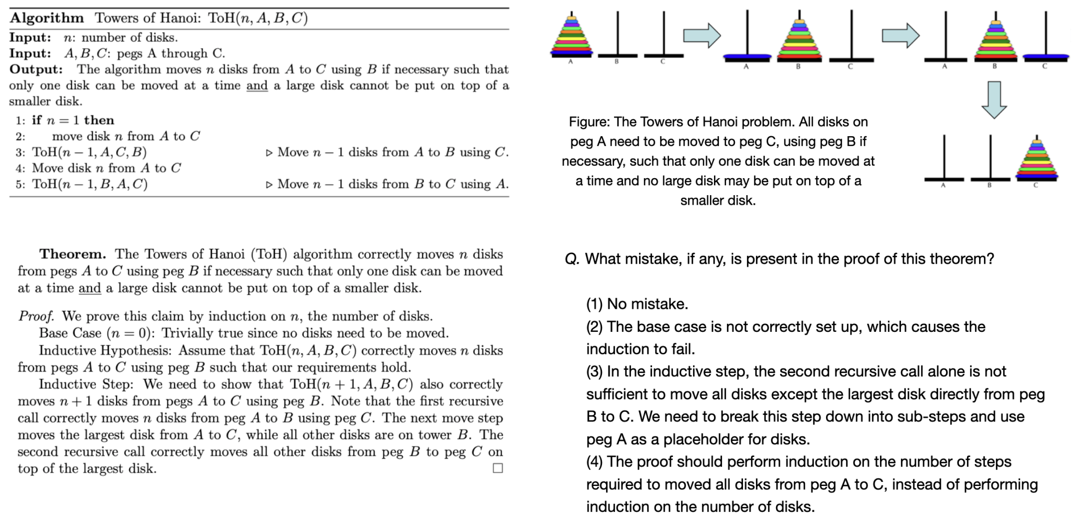
We recruited 34 undergraduate and graduate computer science students at the University of Michigan in an IRB-exempt study (HUM00204278).[1] Of our 34 participants, 23 identified as men, while 11 identified as women. Breaking down our participants by class standing, we recruited 5 first-year students, 10 sophomores, 12 juniors, 6 seniors, and one graduate student. We required participants be over 18, have completed an undergraduate discrete mathematics course, and be either enrolled in or have completed an undergraduate data structures and algorithms course. In addition, to reduce noise in the recorded gaze data, we encouraged (but did not require) our participants to wear contact lenses in lieu of glasses to the experiment session where possible. Participants were compensated $25 for an hour of study time.

4.2.2 Materials and Design

Participants were asked to complete a sequence of formalism comprehension tasks. Each individual task stimulus consisted of an algorithmic solution to a problem commonly taught in core computer science undergraduate courses, a theorem for that algorithm, an accompanying proof of that theorem, and a relevant graphical illustration (or figure). Each formalism comprehension task presented four multiple-choice answers for the presence of mistakes in the proof, of which only one was correct and three were distractors.

We seeded each proof with mistakes commonly made by undergraduate students in discrete mathematics courses (e.g., incorrect base case for proof by induction, logical contradictions in deductive reasoning, arithmetic errors leading to incorrect conclusions, etc.), and asked participants to evaluate each proof for correctness. For each algorithm, participants

[1]A pre-study power analysis, assuming a mean outcome score of 60 ± 10 for less-prepared participants, a performance increase of 20% for the more-prepared participants, an enrollment ratio of 1:1 between the two population groups, and α and power $(1 - \beta)$ values of 0.05 and 0.8, suggests a sample size of 22 participants (11 in each group). Our estimated scores and performance deltas were based on observed student outcomes an instructor expectations associated undergraduate CS theory courses at the University of Michigan.



Algorithm Towers of Hanoi: ToH(n, A, B, C)

Input: n: number of disks.
Input: A, B, C: pegs A through C.
Output: The algorithm moves n disks from A to C using B if necessary such that only one disk can be moved at a time <u>and</u> a large disk cannot be put on top of a smaller disk.

```
1: if $n = 1$ then
2:     move disk $n$ from $A$ to $C$
3: ToH($n-1, A, C, B$)            ▷ Move $n-1$ disks from $A$ to $B$ using $C$.
4: Move disk $n$ from $A$ to $C$
5: ToH($n-1, B, A, C$)            ▷ Move $n-1$ disks from $B$ to $C$ using $A$.
```

Figure: The Towers of Hanoi problem. All disks on peg A need to be moved to peg C, using peg B if necessary, such that only one disk can be moved at a time and no large disk may be put on top of a smaller disk.

Theorem. The Towers of Hanoi (ToH) algorithm correctly moves n disks from pegs A to C using peg B if necessary such that only one disk can be moved at a time <u>and</u> a large disk cannot be put on top of a smaller disk.

Proof. We prove this claim by induction on n, the number of disks.
Base Case ($n = 0$): Trivially true since no disks need to be moved.
Inductive Hypothesis: Assume that ToH(n, A, B, C) correctly moves n disks from pegs A to C using peg B such that our requirements hold.
Inductive Step: We need to show that ToH($n+1, A, B, C$) also correctly moves $n+1$ disks from pegs A to C using peg B. Note that the first recursive call correctly moves n disks from peg A to B using peg C. The next move step moves the largest disk from A to C, while all other disks are on tower B. The second recursive call correctly moves all other disks from peg B to peg C on top of the largest disk. □

Q. What mistake, if any, is present in the proof of this theorem?

(1) No mistake.
(2) The base case is not correctly set up, which causes the induction to fail.
(3) In the inductive step, the second recursive call alone is not sufficient to move all disks except the largest disk directly from peg B to C. We need to break this step down into sub-steps and use peg A as a placeholder for disks.
(4) The proof should perform induction on the number of steps required to moved all disks from peg A to C, instead of performing induction on the number of disks.

Figure 4.2: A sample formalism comprehension stimulus for the Towers of Hanoi problem. The algorithm (upper-left), figure (upper-right), theorem and proof (lower-left) and correct and distractor answer choices (lower-right) represent the six areas of interest for the stimulus. The correct answer is (2): the base case should apply when $n = 1$, and more reasoning is required to establish the claim for the base case.

were always given the option to indicate that a presented proof contains no mistakes.

For our study, we presented 7 algorithms taken from a commonly-used undergraduate discrete mathematics textbook [224]: binary search, greedy change-making, merge-sort, Towers of Hanoi, greedy job scheduling, in-order tree walk, and the Halting Problem. Each algorithm was accompanied by a theorem and a proof copied verbatim from the textbook prior to mistake-seeding. Since the textbook is widely used by educators for introductory computer science theory courses, we are interested in evaluating the efficacy of the presented material for student outcomes, and as such, do not alter the proof to make the prose or logic more or less comprehensible. Since figures are frequently used as an educational instrument (e.g., [230]), we included, with each stimulus, a figure related to that formalism comprehension task taken from the textbook or instructor slides for the theory courses. Figure 4.2 shows a sample stimulus for the Towers of Hanoi problem.

4.2.3 Experimental Protocol

We recruited participants via in-class invitations and online class discussion forums. Participants were asked to read and sign the general consent form prior to their scheduled 60-minute experimental session. Each session had three components: pre-questionnaire survey, eye-tracking session, and post-questionnaire survey and debriefing.

4.2.3.1 Pre-Questionnaire Survey

After participants re-affirmed their consent, we administered a survey to collect basic demographic data (e.g., gender, native language, class standing, etc.). To measure the incoming preparation for participants, we collected data on the theory-related courses they had completed or were enrolled in, and asked participants to complete a screening question from a widely-used undergraduate discrete mathematics textbook.

4.2.3.2 Eye-Tracking Session

Participants were seated in front of a computer screen with a Tobii Pro X3-120 eye tracker in a quiet room with controlled ambient light and screen brightness levels. Participants were encouraged not to look away from the computer screen to help reduce noise in the gaze data. We first calibrated the eye-tracker for each participant. We then showed the participants training slides explaining the study design and purpose. Participants were informed they would be reading several algorithmic proofs from an undergraduate textbook and determining whether or not each proof is correct. Each participant was presented with 7 stimuli. All stimuli were presented within the interface provided by Tobii Pro Lab [97], and participants selected their answers via key presses.

4.2.3.3 Post-Questionnaire Survey

After the eye-tracking session, we instructed participants to complete a post-questionnaire survey and asked them to self-report (on a 1–5 Likert scale) their prior perceived experience with computer science formalisms, difficulty of tasks they were asked to perform, and helpfulness of different aspects of the formalism presentation. Note that while it is common to ask participants to self-report their experience prior to the start of the experiment, we include such questions in the post-questionnaire to mitigate potential decrease in performance due to stereotype threat [231]. This is especially relevant for pedagogy since stereotype threat is reported to disproportionately affect underrepresented groups and students with non-traditional backgrounds [232] that may already struggle with outcomes in theory courses. In addition to the Likert scale data, we also collected qualitative responses from participants on attributes that make a formalism comprehension task easier to complete.

4.2.4 Data Collection

We conducted all experiments on a 64-bit Windows 10 machine connected to a 27 inch monitor with a 1920x1080 resolution. To collect eye gaze data, we used the Tobii Pro X3-120 with an external processing unit, a non-invasive eye tracker that can detect fixations at the granularity of a single line of 10pt text. Our eye tracker was set to sample readings at a frequency of 120Hz (i.e., 120 readings per second). We processed this raw data using Tobii Pro Lab to generate analyzable gaze data.

4.3 Analysis Approach

In this section, we present the mathematical analyses applied to our eye-tracking and functional data. We applied a *false discovery rate* (FDR) threshold at $q < 0.05$ to correct for multiple comparisons (i.e., to avoid false positives as a result of repeated analyses). All reported measures of statistical significance in Section 4.4 correspond to p-values corrected for multiple comparisons.

To preprocess for data quality, we filter outlier data points by removing the responses that were keyed in too quickly (outside 1.5×SD of the mean response time) and therefore, could not reasonably correspond to the participants reading a formalism presentation entirely before selecting an answer. We also filter out data points that correspond to noisy gaze data [183, Sec. 7.1]. This filtering resulted in 191 out of the original 236 data points being usable for experimental analyses.

Following the Goldberg and Helfman guidelines [94] for defining AOIs in terms of size and granularity, we manually divide each presented stimulus into six AOIs: *Algorithm*, *Theorem*, *Proof*, *Figure*, *Correct Answer*, and *Distractors*. The *Algorithm* AOI represents the pseudocode algorithm of interest, including the inputs and outputs of the algorithm and any explanatory comments. The *Theorem* and *Proof* AOIs represent the prose text for the theorem and the proof respectively. The *Figure* AOI corresponds to a graphical illustration of the formalism comprehension task and includes relevant captions and labels. Finally, the *Correct Answer* and *Distractors* AOIs represent the multiple choice responses.

We analyze raw eye-movement data to detect velocity-based fixations (I-VT) [233], a commonly-used fixation extraction method in the research community [234]. We use the following standard metrics to analyze and compare the strategies employed by participants for the formalism comprehension tasks. A *strategy* models gaze data and visual attention trends over time for the duration of a task. The *fixation time* corresponds to the total duration of all fixations on an AOI, while the *fixation count* indicates the total number of fixations on an AOI. Longer fixation times indicate either higher levels of interest or increased difficulty, and as such, increased strain on the working memory, in extracting information from the AOI [90, 235]. The *regression rate* depends on saccadic eye movements and indicates the percentage of backward saccades [236], and higher regression rates indicate increased difficulty in completing a task [96]. The *attention switching* metric depends on fixation counts and measures the total number of switches between AOIs, and can approximate the dynamics of visual attention during a task [90].

Previous work has argued for the use of baseline pupil diameters [102], and we used the training slides administered after eye-tracking calibration to measure the baseline pupil diameter (and as such, approximate the difficulty completing a task) for participants prior to working on the formalism comprehension tasks.

4.4 Results

We consider the following research questions:

RQ 4.1 What is the relationship between incoming preparation and student outcomes for formalism comprehension tasks?

RQ 4.2 How do student self-reports of formalism comprehension tasks align with empirical results?

RQ 4.3 What factors most distinguish higher-performing individuals from lower-performing ones?

Table 4.1 outlines the independent and dependent variables for each RQ, including the metrics used for each variable. Explanations of key terms and eye-tracking metrics in a software engineering context follow in the relevant RQ subsections.

Table 4.1: Formalism comprehension independent and dependent variables with associated metrics. Descriptions of key terms follow in the relevant RQ subsections.

	Independent Variables	**Metrics: Independent Variables**	**Dependent Variables**	**Metrics: Dependent Variables**
RQ 4.1	Incoming preparation	Coursework count, screening proof performance	Task performance	Response times and accuracy
			Visual attention	Fixation times on AOIs
			Task difficulty	Regression rates for proof types
RQ 4.2	Self-perceived experience	Likert scale	Task performance	Response accuracy
	Self-perceived task difficulty			
	Self-perceived helpfulness of figures			
	Self-perceived proof readability			
	Visual attention to pseudocode	Fixation time		
	Visual attention to figures and proofs			
RQ 4.3	Proof type	Categorical (inductive, contradictive, and direct proofs)	Task performance	Response accuracy
	Algorithm type	Categorical (recursive, iterative, and non-repeating algorithms)		
	Visual behavior	Attention switching count		
	Difficulty completing the task	Pupil diameter		

Table 4.2: Mean response accuracy, response time, and fixation times on different AOIs for more-prepared and less-prepared participants. Response and fixation times are given in seconds, while response accuracy is shown as a percentage. Fixation times are abbreviated using FT.

	Mean (SD)		
	More-prepared	**Less-prepared**	p
Response Time	248.7(±109.6)	240.0(±105.3)	0.93
Response Accuracy	34.8(±17.3)	32.5(±16.1)	0.96
Algorithm FT	19.5(±16.2)	17.1(±17.4)	0.21
Correct Answer FT	**1.3(±2.0)**	**0.8(±1.2)**	**0.038**
Distractor Choices FT	**14.6(±12.3)**	**11.1(±11.5)**	**0.03**
Figure FT	11.0(±45.8)	10.9(±38.6)	0.88
Proof FT	**66.8(±12.3)**	**46.1(±11.5)**	**0.005**
Theorem FT	12.9(±2.0)	11.2(±1.2)	0.12

4.4.1 RQ 4.1: Role of Incoming Preparation

We examine the relationship between incoming preparation of participants and outcomes for the formalism comprehension tasks. We consider two facets of preparation: coursework count and performance. First, for *coursework count*, we enumerate the number of computer science theory courses covering formalisms (i.e., courses that include proofs and derivations in their syllabi) that participants had either completed with passing grades or were taking. Second, for *screening proof performance*, we asked participants to identify a mistake in a proof distinct from the stimuli used in the study, and noted whether the participant correctly identified the mistake. Both facets we consider have been used previously in the context of pedagogy to approximate incoming preparation [26, 226]. While we note that factoring in grades for the theory courses would result in a more accurate approximation of incoming preparation, instructors for upper-level courses typically only require a student to pass the prerequisite courses, and do not know how well students did in the core courses. As such, we do not consider grades as a proxy for incoming preparation. We classify a participant who had taken above the median number of theory courses (i.e., coursework count > 4 for our dataset) and passed the screening question as *more-prepared*. Applying our approximations for incoming preparation resulted in 16 out of 34 participants being classified as more-prepared, with the remaining 18 deemed less-prepared.

The mean *response accuracy* (i.e., percentage of correct answers) and *response time* (i.e, time taken to choose an answer) for more-prepared and less-prepared participants are shown in Table 4.2. Surprisingly, we found no evidence of a statistically-significant difference in the outcomes between more-prepared and less-prepared students, both in terms of response accuracy (two-tailed Mann-Whitney U-test with the Benjamini-Hochberg [237] procedure to correct for false discoveries, $p = 0.96$) and response time ($p = 0.93$). Notably, while absence of evidence is not evidence of absence, the effect sizes for both results were extremely small (Cohen's $d = 0.007$ for response accuracy and $d = 0.08$ for response time), giving statistical confidence in the null result (i.e., even if an effect were present, it would be of very low magnitude and thus unlikely to influence outcomes).

We further found no correlation between the number of theory courses taken and response accuracy (Pearson's $r = 0.036, p = 0.84$), nor a correlation between participants' self-perceived experience with formalisms (on a 1–5 Likert scale) and response accuracy (Kendall's $\tau = 0.21, p = 0.18$). Our results indicate that, contrary to conventional wisdom [26, 225, 226, 227] and instructor expectations, students with greater incoming preparation perform no better at these formalism tasks, on average, than students with lower incoming preparation.

These results have potentially major implications, both for pedagogy and the training of new hires for formal software engineering. On the pedagogy front, our results raise questions about course design and undergraduate curricula: upper-level undergraduate courses are often designed with the expectation that students will have completed, and will be familiar with, material covered in core courses. If students with more exposure to the formal material do not show evidence of retention over time, educators may need to reconsider upper-level course design with more of an emphasis on reviewing relevant material covered in lower-level courses. For high-assurance software engineering, some managers may be tempted to make hiring and training decisions based on the number of theory courses taken (e.g., from a transcript or resume). Our results add confidence that regardless of the number of relevant courses taken, new hires for high-assurance software engineering should be considered for training to ensure that they are prepared for the challenges of the job, and that managers should not default to "courses completed" as a proxy of preparation for the job.

Even though participants have similar final outcomes, they employ different strategies. An analysis of visual behaviors between students with different incoming preparation reveals that more-prepared students *fixate longer* on (i.e., spend more time looking at) AOIs corresponding to the proof (two-tailed Mann-Whitney U-test with the Benjamini-Hochberg procedure, $p = 0.005$, Cohen's $d = 0.49$), correct answer ($p = 0.038$, Cohen's $d = 0.29$), and distractor answer choices ($p = 0.03$, Cohen's $d = 0.29$). The mean fixation times for all six AOIs for more- and less-prepared participants are included in Table 4.2. Our results suggest that while more incoming preparation teaches students to read a proof and the associated answer choices thoroughly before selecting an answer, this increased attention to the AOIs is not actually correlated with better student outcomes.

Recall that *regression rate* is the ratio of backward or regressive saccades to the total number of saccades. Quite surprisingly, we observed that students with greater incoming preparation show a higher regression rate (i.e., spend more time re-reading text and figures) — and as such, increased difficulty [96] — while reading direct proofs (Mann-Whitney U-test, $p = 0.035$, Cohen's $d = 0.73$). In particular, we observed that more-prepared participants exhibited a mean regression rate of 0.54 ± 0.09 for direct proofs, while less-prepared participants exhibited a mean regression rate of 0.42 ± 0.09. Given that we found no evidence of a statistically-significant difference in the performance of students with different incoming preparation for direct proofs, our results suggest that, for our sample, students may be trained in theory courses to default to induction (see Section 4.4.3) or contradiction as proof strategies, and may need to put in more mental effort when analyzing a direct proof — a style that remains highly relevant in formal methods for software engineering (e.g., [215]).

The results from our study suggest that traditional metrics for incoming preparation, like course counting and pretests, are ineffective predictors of student performance with formalism comprehension tasks, and that students across the board are not well-trained to employ different tools for evaluating presented logical deductions for correctness. Indeed, the two most-prepared participants in our study had the lowest and second-lowest response accuracies, and the highest-outcome participants were more junior. Our results provide confidence that students with more exposure to the formal material may not show evidence of retention over time. As such, it may benefit students if educators focus on upper-level course design strategies that encourage reviewing relevant material from lower-level undergraduate courses. The trade-off between using a few lectures to ensure students who took core courses several semesters ago retain key concepts and exposing students to novel topics is worth evaluating.

We found no evidence that students with higher incoming preparation, as traditionally assessed, perform better at formalism comprehension tasks (Mann-Whitney U-test, $p = 0.96$). This suggests the need for pedagogical intervention in core theory courses to ensure student outcomes are better aligned with course objectives of having students master and retain the material and better preparing them for formal methods in software engineering.

4.4.2 RQ 4.2: Self-Reporting and Formalism Comprehension Tasks

To collect richer free-response data from our study, we instructed all 34 participants to provide answers to a post-questionnaire reflecting on their experiences with the study and outlining what they thought to be the most important parts of a formalism presentation. In addition to having participants self-report (on a 1–5 Likert scale) the difficulty of tasks they were asked to perform, the helpfulness different aspects of the formalism presentation, etc., we asked three free-response questions:

1. Having completed the study session, would you do anything different the next time around?
2. What is the most important thing that makes a proof easier to understand?
3. What is the most important thing that makes it easier to spot a mistake in the proof?

To better understand the relationship between participant Likert scale responses and *response accuracy*, we use the Kendall's τ test to conduct a quantitative analysis of the data. When analyzing participants' *self-reported experience* with formalisms, we found no evidence of a correlation between experience and response accuracy ($\tau = 0.21, p = 0.18$). We also observed no correlation between *self-reported task difficulty* and study outcomes ($\tau = 0.14, p = 0.35$), nor a correlation between the *self-reported helpfulness of figures* for formalism comprehension tasks and study outcomes ($\tau = -0.22, p = 0.13$). Our results do not provide evidence that students are accurate at self-reporting their experience with formalism comprehension tasks.

To further investigate whether student self-perception is an accurate predictor of factors associated with high outcomes, we also performed a qualitative analysis on the participants' self-reported free-response data. 26 out of 34 participants indicated they would employ a different problem-solving strategy if asked to do the study again. Tied for the most common change in strategy were paying more attention to the algorithmic pseudocode and reviewing the materials from core theory courses prior to the study. Our experimental results, on the other hand, do not indicate a relationship between *fixation time* on algorithmic pseudocode (i.e., time spent reading pseudocode) and higher response accuracy (Mann-Whitney U-test, $p = 0.91$; see Section 4.4.3). The desire to review materials from core theory courses is aligned with our experimental results: students with greater incoming preparation do not perform better, suggesting a lack of retention of course materials over time, and hence, preparation for high-assurance software.

When asked to describe the features that make a proof easier to comprehend, about a third of the participants mentioned concise, easy-to-read English prose in the proof. The second most popular answer (6/34 participants) corresponded to the use of figures and visuals while reading the proof. Interestingly, our empirical results do not show evidence of a significant correlation between *self-perceived proof readability* and outcomes for formalism comprehension tasks (Kendall's $\tau = -0.14, p = 0.32$), or a statistically significant relationship between increased *fixation* on (or attention to) figures and response accuracy ($p = 0.81$).

In response to the traits that make a mistake in a proof easier to spot, the most popular answer (7/34 participants) focused on step-by-step logical reasoning. The second most common answer themes (6/34 participants each) were logical inconsistencies in the proof text and an understanding of the proof strategy. By contrast, only one participant answered "thinking through a different worked example", a strategy that is commonly taught in undergraduate theory courses. Our results suggest that students could benefit from a repertoire of more effective tools for evaluating logical deductions for correctness. The student-perceived traits (i.e., breaking down logical reasoning steps and evaluating the reasoning for logical inconsistencies) remain effective strategies for formalism comprehension tasks. However, the lower outcomes for these tasks suggests that students are less able to apply those strategies in a mistake-finding context.

We find no evidence that students experience reports are accurate predictors of outcomes for comprehending formalisms ($\tau = 0.21, p = 0.18$). We also find no evidence that the factors identified by students are associated with high outcomes for such tasks.

4.4.3 RQ 4.3: Factors Associated with Higher Outcomes

Given the apparent lack of effect of incoming preparation on the outcomes for our study, we are interested in investigating the factors that are correlated with better performance for formalism comprehension tasks. To do so, we perform a sub-population analysis of students with higher and lower outcomes. We require a participant to have achieved above the median response accuracy (i.e., $\geq 3/7$ answers correct for our dataset) to be classified as higher performing. Using this metric, we classify 15 out of 34 participants as *higher performing*, with the remaining 19 considered *lower performing* participants. Only 7 out of the 16 more-prepared participants (Section 4.4.1) were classified as higher performing.

We examined the outcomes for higher and lower performing students for different proof categories (inductive, contradictory, and direct) and algorithm categories (recursive, iterative, and non-repeating[2]). We found that, independent of algorithm category, higher performing students are more likely to spot mistakes in proofs by induction than the lower

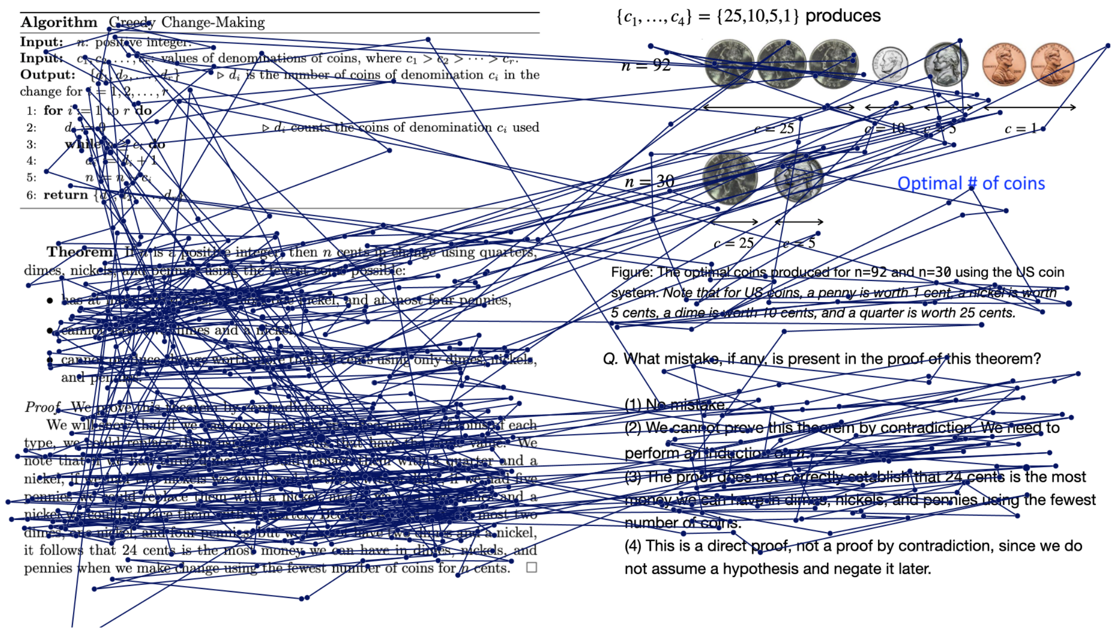
[2]The only algorithm in our stimuli that does not involve loops or recursion corresponds to a proof

performing ones (χ^2 test with the Benjamini-Hochberg procedure, $p = 0.01$). Endres *et al.* [238] have investigated student performance for iterative and recursive problem formulations, and observed poorer student performance for recursive algorithms involving non-branching computation but better student performance for recursive algorithms involving array manipulation. We find that independent of problem or proof type, higher performing students are more likely to get proofs for recursive algorithms correct compared to lower performing students (χ^2 test, $p = 0.006$).

These results have implications for both teaching formalisms for improved outcomes and preparing students for formal methods in software engineering. Previous work by Polycarpou [239] suggests that students who understand recursive or inductive definitions can more successfully perform proofs by induction, while students who do not are either not able to perform proofs by induction, or do so mechanically. The fact that students with higher outcomes in our study are not necessarily those with greater incoming preparation suggests that undergraduate theory courses taken by our participants may not be putting emphasis on teaching and making students comfortable with recursive or inductive definitions. In formal verification for high-assurance software, there has been an increasing interest in automated theorem proving (e.g., the Z3 theorem prover [240]), an activity that often involves the identification of an *inductive invariant* to prove that a certain property holds at all times [241]. Finding an inductive invariant has a direct parallel to correctly identifying an inductive hypothesis for proofs by induction, suggesting that students who are better trained to correctly establish inductive proofs may be better equipped for automated theorem proving tasks.

Recall that *attention switching* measures the total number of switches between AOIs, and can approximate the dynamics of visual attention during a task. We found that higher performing students demonstrate more *attention switching* behaviors, or frequently go back and forth between AOIs on the presented materials (two-tailed Mann-Whitney U-test with the Benjamini-Hochberg procedure, $p = 0.002$, Cohen's $d = 0.56$), with mean attention switches of 62.5 ± 33.1 and 45.9 ± 26.5 for higher and lower performing participants respectively. In particular, we observed a statistically-significant difference in attention switching for proofs by contradiction ($p = 0.009$; 59.8 ± 32.5 and 38.0 ± 22.7 mean attention switches for the two population groups) and iterative algorithms ($p = 0.007$; 65.7 ± 26.9 and 49.5 ± 28.1 mean attention switches). We also observed trends for increased attention switching for higher-performing students for proofs by induction and recursive algorithms, but these trends did not survive correcting for multiple comparisons. Figure 4.3 shows two illustrative visual gaze plots for a higher outcome and another lower outcome participant for a stimulus in-

gadget used for the Halting Problem. For completeness, we consider "non-repeating" a separate category of algorithms.


Algorithm Greedy Change-Making

```
Input:  n: positive integer.
Input:  c_1, c_2, ..., c_r: values of denominations of coins, where $c_1 > c_2 > \cdots > c_r$.
Output: $\{d_1, d_2, \ldots, d_r\}$    ▷ $d_i$ is the number of coins of denomination $c_i$ in the change for $i = 1, 2, \ldots, r$
1: for $i := 1$ to $r$ do
2:     $d_i := 0$                     ▷ $d_i$ counts the coins of denomination $c_i$ used
3:     while $n \geq c_i$ do
4:         $d_i := d_i + 1$
5:         $n := n - c_i$
6: return $\{d_1, d_2, \ldots, d_r\}$
```

Theorem. If n is a positive integer, then n cents in change using quarters, dimes, nickels, and pennies using the fewest coins possible:

- has at most two dimes, at most one nickel, and at most four pennies,
- cannot have two dimes and a nickel,
- cannot produce change worth more than 24 cents using only dimes, nickels, and pennies.

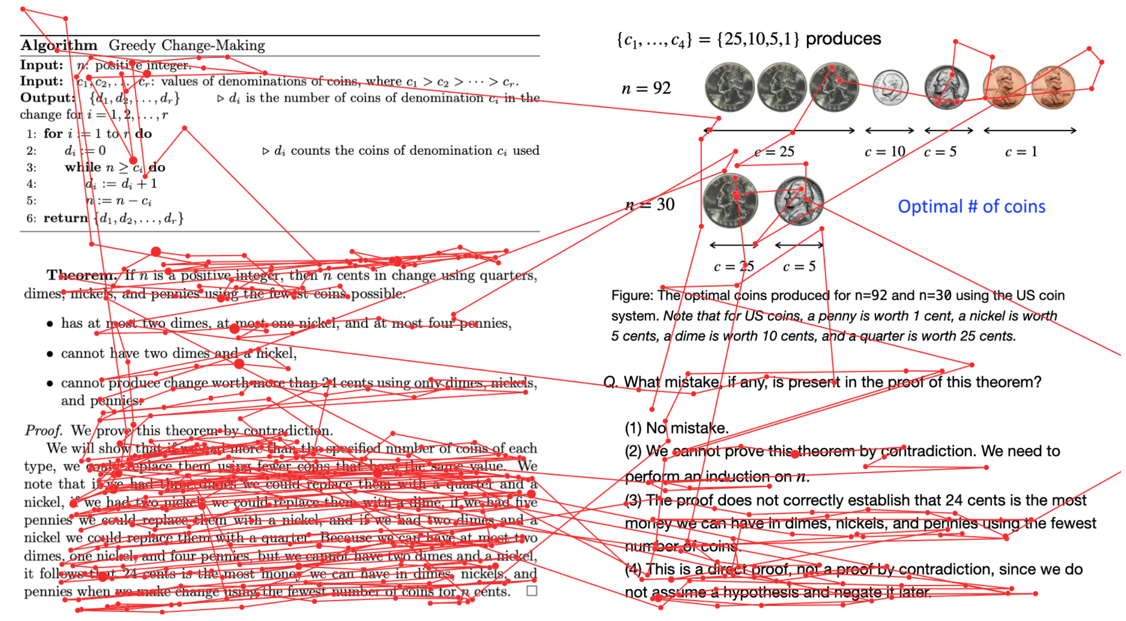
Proof. We prove this theorem by contradiction.
We will show that if we had more than the specified number of coins of each type, we could replace them using fewer coins that have the same value. We note that if we had three dimes we could replace them with a quarter and a nickel, if we had two nickels we could replace them with a dime, if we had five pennies we could replace them with a nickel, and if we had two dimes and a nickel we could replace them with a quarter. Because we can have at most two dimes, one nickel, and four pennies, but we cannot have two dimes and a nickel, it follows that 24 cents is the most money we can have in dimes, nickels, and pennies when we make change using the fewest number of coins for n cents. □

Figure: The optimal coins produced for n=92 and n=30 using the US coin system. *Note that for US coins, a penny is worth 1 cent, a nickel is worth 5 cents, a dime is worth 10 cents, and a quarter is worth 25 cents.*

Q. What mistake, if any, is present in the proof of this theorem?

(1) No mistake.
(2) We cannot prove this theorem by contradiction. We need to perform an induction on n.
(3) The proof does not correctly establish that 24 cents is the most money we can have in dimes, nickels, and pennies using the fewest number of coins.
(4) This is a direct proof, not a proof by contradiction, since we do not assume a hypothesis and negate it later.

(a) Higher-outcome participant gaze plot


Algorithm Greedy Change-Making

```
Input:  n: positive integer.
Input:  c_1, c_2, ..., c_r: values of denominations of coins, where $c_1 > c_2 > \cdots > c_r$.
Output: $\{d_1, d_2, \ldots, d_r\}$    ▷ $d_i$ is the number of coins of denomination $c_i$ in the change for $i = 1, 2, \ldots, r$
1: for $i := 1$ to $r$ do
2:     $d_i := 0$                     ▷ $d_i$ counts the coins of denomination $c_i$ used
3:     while $n \geq c_i$ do
4:         $d_i := d_i + 1$
5:         $n := n - c_i$
6: return $\{d_1, d_2, \ldots, d_r\}$
```

Theorem. If n is a positive integer, then n cents in change using quarters, dimes, nickels, and pennies using the fewest coins possible:

- has at most two dimes, at most one nickel, and at most four pennies,
- cannot have two dimes and a nickel,
- cannot produce change worth more than 24 cents using only dimes, nickels, and pennies.

Proof. We prove this theorem by contradiction.
We will show that if we had more than the specified number of coins of each type, we could replace them using fewer coins that have the same value. We note that if we had three dimes we could replace them with a quarter and a nickel, if we had two nickels we could replace them with a dime, if we had five pennies we could replace them with a nickel, and if we had two dimes and a nickel we could replace them with a quarter. Because we can have at most two dimes, one nickel, and four pennies, but we cannot have two dimes and a nickel, it follows that 24 cents is the most money we can have in dimes, nickels, and pennies when we make change using the fewest number of coins for n cents. □

Figure: The optimal coins produced for n=92 and n=30 using the US coin system. *Note that for US coins, a penny is worth 1 cent, a nickel is worth 5 cents, a dime is worth 10 cents, and a quarter is worth 25 cents.*

Q. What mistake, if any, is present in the proof of this theorem?

(1) No mistake.
(2) We cannot prove this theorem by contradiction. We need to perform an induction on n.
(3) The proof does not correctly establish that 24 cents is the most money we can have in dimes, nickels, and pennies using the fewest number of coins.
(4) This is a direct proof, not a proof by contradiction, since we do not assume a hypothesis and negate it later.

(b) Lower-outcome participant gaze plot

Figure 4.3: Visual gaze plots for a stimulus shown to two participants with different outcomes. The higher-outcome participant (top) displays significantly more attention switching behaviors, as indicated by the number of lines crossing between different AOI quadrants.

volving proof by contradiction. The higher-outcome participant displays significantly more attention switching behavior (as indicated by the increased number of lines between the AOI quadrants, see Figure 4.3a) compared to the lower-outcome participant (Figure 4.3b). Notably, we see no evidence of a statistically significant difference in the response times for the higher performing (with a mean response time of 258.0 ± 106.6 seconds) and lower performing (with a mean response time of 234.8 ± 107.4 seconds) participants (Mann-Whitney U-test, $p = 0.17$, Cohen's $d = 0.22$), adding confidence that the greater attention switching exhibited by the higher performing participants was not an artifact of more time spent on the proof comprehension task. In the context of pedagogy, these results argue for exploring the use of teaching materials (e.g., online tools, lecture slides, and exams) that facilitate perusal with ease (i.e., without requiring multiple page flips).

To estimate the difficulty associated with completing the formalism comprehension tasks (e.g., due to the need for remembering the theorem or variable names while evaluating the proof for correctness), we record the pupil diameters reported by the eye tracker for each stimulus and obtain the pupil diameter delta against a measured baseline (see Section 4.3). We found that students with poorer performance also show increased perceived difficulty going over the figures in general (Mann-Whitney U-test, $p = 0.012$). In particular, we saw trends for increased difficulty for lower outcome students looking at figures for inductive proofs and recursive algorithms, though the latter did not survive correcting for multiple comparisons (Mann-Whitney U-test, $p = 0.032$ and $p = 0.06$ respectively). These results indicate that students who got inductive proofs incorrect more frequently (in a statistically-significant manner) also had stress associated with going over figures explaining the algorithm or proof strategy when compared to their higher-performing peers.

Given that higher-outcome participants exhibited significantly more attention switching, we consider whether current educational materials admit this sort of problem-solving strategy. For instance, our results argue against exams that require page flips to get relevant information, since turning or scrolling between pages is not conducive to going back and forth between presented materials with ease (e.g., [242, Sec. 6.2.1]). In an era of an increasing number of online exams and teaching tools, similar concerns arise: while administering online lectures, quizzes, and exams, educators could benefit from exploring placing relevant bits of information in a spatially-proximate manner for a formalism comprehension task, and contrast the approach against requiring page scrolling or user interface navigation to gather information before answering a question. Our results suggest that the former may yield more favorable student outcomes.

Additionally, since we observe differences in performance depending on the type of algorithm or proof, we advocate for increased emphasis on certain types of proofs. Notably, our study shows that differences in outcomes can be attributed, in a statistically-significant manner, to proofs by induction. Previous work has shown that students' performance with proofs by induction improves after class instruction, but not to the extent intended during course design [239], yet strategies involving proofs by induction remain highly relevant in formal verification of software (e.g., the use of inductive invariants in automated theorem proving [241]). We encourage educators to emphasize familiarity and comfort with recursive and inductive definitions and reasoning.

Higher-performing students are more likely to get proofs by induction (χ^2 test, $p = 0.01$) and recursive algorithms correct (χ^2 test, $p = 0.006$) compared to lower-performing students. Higher-outcome students also demonstrate significantly more attention switching behaviors (Mann-Whitney U test, $p = 0.002$), suggesting that students who frequently go back and forth between presented materials are more likely to achieve better results.

4.5 Threats to Validity

In this section, we outline internal and external validity concerns associated with our study.

Generalizability. Our results may not generalize to a wider population. To mitigate this threat, we recruited participants from a large public university with a wide array of different backgrounds (including native language, incoming preparation, class standings, etc.). We also note that the primary goal of our study is to understand how to better teach formalisms at the undergraduate level (and indirectly, to shed light on hiring considerations for certain software engineering sectors), and thus, recruiting seasoned industry professionals is less relevant.

Choice of study tasks. While we select study tasks to have a parallel to activities involved in high assurance software engineering (e.g., producing machine-checkable proofs for formal verification), we acknowledge that our choice of assessments and task difficulty could be producing a floor effect for participant performance, where participant response accuracy is clustered around lower scores. We encourage researchers to investigate formalism comprehension with different proof comprehension tasks (e.g., completing an existing correct proof) to verify the effect of incoming preparation and expertise on formal reasoning outcomes.

Stimuli construction. Our stimuli construction may introduce researcher bias. To mitigate this threat, we select our stimuli from an undergraduate textbook widely used by educators, and supplement any figures from undergraduate course lecture slides serving thousands of students each year.

Eye-tracking data noise. We use state-of-the-art software to calibrate the eye-tracker [97] and widely-used eye-tracking metrics and analyses [90] to minimize noise in collected data.

Pedagogical suggestions. Our suggestions for pedagogy are not based on controlled human studies investigating the impact of the interventions. Instead, in this thesis, we illuminate promising interventions and advocate for their evaluation in controlled contexts before implementation in a classroom setting.

4.6 Related Work

Previous eye-tracking work investigating problem-solving strategies employed by students has shown that differences in search strategies can lead to significant differences in task outcomes [243, 244, 245].

For instance, Netzel *et al.* found that high-accuracy students were better able to use information in science-related diagrams [243]. We similarly observed that high-performing participants display reduced cognitive load when going over figures related to formalism comprehension tasks. Hegarty *et al.* [244] investigated arithmetic problem solving involving relational terms inconsistent with the required arithmetic operator (e.g., the use of *less than* for tasks involving addition), and found that low-accuracy students made more reversal errors for inconsistent problems, and that high-accuracy ones required more re-readings for previous text fixations. Our study does not reveal a difference in response accuracy between high- and low-outcome students for proofs by contradiction (that involve logical inconsistencies in proof text). We do find, however, that higher-outcome students display significantly more attention switching as they assimilate presented information, a strategy that is analogous to re-reading text.

Figures are frequently used as educational instruments [230, 246, 247], yet their importance as a medium of instruction for a particular field is not always well-understood. For instance, Susac *et al.* [245] found that diagrams were rarely helpful for physics. By contrast, Yoon *et al.* [248] studied the importance of figures for causal reasoning problems, and found that even for questions missing a figure, 48% of the students still frequently fixate on the area where the figure would have been, indicating a relative higher importance for figures. For both studies, however, the inclusion of diagrams did not affect the participants' time taken to respond or response accuracy. In a mistake-finding context for formalism comprehension, we similarly do not find a relationship between fixation on, or perceived importance of, figures on task outcomes.

4.7 Chapter Summary

Formal methods (i.e., mathematical logic) have been increasingly applied to software engineering, but often require mathematical training and advanced logical reasoning abilities that software practitioners often do not possess. Given the challenges associated with integrating formal methods into software, educators are increasingly focusing on formalisms in undergraduate theory courses that already suffer from unsatisfactory student outcomes.

In this chapter, we use eye-tracking to better understand the problem solving strategies for formalism comprehension employed by students with different levels of incoming preparation, and more indirectly, gather insights into how educators can prepare future software engineers for the rigorous logical reasoning that is a core part of high-assurance software engineering.

In a controlled human study involving 34 participants, we find that incoming preparation is not an accurate predictor of task outcomes ($p = 0.96$), that student experience reports and self-perceptions are not effective at predicting task outcomes ($p = 0.18$), and that the increased attention to proof text by more-prepared students is not correlated with higher task outcomes ($p \leq 0.04$). We instead find that students who exhibit more attention switching behaviors are more likely to succeed ($p = 0.002$), and that differences in formalism comprehension outcomes can be attributed to performance for proofs by induction and recursive algorithms ($p \leq 0.01$). Our results advocate for pedagogical interventions in theory courses to better prepare future programmers for formal reasoning for software. We make our datasets publicly available for researchers to replicate or build on our study.

In the next chapter, we explore the use of transcranial magnetic stimulation (TMS) to codify the relationship between spatial reasoning and program comprehension (i.e., programming logic).

CHAPTER 5

Programming and Spatial Reasoning

In recent years, neuroimaging studies — studies that non-invasively measure brain activity — have been used by researchers to pinpoint the brain regions most correlated with common programming activities (i.e., programming logic). Such activities include code comprehension [41, 42, 44, 249], code reading and writing [43], debugging [250, 251], and data structure manipulation [45, 46]. These studies, and subsequent work, identified key cognitive processes correlated with programming tasks.

For example, data structures are often described spatially (e.g., "balanced", "length", "height", etc.), suggesting a potential relationship between how humans reason spatially and how they reason about data structures. Spatial reasoning (or spatial visualization) refers to the ability to mentally manipulate three dimensional objects. On this front, Huang *et al.* confirmed a *correlative* relationship between spatial visualization and data structure manipulation [45]. Such studies have shown the potential to improve our understanding of expertise, to inform pedagogy, and to guide tool development and retraining (see Floyd *et al.* [43, Sec. II-D] for a summary).

Despite these potential benefits and despite researcher interest, to the best of our knowledge, no prior neuroimaging study in computer science has confirmed a *causal* relationship between patterns of neural activation and programming activities. Specific causal relationships from one variable to another cannot usually be assessed from an observed association between them [252, 253] (cf. "correlation is not causation", confounds, etc.).

In this chapter, we propose to probe causality at the neural level for spatial reasoning and program comprehension tasks (i.e., programming logic) using transcranial magnetic stimulation.

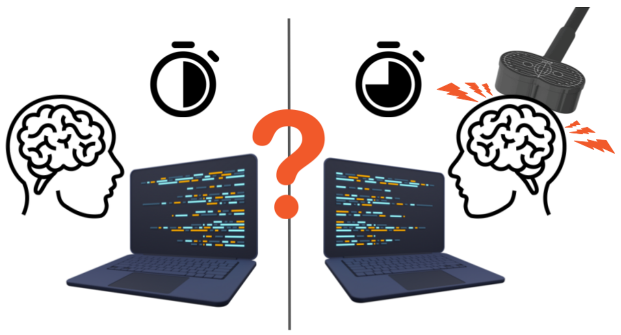


Figure 5.1: High-level TMS experimental architecture: "Does impairing brain regions associated with spatial reasoning influence programming outcomes?"

5.1 Overview of Experimental Design, Results, and Contributions

To scientifically investigate the plausible existence of a causal connection between spatial visualization and programming tasks, we require an approach that limits the effects of any confounding variables: manipulating and influencing brain regions directly. The desired approach should: (1) admit high-confidence causal inference, (2) comprise a non-invasive process, and (3) apply to indicative programming tasks.

We propose the first investigation of a causal relationship between spatial reasoning and programming tasks. We use transcranial magnetic stimulation (TMS) to non-invasively and directly impede or facilitate visualization-associated regions [254, Sec. 5] of the brain and then analyze the effects on programmer performance. Neurostimulation via TMS induces a current within a region of the brain itself, temporarily changing transmembrane potentials and causing neurons to be more or less excitable (see Section 2.4.1 for a discussion on how TMS works). Unlike neuroimaging methods, such as fMRI or fNIRS, that indicate correlations between brain and behavior, TMS can be used to demonstrate causal brain-behavior relations [255, Sec. 3][256, pp. 595–596]. Stimulation that interferes with task performance indicates that the affected brain region is necessary for the task (i.e., establishes causality).

We applied TMS to 16 participants, disrupting three regions of their brains to probe causal relationships between programming and neural activity. The regions were stimulated on different days via an established TMS protocol (Section 5.2.3) and state-of-the-art per-subject brain region localization (Section 5.2.3.1). To the best of our knowledge, this is also the first such study in computer science to feature multiple treatments and visits, more naturally admitting both within-one-subject and between-multiple-subjects analyses (cf. previous computer science neuroimaging replications using different subjects each time [41]). After TMS, subjects completed a randomized set of 180 tasks: code comprehension, data structure manipulation, and mental rotation. Differences in outcomes (e.g., time) give confidence in a causal relationship (see Figure 5.1).

Care is necessary to avoid bias as we probe causal relationships. We *pre-register* hypotheses (mitigating some threats from researcher bias), correct for multiple comparisons (mitigating some threats of false discovery), use special active controls (mitigating some threats of participant response bias [257]), and conduct some analyses with condition labels anonymized (mitigating some threats from researcher bias). In addition, with over 1,600 minutes of neurostimulated performance, our study involves comparable observation to correlative studies (e.g., 1,300 [43], 600 [41], etc.).

The main contributions of this chapter — separately published in the 46th ACM/IEEE International Conference on Software Engineering [258], with an ACM Distinguished Paper Award — are:

- A controlled human study ($n = 16$) investigating the first use of neurostimulation in a computer science context.
- Experimental evidence that suggests a lack of causal relationships for multiple previously-published correlations (e.g., for code understanding [41, Sec. 5.1], data structures [45, Sec. V.B], code complexity [249, Sec. III.B], or code writing [46, Sec. 5.2]).
- A replication of prior findings that stimulation of supplementary motor area degrades mental rotation task completion time, giving confidence that we applied TMS correctly.
- Experimental evidence showing that TMS treatment condition contributes to time outcome dispersion among participants. Via multi-level regression analysis, we find that TMS accounts for 2.2% of the variance in task completion times.
- A summary of lessons learned and costs associated with a neurostimulation study of programming, along with a publicly available de-identified dataset for future replication and exploration.

5.2 Experimental Setup and Methods

We present our study design for investigating the causal link between neural activity and programming via TMS. Each individual underwent a localizing (fMRI) scan and two to four subsequent TMS sessions, each on a different day. At each TMS session, an experimental condition was applied: stimulation of one of two spatial reasoning-associated regions or stimulation of an active control (leg-associated) region. After treatment, participants were tested on a set of stimuli. This design allows for a controlled investigation of a potential causal link between spatial reasoning and program comprehension for programmers.

5.2.1 Participant Recruitment

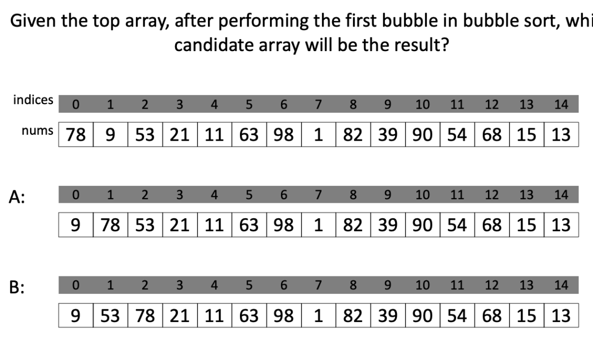
We recruited 16 participants via a combination of email, course forums, posters, and in-class presentations under UM IRBMED-HUM00216195. Eligible subjects were required to be 18 years or older, be right-handed, be native English speakers, have normal to corrected vision, and have at least 1.5 years of programming experience. Due to TMS safety policies, participants were also required to pass a medical screening form. Participants whose medical history indicated any neurological risk factors, drugs active in the central nervous system (e.g., antipsychotics, antidepressants, or recreational stimulants), or poor levels of sleep were excluded from the study [259, 260]. We note that the risks associated with TMS are minimal, with only one known case of a seizure [259].

Because individual humans vary slightly in brain anatomy [261], we obtained anatomical MRI images for each individual to produce a personalized localization. We collected 23 brain scans, of which 16 are part of our final analysis (others dropped out or failed later safety screenings). Additionally, data from one participant was removed from the final analyses due to inconsistencies and outlier data points (i.e., response times more than 2 standard deviations away from the mean). Overall, our final analysis considered 16 participants: 8 male and 8 female.

Background and demographic information were collected from 14 out of 16 participants. 6 of our participants reported being undergraduate students, 4 as graduate computer science students, 2 as industrial professionals (software engineers), 1 as a non-computing student, and 1 as a non-computing-related professional. All subjects were also screened for basic programming knowledge of C++. Participants who completed the study in full, which consisted of a localizing anatomical scan and three subsequent TMS sessions, received $125.

5.2.2 Stimuli and Tasks

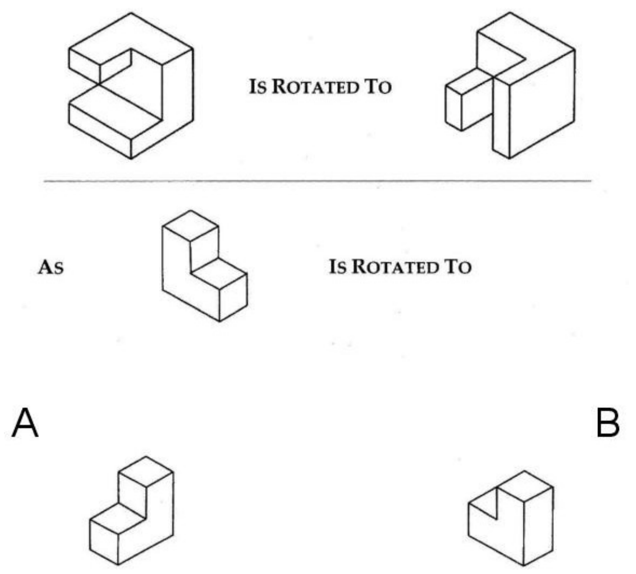
After each TMS treatment, participants were shown a varied set of 61 stimuli from three tasks: Code Comprehension, Data Structure Manipulation, and Mental Rotation. We selected stimuli that were short and concise to fit well within the 60-minute effect window of the TMS treatment (see Section 5.2.3.1). Data structure and mental rotation stimuli were acquired from previously-published studies that examined spatial visualization and program-



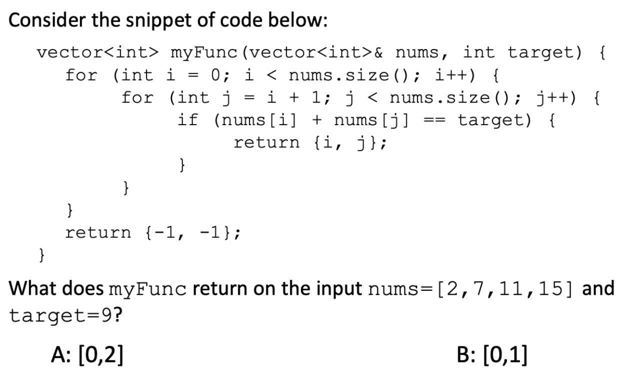
Given the top array, after performing the first bubble in bubble sort, which candidate array will be the result?

indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
nums	78	9	53	21	11	63	98	1	82	39	90	54	68	15	13
A:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	9	78	53	21	11	63	98	1	82	39	90	54	68	15	13
B:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	9	53	78	21	11	63	98	1	82	39	90	54	68	15	13

(a) Data structure manipulation stimulus



(b) Mental rotation stimulus



Consider the snippet of code below:

```
vector<int> myFunc(vector<int>& nums, int target) {
   for (int i = 0; i < nums.size(); i++) {
        for (int j = i + 1; j < nums.size(); j++) {
             if (nums[i] + nums[j] == target) {
                  return {i, j};
             }
        }
   }
   return {-1, -1};
}
```

What does `myFunc` return on the input `nums=[2,7,11,15]` and `target=9`?

A: [0,2]   B: [0,1]

(c) Code comprehension stimulus

Figure 5.2: An example TMS stimuli shown to participants. Data structures stimuli include linked lists and trees, while code comprehension stimuli also cover Big-O complexity.

ming [45, 47] and thus relate to our research questions. Code comprehension stimuli were taken from previous quizzes and exams administered in a data structures and algorithms course at a large public university in the US. Responses to each stimulus were given by selecting one of two answer choices via the '*A*' or '*B*' keys on a standard laptop keyboard. Stimuli were administered via the open-source PyschoPy (version 2022.2.5) package [262]. Individual tasks took 15–60s to complete, with 35 minutes to complete all 61 stimuli. We now describe the stimuli in further detail.

5.2.2.1 Data Structure Manipulation Task

We obtained a total of 89 validated data structure task stimuli from a prior publication reporting a neural correlation with computer science tasks [45]. Stimuli covered arrays, linked lists, and trees. Each stimulus included a starting data structure, an operation to perform, and two answer choices (Figure 5.2a). Answers were either numerical values to describe the outcome of an operation or candidate data structures resulting from an operation. The tree tasks included binary search tree (BST) rotation, insertion, and traversal operations.

5.2.2.2 Mental Rotation Task

We use both the Huang *et al.* [45] and the Endres *et al.* [47] spatial skills stimuli. These include Mental Rotation Stimulus Library questions established by Peters and Battista [263] with varying rotational angle difficulty as well as the Revised Purdue Spatial Visualization Test (PSVT:R II) [264]. PSVT:R II is a standard assessment of different facets of spatial ability. Mental rotation tasks asked participants to compare two 3D objects rotated about an axis (Figure 5.2b). Participants selected the object that matched the starter object, accounting for rotation. Our stimuli included 56 distinct mental rotation tasks.

5.2.2.3 Code Comprehension Task

Code comprehension tasks were acquired from exams and quizzes for a data structures and algorithms course at the University of Michigan, a large public university. All tasks have previously been used to assess thousands of undergraduate students on their knowledge of data structures. For each stimulus, participants were asked to trace through snippets of C++ code and select one of two answer choices (Figure 5.2c). Tasks included deducing the values printed or returned by a function, and analyzing the time and memory complexity of the code. A total of 38 distinct code comprehension stimuli were included in our study.

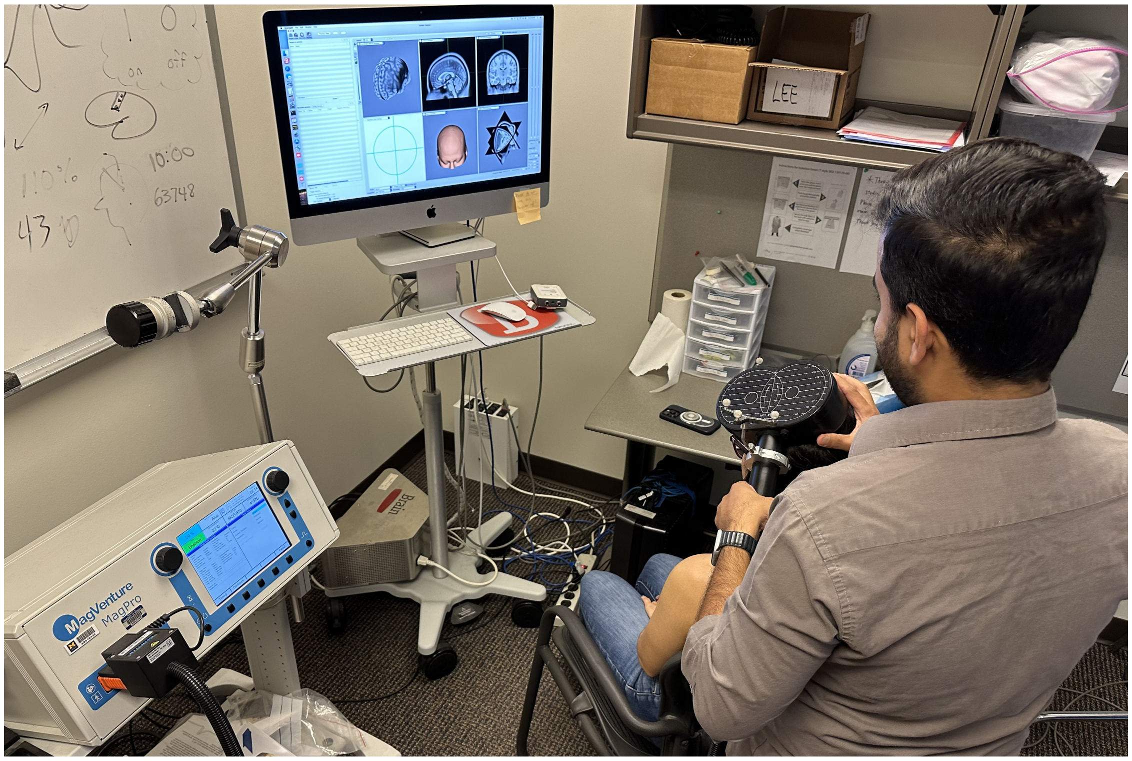


Figure 5.3: Transcranial magnetic stimulation treatment setup. We use per-participant localization (top screen) and the hand-held magnetic coil (center right) on the scalp of the participant (seated center) to induce current in a brain region.

5.2.3 TMS Treatment

We summarize our experimental design decisions at a high level. We claim no novelty in the mechanics of TMS application — indeed, we intentionally use a high-quality and established TMS protocol (see Figure 5.3) for this application in programming. In brief:

1. "How do we apply TMS at all?" We use a best-practice protocol and off-the-shelf hardware and software (Section 5.2.3.1).
2. "How much TMS do we apply?" Following best practices, we find per-participant stimulation thresholds (Section 5.2.3.2).
3. "Where do we apply TMS?" Following best practices, we measure each participant's individual brain anatomy and target brain regions implicated in previous correlative studies (Sections 5.2.3.3 and 5.2.3.4).
4. "How do we minimize bias?" We use a best-practice active control in which an unrelated brain region is stimulated (in a process that still feels like other TMS treatment, Section 5.2.3.3). We randomize treatment conditions and stimuli and blind conditions when possible (Section 5.2.3.5).

Knowledge of TMS details (e.g., theta burst stimulation) is not necessary to understand our results or their import. TMS can be viewed as an effective black box that temporarily impairs brain regions (see Section 2.4); the remainder of this section provides details relevant for replication and justification of best-practice decisions.

5.2.3.1 Stimulation Protocol

We applied a continuous theta burst stimulation (cTBS) protocol consisting of 3 pulses of stimulation at 50 Hz, repeated every 200 ms, for a total of 600 pulses in 40 seconds. The method is an accepted form of stimulation in various psychology and medicine research papers studying TMS effects [113, 265]. This method is effective in providing long-lasting effects of approximately 60 minutes [113]. This is essential for our experiment, since effects should last long enough to complete the 35-minute task block presented after TMS treatment, but short enough to limit effects post-study to mitigate safety concerns. The time to complete this protocol is drastically smaller than that of other stimulation protocols (e.g., [266]), facilitating recruitment.

We used a well-established stimulation procedure to maximize accuracy and time [113]. cTBS was delivered over the scalp through a MagPro X100 magnetic stimulator and a 90 mm figure-8 coil (MC-B70, MagVenture Inc.). The cTBS protocol was tolerated well by all subjects with no negative side effects reported.

5.2.3.2 Thresholding

Because each human is slightly different, we use a well-established protocol to determine an appropriate stimulation intensity for each participant. We first find the participant's individualized active motor threshold (AMT) for the first dorsal interosseous muscle (FDI) of the right hand as they contract the FDI [114, 115]. This common method involves stimulating the primary motor cortex on the left hemisphere at various levels with the aim of eliciting a motor evoked potential (MEP) of $\geq 50\ \mu$V peak-to-peak on five out of ten trials while the participant is subjectively contracting the FDI muscle at 20% of maximum. A stimulation threshold that meets such requirements is known as the AMT and allows us to effectively stimulate each participant safely [114, 115]. In most subjects, the lowest stimulation threshold can be found in this manner [114]. To ensure accurate recording of MEPs and AMT, the participant is adjusted with disposable self-adhesive electromyograph (EMG) on their right hand. EMG activity was amplified ($x1000$) with a BioAmp (AD Instruments, USA) using a Powerlab 4/35 system and digitized (10 kHz) and recorded using "Brainsight TMS" neuronavigation software (Rogue Research, Montréal, Canada). Physiological responses were visually monitored because twitches near or around the FDI of the right hand can indicate if stimulation is occurring at the correct positioning [259]. Once AMT was determined for the participant, cTBS stimulations were applied at 80% AMT to comply with commonly-accepted safety standards [260, 259].

5.2.3.3 Treatment and Control Conditions

Participants were stimulated in multiple brain regions to assess the causal relationship between neural activity and programming. In particular, we stimulated the *primary motor cortex (M1)* (reported as correlated with code understanding [41, Sec. 5.1], data structures [45, Sec. V.B], and code complexity [249, Sec. III.B]) and the *supplementary motor cortex (SMA)* (reported as correlated with code writing [46, Sec. 5.2]). The left primary motor cortex was chosen as the motor sub-area for stimulation since all participants were required to be right-handed.

To ensure that any changes observed in the participant are caused by the stimulation, as opposed to some other general factor (e.g., arousal, attention, altering response to the TMS sounds), we apply an *active control* condition in which the *cranial vertex* (a leg-associated brain region) is stimulated. The vertex region is a commonly-used control in TMS studies with cTBS protocols [267, 268]. Introducing an active control is shown to provide the same sensation of TMS stimulation without affecting the brain areas of interest [269, 270, 271, 272, 273, 274, 275]. An active control thus further mitigates participant response bias [272, 257]. In total, participants were stimulated in three different brain regions, one on each TMS session in randomized order.

5.2.3.4 Stimulation Localization

Every brain is slightly different [261], so we collected individual 3D brain scans to accurately target stimulation on each participant. While some studies report localizing brain anatomy by sight or by feel, we used an fMRI to collect high-resolution imaging following best TMS localization practices [272]. All imaging procedures were conducted on a 3T General Electric MR750 with a 32-channel head coil at the University of Michigan Functional MRI Laboratory. Participants attended a single 45-minute scanning session for brain region localization. High-resolution anatomical images were acquired with a T_1-weighted spoiled gradient recall (SPGR) sequence ($TR = 2300.80$ ms, $TE = 24$ ms, $TI = 975$ ms, $FA = 8°$: 208 slices, 1 mm thickness). We obtained estimates of the static magnetic field using spin-echo fieldmap sequences ($TR = 7400$ ms, $TE = 80$ ms; 2.4 mm slice thickness).

Subjects' heads were reconstructed in 3D using the Brainsight TMS neuronavigation software from their T_1 anatomical scans and the locations of the left primary motor cortex (M1), SMA, and cranial vertex were determined for stimulation. The M1 region was identified using axial scans by locating the "hand knob" and hook in MRI images [260, 276, 277, 278, 279]. The SMA region was located by selecting the voxel in individual anatomical scans best corresponding to the Brodmann area definition for pre-SMA and SMA (Talairach coordinates $x = -28, y = 0, z = 48$) [280, 281] (cf. [282]). The cranial vertex control region was located by selecting the intersection of an abscissa between the nasion and the inion, and an abscissa between the left and right tragus on individual structural brain scans [283, 272]. Localization methods used were overseen by two independent TMS experts, adding confidence.

Localized regions were marked for stimulation as targets via Brainsight's frameless stereotaxy system which uses an infrared camera for monitoring head locations of the participant by tracking reflexive markers attached to the head of the participant [284, Sec. 2]. Head locations are then related to the structural MRI brain data of the participant, guiding precise positioning of the magnetic coil.

5.2.3.5 Minimizing Bias

In addition to our use of an active control (see Section 5.2.3.3), we took additional steps to reduce bias. First, participants were not informed of which brain region was being stimulated at the time of the session. Second, participants were not given information on the expected effects [257]. This was single-blind, not double-blind, since the researcher manually targets the TMS coil at the brain region and thus knows the treatment condition. After each TMS stimulation was applied, participants were presented with randomized task stimuli on an automated, online platform which required no interaction with the researchers. A final post-test survey was administered on a printed page. We believe that these (non-)interaction procedures help minimize threats associated with participant response bias.

5.3 Analysis Approach

We analyze our results via statistical assessments and modeling. Critically, unlike fMRI-based computer science papers (which must use nuanced methods to account for large numbers of noisy voxels, etc., when analyzing brain scans, e.g., [43, Sec. IV]), our primary analyses are of the broad form "did the participants in the treatment condition answer the test questions better (or faster) than those in the control condition?". While some modeling sophistication is required (e.g., to account for heterogeneity, see Section 5.3.2), we do not analyze brain scan data for our research questions.

However, to form robust experimental conclusions, especially involving "null" results, we must minimize the potential for bias, including researcher bias during analysis. In addition to approaches taken in our experimental protocol (e.g., randomization, single-blind, etc., see Section 5.2) we also follow two practices in our analysis: pre-registering our hypotheses and partial blind analysis.

5.3.1 Pre-Registration and Bias

Pre-registration is a scientific process in which the "research rationale, hypotheses, design and analytic strategy" are submitted before beginning the study [285]. This helps mitigate biases associated with researchers choosing which results to present post hoc: "pre-registration can prevent or suppress HARKing, p-hacking, and cherry picking since hypotheses and analytical methods have already been declared before experiments are performed" [286]. Similarly, following a discipline of pre-registration may mean that "researchers will not be motivated to engage in practices that increase the likelihood of making a type I error" [285]. While not as common in computer science (but see the "Registered Reports" track of Mining Software

Repositories [287], for example), pre-registration is increasingly adopted by journals and researchers, especially in fields such as psychology and social science (e.g., [288]).

Our hypotheses, such as "TMS stimulation in the SMA or motor cortex will significantly disrupt accuracy and or reaction times on both mental rotation and programming tasks compared to an active control condition (TMS stimulation in the vertex)", were pre-registered with the Open Science Framework (https://osf.io/m4p6e) along with our data collection strategy and statistical analysis methods. This includes our criteria for excluding data and inferring significant correlations.

In addition, our final analysis was conducted blind: labels representing the treatments (vertex, SMA and M1) were randomly coded as `A`, `B`, and `C`, before the analysis strategy was set. This helps mitigate researcher bias in the choice of analysis tools or methods.

Finally, the Benjamini-Hochberg (BH) adjustment was used to correct for multiple comparisons when necessary in evaluating p-values [289]. Prior work has shown the choice of statistical software is important [290]: our analysis primarily used the R package `lme4` and the Python package `scikit-learn`.

5.3.2 Multi-level Regression Analysis

Our experimental design produces item-level assessment data, where each response to a question contributes an observation to the dataset. We broadly follow the framework of *Item Response Theory* (IRT), a branch of psychometrics which is concerned with the analysis of this type of data [291, 292]. Specifically, we employ multi-level regression models to examine relationships between a response variable, stimulated brain region, and control variables. We linearly model response time and self-reported perceived difficulty, and logistically model accuracy. This is collectively referred to as multi-level regression analysis, or mixed-effects modeling.

We claim no novelty in statistics, and focus our discussion on why this analysis appropriately incorporates important aspects of our data and research hypotheses. For general information about our methods, we refer the reader to Bates *et al.* [293] and Faraway [294, Ch. 8].

5.3.2.1 Suitability of Multi-level Regression

Multi-level regression analysis is well-suited to handling heterogeneity between groups of observations, such as those that arise from repeated measures [295]. In our experiment, each participant response (to 150–183 stimuli) is a distinct data item; these may be correlated due to an underlying person-dependent "skill". We also have repeated measures for each stimulus, as multiple participants answer every question; such observations may be correlated due to variation in question difficulty. We can also posit heterogeneity between content domains (e.g., code comprehension vs. data structures). Such considerations are common in the analysis of item-level assessment data [296, 297]. Multi-level models also perform well with unbalanced group sizes. Our experiment has modest imbalance (e.g., 843 observations of SMA stimulation vs. 939 of M1). Moreover, not all questions have responses from all participants (e.g., from drop-out).

Multi-level regression analysis allows us to test hypotheses about both systematic and heterogeneous TMS effects, as discussed below.

5.3.2.2 Systematic and Heterogeneous Effects

A mixed-effects model can include independent variables whose effects are systematic (*fixed effects*), heterogeneous (*random effects*), or both. *Interactions* of fixed effects further permit modeling effects that are systematic within specific groups of observations. This is relevant because we hypothesize a systematic TMS treatment effect within each programming task (e.g., data structures vs. code comprehension).

Random effects can pertain to multiple levels of *grouping* in the data. For example, they can model heterogeneity between people, between person-domains, or both. This is relevant because we hypothesize a heterogeneous TMS treatment effect that varies between people, as has been found in TMS studies of other disciplines [114, 115, 298, 299, 300, 301, 302, 303, 304]. That is, some people may improve performance under TMS while others reduce performance.

We are interested in the TMS effect distribution over the population represented by our study subjects. This is mirrored in our experiment design, which features person-specific localization (Section 5.2.3.4) and person-specific TMS intensity thresholding (Section 5.2.3.2). Mixed-effects models can express our hypothesized person-dependent TMS effect using a random effect that describes interactions (combinations) of TMS conditions and participants.

5.3.2.3 Model Specification, Parameter Estimation, and Inference

The dependent variables we consider are per-question accuracy, per-question response time, and perceived difficulty. We first consider plausible effect structures for the available independent variables, based on existing literature and our experimental design [297]. Examples are given in Section 5.3.2.2 (see replication package linked in Section 5.3.3 for the full list).

We apply logarithmic transformation to response times to address skew (discussed in Sections 5.4.1 and 5.4.3). All models are fit by maximum likelihood estimation (MLE) to the programming and mental rotation data separately. To find the best-fitting candidate model for each dependent variable, we optimize Akaike Information Criterion (AIC), a widely-used model selection metric [305, 306].

We are interested in the TMS treatment condition (i.e., which brain region was stimulated), which may exhibit a fixed or a random effect. If the best-fitting model has a fixed (systematic) TMS effect, we explicitly verify statistical significance via a likelihood ratio "omnibus" test relative to a model without the TMS effect [294, Appendix A.2]. We then pinpoint the source using post-hoc pairwise contrasts, with Benjamini-Hochberg adjustment for the 3 comparisons. Alternatively, if the best-fitting model has a random (heterogeneous) TMS effect, we explicitly verify statistical significance using profile likelihood analysis [293] and parametric bootstrap methods [307] to find the 95% confidence bounds of the statistic.

5.3.3 Replication

Our replication package (publicly available at https://github.com/hammad-a/ICSE24_TMS) contains raw data for de-identified participants and relevant analysis information, including scripts, data management, and statistical assumption checking.

5.4 Results

With behavioral and survey data, we ask:

RQ 5.1 Can we replicate prior findings that neurostimulation of the SMA reduces mental rotation completion times?

RQ 5.2 Is there a direct causal relationship between activity in the SMA or M1 brain region alone and performance?

RQ 5.3 Does neurostimulation of the SMA or M1 brain regions affect objective computing performance outcomes?

RQ 5.4 Does neurostimulation in the SMA or M1 brain region affect self-perceived problem difficulty?

5.4.1 RQ 5.1: TMS and Mental Rotation

The supplementary motor area (SMA, Broadmann area #6) is a part of the frontal cortex and coordinates complex and internally-guided motor actions for extremities. The primary motor cortex (M1, Broadmann area #4) is in the anterior bank of the precentral sulcus and is involved in the execution of voluntary, external body movements (such as contracting skeletal muscles).

Prior psychology studies using TMS found a causal link between the SMA and mental rotation, but no such link for the M1 [308]. To gain confidence in the accuracy of our results regarding the SMA, M1, and computing, we attempt to replicate this causal link between the SMA and mental rotation.

We find that TMS stimulation of the SMA impairs response time for spatial reasoning stimuli, compared to TMS of the vertex region (our active control condition). With $p \leq 0.02$, TMS stimulation of the SMA results in an increase of 0.143 log-seconds in expected per-question log-transformed response time[1] (a 15.3% increase in raw response time, or 1.5 s slower on our median response time of 9.82 s) relative to stimulation of the vertex region.

We replicate a prior study showing that stimulating the SMA influences spatial reasoning performance ($p \leq 0.02$), adding confidence in our correct administration of TMS.

5.4.2 RQ 5.2: SMA, M1 and Computing

Overall, we find *no evidence of a causal relationship* between activity in the supplementary motor area and computing outcomes. In particular, we find no question type for which accuracy in the SMA treatment condition and accuracy in the control condition are statistically different ($p \geq 0.81$). Similarly, there is no question type for which response times for the SMA treatment condition and time taken in the control condition are statistically different ($p \geq 0.22$). We also find *no evidence of a causal relationship* between activity in the primary motor area and computing outcomes for any question type ($p \geq 0.50$ for accuracy, $p \geq 0.73$ for time taken). The mean response accuracy and response times (prior to log-transformation) across different TMS treatments are shown in Table 5.1.

Quite surprisingly, our results do not agree with multiple previously-established correlations. For instance, for the SMA region, Siegmund *et al.* found a correlation between brain activity and code understanding [41, Sec. 5.1], Huang *et al.* found a correlation between activity and data structure manipulation [45, Sec. V.B], and, most recently, Peitek *et al.* found a correlation between neural activity and comprehension of code with higher complexity metrics [249, Sec. III.B]. Likewise, for the M1 region, Krueger *et al.* found a correlation between activity and code writing (as opposed to prose writing) [46, Sec. 5.2]. The lack of evidence of a causal relationship between single-region brain activity and computing outcomes calls into question the research community's understanding of cognition for programming tasks. Simply disrupting activity in one region does *not* uniformly result in lower outcomes. Our results suggest that interpreting cognition for programming is complex: multiple brain regions could be causally responsible for outcomes for programming tasks (cf. [309]).

[1]We log-transform the dependent variable to address right skew in raw response times (skewness 3.18 $\rightarrow$ 0.35) and residuals of the optimal-AIC fitted model (2.93 $\rightarrow$ 0.59).

Table 5.1: Mean response accuracy and times across TMS conditions. Response times are given in seconds, while response accuracy is shown as a percentage.

	Mean (SD)		
	M1	**SMA**	**Vertex**
Response Time (Overall)	20.4(±17.2)	20.5(±16.2)	20.3(±18.3)
Response Accuracy (Overall)	95.1(±7.2)	94.7(±6.7)	94.4(±7.5)
Response Time (Data Structures)	22.4(±17.2)	21.8(±16.8)	21.7(±17.8)
Response Time (Mental Rotation)	14.0(±13.4)	15.5(±14.1)	14.2(±15.2)
Response Time (Code Comprehension)	25.4(±19.6)	25.1(±15.9)	26.2(±21.0)
Response Accuracy (Data Structures)	93.8(±8.9)	93.3(±9.4)	92.6(±10.7)
Response Accuracy (Mental Rotation)	95.3(±9.9)	95.2(±7.5)	95.0(±8.4)
Response Accuracy (Code Comprehension)	92.9(±14.5)	93.2(±9.9)	93.9(±7.9)

Our null results further argue for nuance in pedagogical interventions based on cognition. Indeed, a recent investigation by Endres *et al.* [47] concluded that student training based on spatial visualization produced worse results than technical reading, a result not in line with prior correlative studies (e.g., [45]). The lack of a causal relationship between brain regions associated with spatial reasoning and programming outcomes helps further explain these recent results, and cautions against misdirected research and pedagogical interventions that may otherwise be undertaken if correlation and causation are confused and not thoroughly investigated (e.g., via a controlled study).

We fit a multi-level linear model to mental rotation responses (see Section 5.3.2). Critically, our optimal-AIC model contains the TMS condition as a fixed effect. We calculate post-hoc pairwise contrasts between TMS conditions (with Benjamini-Hochberg adjustment), to obtain the significance result ($p \leq 0.02$). This result generalizes over both types of mental rotation stimuli from prior work: we find no significant difference in the impact by stimulus source (see Section 5.2.2.2).

Of note, we also find no significant difference in response times between TMS stimulation of the M1 and control ($p = 0.18$). Our results thus replicate prior findings that TMS of the SMA impacts mental rotation response times, but that TMS of the M1 does not have a significant effect [308]. Our replication results give confidence that we have applied TMS correctly.

Table 5.2: TMS mixed-effects model parameter estimates predicting log-transformed response times. The "C.I." columns give the confidence interval for the standard deviation estimate of the corresponding random effect. Critically, the "Participant by Brain Region" interval (bolded) excludes 0, indicating a statistically significant person-specific effect involving TMS neurostimulation.

Random Effect	**Vari.**	**Std. Dev.**	**2.5% C.I.**	**97.5% C.I.**
Stimulus	.204	.452	.398	.517
Participant by Question Type	.019	.137	.100	.190
Participant by Brain Region	**.010**	**.099**	**.066**	**.143**
Participant	.037	.193	.118	.308
Residual	.175	.418	.404	.433

We find no evidence of a causal relationship between computing outcomes and activity in SMA ($p \geq 0.22$) and M1 ($p \geq 0.5$). Our results do not agree with multiple previously established correlations, warranting further exploration of the cognitive basis of programming.

5.4.3 RQ 5.3: TMS and Computing Outcomes

While our analyses for RQ 5.2 find no evidence of a monotonic causal relationship (e.g., "stimulating the SMA alone always reduces performance on data structure questions"), multi-level regression analysis finds that *TMS stimulation does have a statistically significant non-systematic person-dependent effect on response time.* Per Section 5.3.2, we produce a best-fit model of response times.[2] Equation 5.1 outlines our best fit model (presented using R syntax), and Table 5.2 shows point estimates and 95% confidence intervals.

$$\begin{aligned}\log(\texttt{Response Time}) \sim\ & \texttt{Question Type} * \texttt{Session Number} \\ & + (1|\texttt{Participant}) + (1|\texttt{Stimulus}) \\ & + (1|\texttt{Participant}:\texttt{Question Type}) + (1|\texttt{Participant}:\texttt{Brain Region})\end{aligned} \tag{5.1}$$

Equation 5.1 can be interpreted as a model for log-transformed response times that contains fixed (systematic) effects for question type, the session number (e.g., to model learning effects), and interactions between the two fixed effects. The model further contains random

[2]As with RQ1, we log-transform to reduce right skew in raw times (skewness 2.21 → 0.17) and model residuals (2.26 → 0.11).

(heterogeneous) effects for participant "ability" (e.g., some participants may have more expertise for programming than others), the stimulus presented (e.g., some questions may be more difficult than others), an interaction between participant and question type (e.g., some participants may be better at certain types of questions than others), and an interaction between participant and brain region stimulated (e.g., some participants may have different reactions to different TMS conditions than others).

The confidence interval for the standard deviation of the "Participant by Brain Region" (denoted as $(1|\texttt{Participant} : \texttt{Brain Region})$ in Equation 5.1) random effect excludes zero, indicating a significant effect. The estimated proportion of variance explained (PVE, equal to the variance of estimate interest divided by the sum of all variances) of this effect is 2.2%.[3] The 95% confidence interval for that figure is (0.7%, 4.0%), calculated using the methods in Section 5.3.2.

This is evidence for a person-dependent TMS effect that is heterogeneous. We note that any non-zero effect is important for a new intervention. This result is particularly exciting, since while TMS has successfully been used to improve performance in other domains (see Section 5.6), ours is the first study providing evidence that TMS can alter outcomes for programming tasks. Our results argue for further exploration of using TMS (e.g., with protocol that strictly excites brain activity) to improve computing outcomes.

We also note that there is no evidence for a systematic effect from certain TMS conditions that improves or impairs programming ability relative to other TMS conditions. That is, while the effects of SMA and M1 stimulation on programming question response times are different ($p = 0.028$), one is not overall better or worse at improving outcomes. This is expected from our protocol (which focused on demonstrating the possibility of any effect, not on positive-only effects).

We can interpret our result using a "difference-in-differences" approach from generalization theory [310]. Consider an arbitrary member of the population placed in two scenarios. In each scenario, they are presented with the same set of questions from our stimuli. We consider the subject's average response time in each scenario, with the set of questions large enough that residual variance is negligible. If the subject undergoes TMS stimulation to the same brain region in both scenarios, then the difference in average response times is zero (with probability 1). By contrast, if the brain regions stimulated differ, the "difference in differences" of log-transformed response times is 0.099, equivalent to a ratio of 1.10$\times$ between the two differences in raw response times.

[3]Informally, if we assign the random effect for stimulus on log-transformed response times a normalized value of 1.0, we see that the random effect for the interaction between participant and brain region stimulated is $\frac{0.010}{0.204} \times 1.0 = 0.049$. In other words, the random effect for the interaction between participant and TMS to a particular brain region is 4.9% that of question difficulty on log-transformed response times.

TMS has previously been shown to significantly improve or impair performance in many other fields (see Section 5.4), and our results extend this to programming. Although the effect size shows that the treatment condition (i.e., the region of the brain that is stimulated by TMS) accounts for 2.2% of the variance we see in response times, this is a more substantial report than it may first appear. Many papers on computer science interventions do not include effect size in their results at all, or omit comparisons to a non-intervention baseline: in a 2018 review of 129 papers on pedagogical interventions in computer science, none included an effect size [311].

Some published results of interventions in computer science that do include such information may report similar variance in outcomes to our results. For example, one longitudinal study by Cooper *et al.* [312] considered a two-week, full-day workshop completed by high school students, of which 7.5 hours were devoted to spatial skills for a treatment group. They report that "the treatment group improved by an average score of 1.06 [out of 16 APCS questions], and that this was significant at the $p = 0.07$ level" [312, Sec. 4.2]. Although our approach uses a very different methodology and the results are not directly comparable, we note that that initial study provided the basis for subsequent studies of hundreds of students [50] and is associated with a 1-credit spatial skills course at a large university [49] where it improved retention in the major and grade outcomes for other classes. A small effect in an initial study may lead to a useful intervention later.

Additionally, since our main goal was to determine if neurostimulation could impact computing performance, we selected a protocol that may help *or* hinder ability. However, other TMS protocols exist that solely excite regions of the brain and/or otherwise observe predominantly positive results (e.g., [300, 313, 314], see Section 5.6). As a concrete example, an intermittent protocol (rather than the continuous protocol used for this experiment) involving applying theta burst TMS for 2s every 10s may result in heightened neural signal transmission [315]. Future work should investigate whether such heightened transmission translates to improved outcomes on computing tasks.

In addition to varying the protocol, future studies would benefit from varying the target brain area. While this paper investigated spatial skills, other studies implicate that brain regions associated with working memory (e.g., the dorsolateral prefrontal cortex) or language skills (e.g., Wernicke's or Broca's areas) may be correlated with other programming activities [105, 47]. Having demonstrated the applicability of TMS neurostimulation to computing outcomes, we encourage investigating positive impacts on other tasks.

We find evidence that TMS stimulation accounts for a 2.2% variance in response time (statistically significant). Ours is the first study providing evidence that TMS can alter outcomes for programming tasks.

5.4.4 RQ 5.4: TMS and Self Perception

Following each TMS treatment, participants reported their subjective perception of the task difficulty, both in isolation and relative to the last session (if applicable), on a Likert scale. Overall, we find *no statistically significant evidence of differentiation in the subjective perception of participants* across all treatments and all question types ($p \geq 0.21$).

While we observed no differences in participants' subjective perception of task difficulty following treatments, we note that participants self-reports are generally not reliable [257], and TMS may still be influencing task difficulty without participant conscious perception. Conversely, the lack of perceived differences may remove a potential barrier to participant retention in future investigations of TMS-based treatments for programming (e.g., if TMS were to make tasks seem more difficult or frustrating, participant drop-out might be impacted, cf. [47, 312], etc.).

We do not find any evidence of differences in the subjective perception of task difficulty for participants across treatments and question types.

5.5 Threats to Validity

In this section we briefly summarize internal (e.g., did we apply TMS correctly?) and external (e.g., do our participants generalize to other populations?) threats, referencing earlier mitigation details.

Stimulation procedures. We adopt a well-established TMS protocol, supervised and approved by an outside TMS expert. Additionally, we use per-person treatment intensity in line with best stimulation practices. Further, we use individual anatomical brain scans for accurate brain region localization. Finally, our replication of a previously-published [308] non-computing TMS result (the impact of SMA stimulation on mental rotation, Section 5.4.1) gives confidence in aspects of interval validity (i.e., applying TMS correctly).

Participant bias. We use an active stimulation control session and did not convey expected TMS effects until after a participant completed all study sessions.

Tasks. We use stimuli validated in prior work and covering multiple distinct domains. We do acknowledge that the tasks considered may not generalize to other activities (e.g., pair programming), and plan such exploration for future research.

Population. Our participants (largely students) may not generalize to other populations (e.g., professional programmers). We partially mitigate this by observing each subject longer, strengthening within-one-subject analyses.

Training. We observe a statistically significant ($p < 0.01$) question type-dependent training/learning effect, which we account for in our data analyses.

Subject variability. We use multi-level regression analysis, a well-established method to effectively account for between-subject heterogeneity.

Researcher bias. We pre-registered hypotheses and methodology, conducted our preliminary analysis with anonymized labels, and corrected for multiple comparisons.

5.6 Related Work

In this section, we discuss other interventions impacting programming outcomes, contrasting them with TMS.

Neurostimulation represents a different, possibly orthogonal, mechanism for improving software developer abilities compared to standard approaches such as pedagogical structures (e.g., transfer training, tools, gamified or flipped classrooms), environmental factors and development methodologies at software jobs (e.g., work from home, Agile/Scrum), and the use of substances in software workplaces (e.g., Adderall, cannabis).

5.6.1 Pedagogy

Dozens of studies have investigated the benefits of the flipped classroom model (in which instruction/learning is completed externally and discussion is done during traditional lecture time to enforce concepts) in computer science pedagogy [316]. Similarly, gamified learning (in which elements of games, such as leaderboards or points, are used in class) has been studied to see how extrinsic rewards can motivate engagement of students [317].

There has been preliminary success with pedagogical interventions involving spatial reasoning and STEM outcomes [318, 50, 49]. Despite positive outcomes, pure spatial reasoning training in engineering or computer science educations has not been widely adopted. We believe that a more rigorous understanding of why spatial reasoning cross-training improves behavioral outcomes, and its costs and benefits, would make it easier for institutions to adopt.

5.6.2 Work Structure

The structure of offices and work hierarchies has been an ongoing and evolving topic broadly in the field of computer science for many decades, especially with the express goal of improving company or individual programmer productivity [319]. For example, since the COVID-19 pandemic, working from home has become more relevant, and a survey of 3,634 software developers and managers from Microsoft found that 68% perceived they were just as, or more, productive working from home [320]. Similarly, pair programming, a key component associated with Agile development and some pedagogical methodologies, has been linked to higher satisfaction and learning outcomes, fewer bugs, and better communication between software engineers [321, 322, 323].

5.6.3 Medication

Many individuals program with the aid of psychoactive substances, citing enhanced abilities or the alleviation of symptoms. For example, recent surveys and interviews of 801 and 26 professional programmers (respectively) in software workplaces who use such substances found that many who use cannabis while programming do so for enjoyment, but also to enhance creativity or brainstorming, while many who use stimulant medications do so for perceived enhancements for focus and specific focus-intensive software tasks such as debugging [324, 325]. Although many substances may improve abilities or health, administering them at a company level may have serious legal or health impacts (e.g., Adderall usage among people not diagnosed with ADHD has been linked to stress or the pressure to make tight deadlines [326]).

5.6.4 Intervention Summary

In contrast to such traditional interventions, TMS does not require the use of language, effort on the part of either a teacher or a student, or much time to use. If TMS is found to be effective in some capacity for computer science outcomes, it could be used as a non-pedagogical intervention in tandem with other instructional, structural, or medical interventions.

5.7 Costs and Subjective Experience

In this section, we outline the unique costs and considerations we encountered during our TMS study, emphasizing the differences from correlative studies that solely employ other methods (e.g., fMRI or fNIRS).

5.7.1 Participant Recruitment

Unlike fMRI or fNIRS, TMS protocols may preclude subjects with a history of seizures or anxiety-related disorders, as well as those reporting a lack of sleep the prior night. However, TMS does not suffer from fNIRS data quality issues from hair types (cf. [45]). TMS causal studies require participants to attend multiple sessions (treatment and control) on different days. Subjectively, we found the multi-session constraint to be challenging for recruiting.

5.7.2 Time

For both TMS (applied quickly in advance, lasting up to an hour) and fMRI and fNIRS (typically measured over an hour-long session), the effective interaction duration per session is similar. Critically, however, TMS is not limited to 60-second stimuli (unlike fMRI or fNIRS, which are limited by the hemodynamic response function [327]). We used short stimuli here for comparison to previous work, but future studies could use more complex programming tasks.

5.7.3 Costs

TMS and fNIRS offer cost advantages over fMRI in terms of both initial costs and operating costs. An institution with an fMRI lab often charges per scan (e.g., \$500 per hour [43]); a TMS or fNIRS machine can typically be used for free if present.

Our base experiment cost was \$2,200 (\$125 per participant for reimbursement, \$200 for electrodes); we elected to use high-quality fMRI localization (30 scan-minutes per participant, an additional \$4,000). Future work may investigate the necessity of fMRI-quality localization for computer science TMS treatments.

5.7.4 Researcher Training

Each research team member completed over 20 hours of training before being authorized to operate the TMS machine.

5.7.5 IRB Process

An Institutional Review Board or Ethics Board handles human study research at our institution. Depending on the review board's experience with neuroimaging or stimulation techniques, getting approval for a study with fMRI or TMS can require a substantial amount of time and effort. For reference, this TMS study involved a 24-page IRB application plus a 14-page consent form. Using fMRI to localize brain regions required an additional 4-page data protection and privacy plan (in the United States, brain scans are HIPAA-protected). From our first submission to approval, the IRB process took four months.

5.7.6 Lessons Learned

Subjectively, the most difficult aspects of the experiment were training and participant scheduling. Conducting thresholding sessions under time constraints and manually targeting the hand-held TMS magnetic coil required practice. Our multi-visit protocol amplified scheduling intersection challenges between researcher, TMS equipment and participant availability.

5.8 Chapter Summary

To the best of our knowledge, this paper is the first exploration of the *causal* relationship between program comprehension (i.e., programming logic) and neural activity via transcranial magnetic stimulation, a non-invasive technique well-established in the literature. Previous correlative findings have revealed intriguing connections between specific neural regions and programming tasks. These findings laid the foundation for enhanced understandings of expertise, pedagogy, and retraining. However, the absence of studies confirming the causal nature of these relationships has constrained their practical applications and interpretations in the real world.

We address causality by applying TMS treatment to 16 participants, directly targeting two indicative brain regions (M1 and SMA) known to exhibit correlative connections to programming tasks. We compare stimulation effects to participant performance on computing tasks, including data structure manipulation, mental rotation, and coding comprehension. We followed established, state-of-the-art TMS practices that were overseen by independent TMS experts. To mitigate bias, we used a special active control, pre-registered our hypothesis, conducted aspects of the experiment and analysis blinded, and correct for multiple comparisons.

We replicate prior psychology results that TMS impacts mental rotation (Section 5.4.1, $p \leq 0.02$) — supporting replication in science and giving confidence that we are applying TMS correctly. We find no evidence of a simple causal relationship: disrupting activity in M1 or SMA does not uniformly reduce outcomes on computing tasks (Section 5.4.2, $p \geq 0.22$) — results that do not agree with multiple previously-established correlations [41, 45, 249, 46] and suggest that interpreting cognition for programming is complex (cf. [309]).

Critically, we find that TMS has an effect on response time for data structure and code comprehension tasks. TMS accounts for 2.2% of the variance in observed outcomes, a statistically-significant effect (Section 5.4.3). This provides evidence that TMS (neurostimulation) can alter outcomes for programming tasks. Neurostimulation is a distinct approach from traditional pedagogy (e.g., it does not require a shared language, or indeed any communication at all) and has produced positive outcomes in computing-related areas (e.g., creativity, mathematics, etc.). Now that TMS has been demonstrated to impact programming outcomes, we look forward to future work investigating, and making real, the potential benefits of neurostimulation for programming.

With the conclusion of this chapter, we have presented all three research components investigating cognition for computational logic associated with this thesis. The next chapter summarizes the thesis and parts with future directions.

CHAPTER 6

Conclusion

Maintenance activities for hardware and software can correspond to up to 90% of programmer time, making the activities the most expensive stage of the development process. Since computers fundamentally do not reason like humans do, finding and fixing bugs is a cognitively-demanding and time-consuming task. Understanding programming behavior and cognition behind computational logic reasoning (with a focus, in this thesis, on *digital logic*, *mathematical logic*, and *programming logic*) can illuminate ways to help programmers reason about computational logic more effectively (e.g., through interventions in the classroom or re-training activities).

While previous research investigating cognition for computer science activities has shown potential to improve programming outcomes in the classroom, we show that using **non-intrusive and objective measures** for cognition to establish correlations, using a **non-invasive and time-efficient medical technique** in a computer science context to establish causation, and using **advanced statistical models** to account for programmer background can help us better probe how programmers understand computational logic.

Using three research components presented in this thesis, we demonstrate:

> It is possible to use objective measures (ranging from functional to physiological to medical) to obtain mathematical models describing programmer behavior and cognition for computational logic reasoning tasks, and these models can highlight prospective cognitive interventions for student training.

- In Chapter 3, we develop an automated program repair algorithm for hardware designs (i.e., digital logic) and investigate its use as a debugging assistant for programmers. Our framework, CirFix, makes use of readily-available artifacts included in the hardware design process (e.g., testbenches) to diagnose and repair defects in the circuit description. These repairs can then be shown to programmers for validation before the synthesis phase, reducing maintenance costs. CirFix makes use of our novel approaches to implicate faulty lines of code and to guide the search for repairs using the existing hardware development process.

Our evaluation of CirFix presents a new publicly available benchmark suite of 32 defect scenarios spanning 11 different Verilog projects. **CirFix produces plausible repairs for 21 out of 32 (65.6%) and fully correct repairs for 16 out of 32 (50%) of the Verilog defects** – with respect to the programmer-provided testbench – within reasonable resource bounds. Further, we evaluated the utility of our novel fault localization algorithm independent of our automated repair context via a controlled human study involving 41 participants. **We find a statistically significant preference** ($p \leq 0.003$) **for CirFix fault localization as a debugging aid in fixing multi-line defects, primarily in student applications** ($p \leq 0.02$).

- In Chapter 4, we use eye-tracking to better understand how programmers read and understand formal algorithmic claims (i.e., mathematical logic). Formal methods have been increasingly applied to software engineering, but often require mathematical training and advanced logical reasoning abilities that software practitioners may not posses. Further, undergraduate computer science courses covering formal reasoning often suffer from unsatisfactory student outcomes.

  In a controlled human study involving 34 participants, we find that **incoming preparation (as traditionally assessed) is not an accurate indicator of task outcomes** ($p = 0.96$), that **student experience reports and self-perceptions are not effective at predicting task outcomes** ($p = 0.18$), and that **more-prepared students tend to pay more visual attention to the proof text but do not achieve higher task outcomes** ($p = 0.005$). Teasing apart higher-performing students from the lower-performing onces, we find that **students who exhibit more attention switching behaviors are more likely to succeed** ($p = 0.002$), and that **differences in formalism comprehension outcomes can be attributed to performance for proofs by induction and recursive algorithms** ($p \leq 0.01$).

- In Chapter 5, we probe the neural link between spatial reasoning and program comprehension (i.e., programming logic) using transcranial magnetic stimulation, demonstrating the first use of the medical technique in a computer science context. While previous correlative findings have revealed intriguing connections between specific neural regions involved with spatial reasoning and programming tasks, such a causal relationship has not been confirmed.

  In a controlled human study involving 16 participants, we investigate causality by ap-

Table 6.1: Major peer-reviewed publications supporting this dissertation

Venue	**Title**
ASPLOS'22	CirFix: Automatically Repairing Defects in Hardware Design Code [130] (Chapter 3)
ICSE'23	How Do We Read Formal Claims? Eye-Tracking and the Cognition of Proofs about Algorithms [228] (Chapter 4)
TSE (Vol. 9, Issue 7)	CirFix: Automated Hardware Repair and its Real-World Applications [131] (Chapter 3)
ICSE'24	Causal Relationships and Programming Outcomes: A Transcranial Magnetic Stimulation Experiment [258] (Chapter 5; **ACM Distinguished Paper Award**)

plying TMS to two brain regions known to be correlated with programming (M1 and SMA) and associated with spatial reasoning, and one control brain region (vertex). We **replicate prior psychology results that TMS impacts mental rotation ($p \leq 0.02$)** — supporting replication in science and giving confidence that we are applying TMS correctly. **We further find no evidence of a simple causal relationship: disrupting activity in M1 or SMA does not uniformly reduce outcomes on computing tasks ($p \geq 0.22$)** — results that do not agree with multiple previously-established correlations. Finally, we find that **TMS can effect response time for programming tasks, accounting for 2.2% of the variance in observed outcomes** (statistically-significant). Ours is the first study providing evidence that TMS (or neurostimulation) can alter outcomes for programming tasks, warranting further exploration of the research area.

Table 6.1 lists major peer-reviewed publications supporting the results in this thesis.

Work in this thesis presents a systematic approach to better understanding how programmers reason about computational logic. Our approach:

- employs non-intrusive methodologies to investigate programmer behavior and cognition in a more ecologically-valid setting with minimal interference to user attention;
- uses objective measures ranging from functional (e.g., responses to questions) to physiological (e.g., tracking eye gaze movements) to medical (e.g., stimulating brain regions) to understand cognition for logical reasoning;
- produces context-specific models for a variety of comprehension tasks for computational logic;

- accounts for the incoming preparation or expertise of programmers in interpreting results.

Adapting results from other fields (e.g., neuroscience, psychology, hardware design engineering) and advanced statistical methods (e.g., multi-level regression analyses) allows us to better understand how programmers reason about computation logic. Results in this thesis also shed light on pedagogical interventions worth investigating before implementing them in the classroom to better teach students how to reason as computers do.

6.1 A Look to the Future

While the work presented in this thesis provides foundational understanding of how programmers reason about a variety of tasks involving computational logic, it also delineates future directions to obtain a more complete picture of logical cognition. In this section, we summarize future directions and opportunities for results presented in this thesis.

6.1.1 Understanding understanding: Digital logic

Hardware debugging remains a notoriously difficult task, with very limited tool support available to designers for debugging tasks. This difficulty extends to the classroom, where designers with less incoming preparation often struggle with finding and fixing hardware bugs at the simulation level (e.g., using Verilog).

Since the publication of results in Chapter 3, several follow-on studies from independent research teams extending CirFix or improving on our technique have been accepted in peer-reviewed venues (e.g., [328, 329]), highlighting the research community's interest in automated repair of hardware designs. Further, our work has garnered interest from industrial settings to be deployed locally. We believe that novel hardware APR approaches and industrial interest signal a start to the use of automated repair of hardware designs to reduce the maintenance costs and manual effort associated with the hardware process (cf. Getafix [330] and SapFix [331] at Facebook for software bugs).

Our results from Chapter 3 also suggest that fault localization approaches from techniques like CirFix can be helpful in a classroom setting for designers with varying levels of experience. Given the emergence of several automated repair techniques following CirFix, we encourage researchers to investigate the use of a debugging assistant with a plug-and-play fault localization module (i.e., different ways to implicate faulty lines of code) in a classroom setting. We firmly believe that such work, while outside the immediate scope of this thesis, has the potential to reduce the cognitive load associated with learning and debugging Verilog designs, particularly for programmers with less incoming preparation. Should results from such investigations appear promising, we hope see debugging assistants for hardware incorporated into the professional hardware designer workflow.

6.1.2 Understanding understanding: Mathematical logic

Formal (or mathematical reasoning) about hardware and software is becoming increasingly important as the complexity of safety-critical IT infrastructure around us increases. At the same time, such formal reasoning remains a very difficult task. Our results in Chapter 4 open up several research opportunities, both inside and outside the classroom.

Given that we find no evidence that traditional incoming preparation (e.g., taking more theory courses) yields better formalism comprehension outcomes, we see this as an opportunity for educators to re-evaluate course design with material retention and the cognitive load theory as a focal point. Indeed, similar efforts are already underway at the University of Michigan for an introductory computer science theory course. Further, our results indicating that inductive and recursive reasoning remain difficult for students may serve as an opportunity for educators to evaluate student outcomes for the traditionally challenging topics and re-allocate learning resources as necessary.

Even for more experienced programmers, mathematical reasoning about machine-checkable proofs of correctness in hardware and software is such a cognitively-demanding task that team managers often find it not a good use of programmer time. We believe that investigating cognition for such formal reasoning (e.g., a programmer trying to find an inductive invariant for a safety property) through objective, non-intrusive measures can be instrumental in finding ways to support such reasoning activities. Answering research questions like “what parts of a machine-checkable proof are most difficult for a programmer to read and understand” or “what built-in libraries make a proof easier to write” can help reduce costs associated with, and make more practical, formal reasoning of correctness in an industrial setting.

6.1.3 Understanding understanding: Programming logic

Cognition for programming is complex and nuanced. While our work in Chapter 5 provides the first instance of neurostimulation within a programming context, it opens many doors for further exploration. We find no evidence that traditionally believed correlations by the research community extend to a causal relationship, and see this result as an opportunity: if brain regions associated with spatial reasoning are not indeed causally associated with programming, then what contributes to programming success? Are other brain regions mentioned in software engineering literature (e.g., Broca's region) causally involved? Is it a variety of brain regions working together to help a human think like a programmer? Just as spatial training has been investigated as an intervention to help students in introductory computer science courses, answers to these questions can help pave way for cognitive interventions helping new programmers succeed in computer science (e.g., if brain regions causally linked with working memory are also responsible for programming outcomes, educators can investigate the effects of working memory training on programming abilities).

Additionally, measures like eye-tracking have been increasingly used in programming contexts to obtain a better understanding of user behavior while programming. We see the opportunity for more exploration here: it is possible to use eye-tracking to non-intrusively investigate programmer cognition for software engineering preparation and training tasks. For instance, Leetcode is an increasingly relevant resource to help future practitioners prepare for technical interviews, yet we lack a foundational understanding of what makes programmers succeed at Leetcode. Similarly, junior practitioners often undergo several weeks of training (and hence, expensive programmer time) before being able to meaningfully contribute to their development teams, yet we lack an understanding of the parts of training that are most difficult and could benefit from additional cognitive interventions. We believe that such investigations could elucidate ways to support new programmers entering the workforce.

6.1.4 Final Remarks

Given that research into cognition for computer science is relatively new, we firmly believe in the importance of open and reproducible science. We make publicly available any (de-identified) datasets, experimental protocols, and analysis scripts for work in this thesis:

- Chapter 3 replication package: https://github.com/hammad-a/verilog_repair
- Chapter 4 replication package: https://doi.org/10.5281/zenodo.7626901
- Chapter 5 replication package: https://github.com/hammad-a/ICSE24_TMS

We further provide discussions on costs and difficulties associated with adopting novel approaches from other fields to computer science research.

As a closing remark, we believe that there remain many exciting research problems in this field of work, and hope that work in this thesis paves the way for better understanding how programmers reason about computational logic.

BIBLIOGRAPHY

[1] Catherine Howley and Matt LoDolce. Gartner forecasts worldwide it spending to grow 6.8% in 2024. https://www.gartner.com/en/newsroom/press-releases/01-17-2024-gartner-forecasts-worldwide-it-spending-to-grow-six-point-eight-percent-in-2024, 2024. accessed March 2, 2024.

[2] Maricarmen Vizcaino, Matthew Buman, C Tyler DesRoches, and Christopher Wharton. Reliability of a new measure to assess modern screen time in adults. *BMC public health*, 19:1–8, 2019.

[3] Matthew A Christensen, Laura Bettencourt, Leanne Kaye, Sai T Moturu, Kaylin T Nguyen, Jeffrey E Olgin, Mark J Pletcher, and Gregory M Marcus. Direct measurements of smartphone screen-time: relationships with demographics and sleep. *PloS one*, 11(11):e0165331, 2016.

[4] Ilaria Montagni, Elie Guichard, Claire Carpenet, Christophe Tzourio, and Tobias Kurth. Screen time exposure and reporting of headaches in young adults: A cross-sectional study. *Cephalalgia*, 36(11):1020–1027, 2016.

[5] Shaun S Wang and Ulrik Franke. Enterprise it service downtime cost and risk transfer in a supply chain. *Operations Management Research*, 13(1):94–108, 2020.

[6] Joey Oostenbrink. Financial impact of downtime decrease and performance increase of it services. B.S. thesis, University of Twente, 2015.

[7] Sarah Beecham, Nathan Baddoo, Tracy Hall, Hugh Robinson, and Helen Sharp. Motivation in software engineering: A systematic literature review. *Information and software technology*, 50(9-10):860–878, 2008.

[8] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In *International Conference on Computer Aided Verification*, pages 5–10. Springer, 2008.

[9] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[10] Herb Krasner. The cost of poor software quality in the US: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pages 1–46, 2021.

[11] Robert Kowalski. *Computational logic and human thinking: how to be artificially intelligent*. Cambridge University Press, 2011.

[12] Samuel Gershman. *What makes us smart: The computational logic of human cognition*. Princeton University Press, 2021.

[13] James A Anderson. *An introduction to neural networks*. MIT press, 1995.

[14] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM/IEEE international symposium on empirical software engineering and measurement*, pages 383–392. IEEE, 2013.

[15] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25:83–110, 2017.

[16] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[17] National Academies of Sciences Engineering Medicine. *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press, 2018.

[18] David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, 1993.

[19] Kamal Uddin Sarker, Aziz Bin Deraman, and Raza Hasan. Descriptive logic for software engineering ontology: Aspect software quality control. In *2018 4th International Conference on Computer and Information Sciences (ICCOINS)*, pages 1–5. IEEE, 2018.

[20] Peter B Henderson. Mathematical reasoning in software engineering education. *Communications of the ACM*, 46(9):45–50, 2003.

[21] ABET. 2022-2023 criteria for accrediting computing programs. https://www.abet.org/wp-content/uploads/2022/03/2022-23-CAC-Criteria.pdf, 2021. Accredication Board for Engineering and Technology, accessed July 7, 2022.

[22] David Lorge Parnas. Software engineering programs are not computer science programs. *IEEE software*, 16(6):19–30, 1999.

[23] Ian Barland, Matthias Felleisen, Kathi Fisler, Phokion Kolaitis, and Moshe Y Vardi. Integrating logic into the computer science curriculum. In *Annual Joint Conference on Integrating Technology into Computer Science Education*, 2000.

[24] Chip Cutter. Amazon to retrain a third of its u.s. workforce - WSJ. https://www.wsj.com/articles/amazon-to-retrain-a-third-of-its-u-s-workforce-11562841120, Jul 2019. accessed March 18, 2024.

[25] Paul Luo Li, Amy J Ko, and Jiamin Zhu. What makes a great software engineer? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 700–710. IEEE, 2015.

[26] Shima Salehi, Sehoya Cotner, and Cissy J Ballen. Variation in incoming academic preparation: Consequences for minority and first-generation students. In *Frontiers in Education*, volume 5, page 552364. Frontiers Media SA, 2020.

[27] Willard Van Orman Quine. *Philosophy of logic*. Harvard University Press, 1986.

[28] Brian Holdsworth and Clive Woods. *Digital logic design*. Elsevier, 2002.

[29] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 2009.

[30] Brian Rush and Alan Ogborne. Program logic models: expanding their role and structure for program planning and evaluation. *Canadian Journal of Program Evaluation*, 6(2):95–106, 1991.

[31] Mary M Smyth. *Cognition in action*. Psychology Press, 1994.

[32] Michael W Eysenck and Marc Brysbaert. *Fundamentals of cognition*. Routledge, 2018.

[33] Jan T Wagner and T Nef. Cognition and driving in older persons. *Swiss medical weekly*, 141(0102):w13136–w13136, 2011.

[34] Donald Sharp, Michael Cole, Charles Lave, Herbert P Ginsburg, Ann L Brown, and Lucia A French. Education and cognitive development: The evidence from experimental research. *Monographs of the society for research in child development*, pages 1–112, 1979.

[35] Peter Carruthers. The cognitive functions of language. *Behavioral and brain sciences*, 25(6):657–674, 2002.

[36] Claudiu V Dimofte. Implicit measures of consumer cognition: A review. *Psychology & Marketing*, 27(10):921–937, 2010.

[37] Vimla L Patel, David R Kaufman, and Jose F Arocha. Emerging paradigms of cognition in medical decision-making. *Journal of biomedical informatics*, 35(1):52–75, 2002.

[38] Valentin Magnon, Guillaume T Vallet, and Catherine Auxiette. Sedentary behavior at work and cognitive functioning: a systematic review. *Frontiers in public health*, 6:401440, 2018.

[39] Joan Fuster, Toni Caparrós, and Lluis Capdevila. Evaluation of cognitive load in team sports: literature review. *PeerJ*, 9:e12045, 2021.

[40] Jessica R Cohen, Courtney L Gallen, Emily G Jacobs, Taraz G Lee, and Mark D'Esposito. Quantifying the reconfiguration of intrinsic networks during working memory. *PloS one*, 9(9):e106636, 2014.

[41] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389, 2014.

[42] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[43] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *International Conference on Software Engineering*, pages 175–186. IEEE, 2017.

[44] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *International Conference on Program Comprehension*, 2018.

[45] Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. Distilling neural representations of data structure manipulation using fMRI and fNIRS. In *International Conference on Software Engineering*, pages 396–407, 2019.

[46] Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. Neurological divide: An fMRI study of prose and code writing. In *International Conference on Software Engineering*, ICSE '20, page 678–690, 2020.

[47] Madeline Endres, Madison Fansher, Priti Shah, and Westley Weimer. To read or to rotate? Comparing the effects of technical reading training and spatial skills training on novice programming ability. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *Foundations of Software Engineering*, pages 754–766. ACM, 2021.

[48] Sheryl A. Sorby, Edmund Nevin, Avril Behan, Eileen Mageean, and Sarah Sheridan. Spatial skills as predictors of success in first-year engineering. In *IEEE Frontiers in Education Conference*, pages 1–7, 2014.

[49] Sheryl Sorby, Norma Veurink, and Scott Streiner. Does spatial skills instruction improve STEM outcomes? The answer is 'yes'. *Learning and Individual Differences*, 67:209–222, 2018.

[50] Ryan Bockmon, Stephen Cooper, William Koperski, Jonathan Gratch, Sheryl Sorby, and Mohsen Dorodchi. A CS1 spatial skills intervention and the impact on introductory programming abilities. In *Technical Symposium on Computer Science Education*, pages 766–772, 2020.

[51] Jack Parkinson and Quintin Cutts. The effect of a spatial skills training course in introductory computing. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 439–445, 2020.

[52] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. "Yours is better!" Participant response bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1321–1330, 2012.

[53] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[54] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines. In *Foundations of Software Engineering*, page 456–468, 2020.

[55] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pages 125–132, 2006.

[56] Roy D Pea and D Midian Kurland. On the cognitive prerequisite of learning computer programming (tech. rep.). *New York: Bank Street College of Education, Center of Children and Technology.(ERIC Document Reproduction Service No ED 249 931)*, 1983.

[57] Gene Alarcon and Tyler Ryan. Trustworthiness perceptions of computer code: A heuristic-systematic processing model. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[58] Denae Ford, Mahnaz Behroozi, Alexander Serebrenik, and Chris Parnin. Beyond the code itself: how programmers really look at pull requests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pages 51–60. IEEE, 2019.

[59] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ Computer Science*, 3:e111, 2017.

[60] Thad Crews and Jeff Butterfield. Gender differences in beginning programming: an empirical study on improving performance parity. *Campus-Wide Information Systems*, 20(5):186–192, 2003.

[61] Kagan Erdil, Emily Finn, Kevin Keating, Jay Meattle, Sunyoung Park, and Deborah Yoon. Software maintenance as part of the software life cycle. *Comp180: Software Engineering Project*, 1:1–49, 2003.

[62] Jussi Koskinen. Software maintenance costs. *Information Technology Research Institute, ELTIS-Project University of Jyväskylä*, page 16, 2003.

[63] Uttamjit Kaur and Gagandeep Singh. A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, 118(1):6–11, 2015.

[64] Barry Boehm and Victor R Basili. Defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[65] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

[66] Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018.

[67] Yuzhen Liu, Long Zhang, and Zhenyu Zhang. A survey of test based automatic program repair. *Journal of Software*, 13(8):437–452, 2018.

[68] Douglas Thain. *Introduction to compilers and language design*. Lulu. com, 2016.

[69] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley 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)*, pages 3–13. IEEE, 2012.

[70] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[71] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434, 2011.

[72] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[73] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[74] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[75] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[76] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java. In *Proceedings of International Symposium on Software Testing and Analysis*, 2016.

[77] AWS. What is a computer chip? Computer chips explained. https://aws.amazon.com/what-is/computer-chip/, n.d. accessed March 1, 2024.

[78] M Morris Mano and Michael Ciletti. *Digital design: with an introduction to the Verilog HDL*. Pearson, 2013.

[79] Desire Athow. Pentium fdiv: The processor bug that shook the world. https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773, Oct 2014. accesssed March 18, 2024.

[80] Jayce Wagner. Did I do that? Intel is going to make a killing fixing its own Meltdown. https://www.digitaltrends.com/computing/intel-could-make-billions-off-meltdown-spectre/, Feb 2018. accessed March 18, 2024.

[81] ISO. *International Standard (ISO/IEC 9899:2018)*. ISO/IEC, 4th edition, 2018.

[82] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[83] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[84] Chapter 6 - the case for synchronous design. In Hubert Kaeslin, editor, *Top-Down Digital VLSI Design*, pages 357–389. Morgan Kaufmann, Boston, 2015.

[85] Nienke van Atteveldt, Marlieke TR van Kesteren, Barbara Braams, and Lydia Krabbendam. Neuroimaging of learning and development: improving ecological validity. *Frontline Learning Research*, 6(3):186, 2018.

[86] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. *Eye tracking: A comprehensive guide to methods and measures*. oup Oxford, 2011.

[87] Tommy Strandvall. Eye tracking in human-computer interaction and usability research. In *IFIP Conference on Human-Computer Interaction*, pages 936–937. Springer, 2009.

[88] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys*, 51(1):1–58, 2018.

[89] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, 67:79–107, 2015.

[90] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. Eye-tracking metrics in software engineering. In *Asia-Pacific Software Engineering Conference*, pages 96–103, 2015.

[91] Dario Babić, Helena Dijanić, Lea Jakob, Darko Babić, and Eduardo Garcia-Garzon. Driver eye movements in relation to unfamiliar traffic signs: an eye tracking study. *Applied ergonomics*, 89:103–191, 2020.

[92] Michel Wedel and Rik Pieters. A review of eye-tracking research in marketing. *Review of marketing research*, pages 123–147, 2017.

[93] Lawrence Z Cai, John AM Paro, Gordon K Lee, and Rahim S Nazerali. Where do we look? Assessing gaze patterns in breast reconstructive surgery with eye-tracking technology. *Plastic and Reconstructive Surgery*, 141(3):331e–340e, 2018.

[94] Joseph H Goldberg and Jonathan I Helfman. Comparing information graphics: a critical look at eye tracking. In *BEyond time and errors: novel evaLuation methods for Information Visualization*, pages 71–78, 2010.

[95] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering*, 25(5):3128–3174, 2020.

[96] Joseph H Goldberg and Xerxes P Kotval. Computer interface evaluation using eye movements: methods and constructs. *Journal of industrial ergonomics*, 24(6):631–645, 1999.

[97] Tobii Pro AB. Tobii Pro Lab. Computer software, 2014.

[98] Marcel A Just and Patricia A Carpenter. A theory of reading: from eye fixations to comprehension. *Psychological review*, 87(4):329, 1980.

[99] Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[100] Michael A Eskenazi and Jocelyn R Folk. Regressions during reading: The cost depends on the cause. *Psychonomic bulletin & review*, 24(4):1211–1216, 2017.

[101] Siyuan Chen and Julien Epps. Using task-induced pupil diameter and blink rate to infer cognitive load. *Human–Computer Interaction*, 29(4):390–413, 2014.

[102] Jan-Louis Kruger, Esté Hefer, and Gordon Matthew. Measuring the impact of subtitles on cognitive load: Eye tracking and dynamic audiovisual texts. In *Eye Tracking South Africa*, pages 62–66, 2013.

[103] Peter Kiefer, Ioannis Giannopoulos, Andrew Duchowski, and Martin Raubal. Measuring cognitive load for map tasks through pupil diameter. In *Geographic Information Science*, pages 323–337, 2016.

[104] Eckhard H Hess and James M Polt. Pupil size in relation to mental activity during simple problem-solving. *Science*, 143(3611):1190–1192, 1964.

[105] Madeline Endres, Zachary Karas, Xiaosu Hu, Ioulia Kovelman, and Westley Weimer. Relating reading, visualization, and coding for new programmers: A neuroimaging study. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 600–612. IEEE, 2021.

[106] Yasuo Terao and Yoshikazu Ugawa. Basic mechanisms of TMS. *Journal of clinical neurophysiology*, 19(4):322–343, 2002.

[107] A Irem Sonmez, Deniz Doruk Camsari, Aiswarya L Nandakumar, Jennifer L Vande Voort, Simon Kung, Charles P Lewis, and Paul E Croarkin. Accelerated TMS for depression: a systematic review and meta-analysis. *Psychiatry research*, 273:770–781, 2019.

[108] Limor Dinur-Klein, Pinhas Dannon, Aviad Hadar, Oded Rosenberg, Yiftach Roth, Moshe Kotler, and Abraham Zangen. Smoking cessation induced by deep repetitive transcranial magnetic stimulation of the prefrontal and insular cortices: a prospective, randomized controlled trial. *Biological psychiatry*, 76(9):742–749, 2014.

[109] Alisson Paulino Trevizol, Pedro Shiozawa, Ian A Cook, Isa Albuquerque Sato, Caio Barbosa Kaku, Fernanda BS Guimarães, Perminder Sachdev, Sujit Sarkhel, and Quirino Cordeiro. Transcranial magnetic stimulation for obsessive-compulsive disorder: an updated systematic review and meta-analysis. *The journal of electroconvulsive therapy (ECT)*, 32(4):262–266, 2016.

[110] Hartwig R Siebner, Gesa Hartwigsen, Tanja Kassuba, and John C Rothwell. How does transcranial magnetic stimulation modify neuronal activity in the brain? Implications for studies of cognition. *cortex*, 45(9):1035–1042, 2009.

[111] Alexander Rotenberg, Jared Cooney Horvath, and Alvaro Pascual-Leone. *The transcranial magnetic stimulation (TMS) device and foundational techniques.* Springer, 2014.

[112] Wanalee Klomjai, Rose Katz, and Alexandra Lackmy-Vallée. Basic principles of transcranial magnetic stimulation (TMS) and repetitive TMS (rTMS). *Annals of physical and rehabilitation medicine*, 58(4):208–213, 2015.

[113] Ying-Zu Huang, Mark J. Edwards, Elisabeth Rounis, Kailash P. Bhatia, and John C. Rothwell. Theta burst stimulation of the human motor cortex. *Neuron*, 45(2):201–206, 2005.

[114] J.C Rothwell. Techniques and mechanisms of action of transcranial stimulation of the human motor cortex. *J. Neuroscience Methods*, 74(2):113–122, 1997.

[115] Saxby Pridmore, J. Americo Fernandes Filho, Ziad Nahas, Chris Liberatos, and Mark S. George. Motor threshold in transcranial magnetic stimulation: A comparison of a neurophysiological method and a visualization of movement method. *The Journal of ECT*, 14(1), 1998.

[116] Frank Schirrmeister, Michael McNamara, Larry Melling, and Neeti Bhatnagar. Debugging at the hardware/software interface, June 2012.

[117] Tai-Ying Jiang, C-NJ Liu, and Jing Ya Jou. Estimating likelihood of correctness for error candidates to assist debugging faulty HDL designs. In *2005 IEEE International Symposium on Circuits and Systems*, pages 5682–5685. IEEE, 2005.

[118] Jiann-Chyi Ran, Yi-Yuan Chang, and Chia-Hung Lin. An efficient mechanism for debugging RTL description. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.*, pages 370–373. IEEE, 2003.

[119] Roderick Bloem and Franz Wotawa. Verification and fault localization for VHDL programs. *Journal of the Telematics Engineering Society (TIV)*, 2:30–33, 2002.

[120] Kai-hui Chang, Ilya Wagner, Valeria Bertacco, and Igor L Markov. Automatic error diagnosis and correction for RTL designs. In *2007 IEEE International High Level Design Validation and Test Workshop*, pages 65–72. IEEE, 2007.

[121] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagnosis and the rectification of design errors with PRIAM. In *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 30–33, 1989.

[122] B. Peischl and F. Wotawa. Automated source-level error localization in hardware designs. *IEEE Design & Test of Computers*, 23(1):8–19, 2006.

[123] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1):3–39, 1999.

[124] Franz Wotawa. Using multiple models for debugging VHDL designs. In *Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Engineering of Intelligent Systems*, IEA/AIE '01, page 125–134, Berlin, Heidelberg, 2001. Springer-Verlag.

[125] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1):125–143, 2002.

[126] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[127] Yu Huang, Hammad Ahmad, Stephanie Forrest, and Westley Weimer. Applying automated program repair to dataflow programming languages. In Justyna Petke, Bobby R. Bruce, Yu Huang, Aymeric Blot, Westley Weimer, and W. B. Langdon, editors, *GI @ ICSE 2021*, internet, 30 May 2021. IEEE.

[128] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[129] Robert Keim. What is a Hardware Description Language (HDL)? https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/, 2020. accessed January 11, 2021.

[130] Hammad Ahmad, Yu Huang, and Westley Weimer. Cirfix: Automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 990–1003, New York, NY, USA, 2022. Association for Computing Machinery.

[131] Priscila Santiesteban, Yu Huang, Westley Weimer, and Hammad Ahmad. Cirfix: Automated hardware repair and its real-world applications. *IEEE Transactions on Software Engineering*, 2023.

[132] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[133] Andreas Zeller. Automated debugging: Are we close? *Computer*, (11):26–31, 2001.

[134] Westley Weimer. Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, page 181–190, New York, NY, USA, 2006. Association for Computing Machinery.

[135] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? A unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 75–87, New York, NY, USA, 2020. Association for Computing Machinery.

[136] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21(3):421–443, 2013.

[137] Sangeetha Sudakrishnan, Janaki Madhavan, E James Whitehead Jr, and Jose Renau. Understanding bug fix patterns in Verilog. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 39–42, 2008.

[138] James A Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[139] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[140] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[141] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *ICGA*, volume 95, pages 184–192, 1995.

[142] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[143] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019.

[144] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 456–467. IEEE, 2019.

[145] Stuart Sutherland. *RTL Modeling with SystemVerilog for Simulation and Synthesis Using SystemVerilog for ASIC and FPGA Design.* Sutherland HDL, Incorporated, 2017.

[146] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[147] Riccardo Poli and William B Langdon. Genetic programming with one-point crossover. In *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer, 1998.

[148] Brad L Miller and David E Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary computation*, 4(2):113–131, 1996.

[149] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 959–966, 2012.

[150] Christopher S Timperley. *Advanced techniques for search-based program repair.* PhD thesis, University of York, 2017.

[151] JA Vasconcelos, Jaime Arturo Ramirez, RHC Takahashi, and RR Saldanha. Improvements in genetic algorithms. *IEEE Transactions on magnetics*, 37(5):3414–3417, 2001.

[152] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, 2009.

[153] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[154] IEEE. IEEE standard for Verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[155] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. A multi-study investigation into dead code. *IEEE Transactions on Software Engineering*, 2018.

[156] Shinya Takamaeda-Yamazaki. Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

[157] VCS Synopsys. Verilog simulator. *Available at http://www.synopsys.com/products/simulation/simulation.html*, 2004.

[158] Synopsys. VCS functional verification solution, 2020.

[159] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, 2014.

[160] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[161] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[162] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. Finding and fixing faults. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 35–49. Springer, 2005.

[163] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1689–1696, 2009.

[164] Robert Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEE Proceedings-Software*, 145(6):228–236, 1998.

[165] L. I. Manolache and Derrick G. Kourie. Software testing using model programs. *Software: Practice and Experience*, 31(13):1211–1236, 2001.

[166] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774–788, 2011.

[167] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–6, 2004.

[168] Farshad Gholami, Niousha Attar, Hassan Haghighi, Mojtaba Vahidi Asl, Meysam Valueian, and Saina Mohamadyari. A classifier-based test oracle for embedded software. In *2018 Real-Time and Embedded Systems and Technologies (RTEST)*, pages 104–111, 2018.

[169] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[170] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. Supporting oracle construction via static analysis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189, 2016.

[171] Mohamed Hanafy, Hazem Said, and Ayman M. Wahba. New methodology for digital design properties extraction from simulation traces. In *2015 Tenth International Conference on Computer Engineering Systems (ICCES)*, pages 91–98, 2015.

[172] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. A comparative study on automated software test oracle methods. In *2009 Fourth International Conference on Software Engineering Advances*, pages 140–145, 2009.

[173] Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[174] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 182–191, 2010.

[175] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, 2013.

[176] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 247–258, 2016.

[177] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[178] Charles L Seitz, C Mead, and L Conway. System timing. *Introduction to VLSI systems*, pages 218–262, 1980.

[179] 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, 2012.

[180] Nada Alsolami, Qasem Obeidat, and Mamdouh Alenezi. Empirical analysis of object-oriented software test suite evolution. *International Journal of Advanced Computer Science and Applications*, 10(11), 2019.

[181] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[182] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[183] Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer. Eyes on code: A study on developers code navigation strategies. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[184] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13, 2020.

[185] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 87–102. ACM, 2009.

[186] Vaibbhav Taraate. *Digital logic design using Verilog: coding and RTL synthesis*. Springer, 2016.

[187] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. On the introduction of automatic program repair in Bloomberg. *IEEE Software*, 38(4):43–51, 2021.

[188] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. Evaluating the strategies of statement selection in automated program repair. In *International Conference on Software Analysis, Testing, and Evolution*, pages 33–48. Springer, 2018.

[189] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. An empirical study on the effect of dynamic slicing on automated program repair efficiency. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 554–558. IEEE, 2018.

[190] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F Bissyandé. Where were the repair ingredients for Defects4J bugs? *Empirical Software Engineering*, 26(6):1–33, 2021.

[191] Deheng Yang, Yuhua Qi, Xiaoguang Mao, and Yan Lei. Evaluating the usage of fault localization in automated program repair: an empirical study. *Frontiers of Computer Science*, 15(1):1–15, 2021.

[192] Deheng Yang, Yan Lei, Xiaoguang Mao, David Lo, Huan Xie, and Meng Yan. Is the ground truth really accurate? Dataset purification for automated program repair. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 96–107. IEEE, 2021.

[193] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Beslon, David M. Bryson, Nick Cheney, Patryk Chrabaszcz, Antoine Cully, Stephane Doncieux, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Fundefinedenoy, Christian Gagńe, Leni Le Goff, Laura M. Grabowski, Babak Hodjat, Frank Hutter, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David Parsons, Robert T. Pennock, William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Schulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Richard Watson, Westley Weimer, and Jason Yosinski. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artif. Life*, 26(2):274–306, may 2020.

[194] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot? Insights from the Repairnator Project. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 95–104, 2018.

[195] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

[196] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[197] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[198] Seemanta Saha et al. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2019.

[199] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[200] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113, 2019.

[201] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426, 2017.

[202] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[203] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 13–24. IEEE Press, 2019.

[204] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[205] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. Trigger-action-circuits: Leveraging generative design to enable novices to design and build circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 331–342, New York, NY, USA, 2017. Association for Computing Machinery.

[206] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47. Springer, 2011.

[207] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[208] Vinu George and Rayford Vaughn. Application of lightweight formal methods in requirement engineering. *STSC CrossTalk–The Journal of Defense Software Engineering*, 2003.

[209] Egon Börger and Robert Stärk. ASM design and analysis method. In *Abstract State Machines*, pages 13–86. Springer, 2003.

[210] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[211] Cliff B Jones. Systematic software development using VDM. *Prentice Hall International Series in Computer Science*, 1990.

[212] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[213] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[214] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[215] T Hoare. Assert early and assert often: Practical hints on effective asserting. *Presentation at Microsoft Techfest*, 2002.

[216] Constance Heitmeyer. On the need for practical formal methods. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 18–26. Springer, 1998.

[217] Mario Gleirscher, Simon Foster, and Jim Woodcock. New opportunities for integrated formal methods. *ACM Computing Surveys*, 52(6):1–36, 2019.

[218] Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, and Toshihiro Nakayama. Teaching formal methods in the context of software engineering. *ACM SIGCSE Bulletin*, 41(2):17–23, 2009.

[219] John Kloosterman and Dwight Fontenot. *Comprehensive Studies Program (CSP) Support Planning Discussions AY 2019-2020.* 2020. University of Michigan meeting held on 08/04/2020.

[220] Valeria Bertacco and Amir Kamil. *Computing CARES Survey AY 2019-2020.* University of Michigan, 2020.

[221] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering*, pages 761–768, 2006.

[222] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C Hofmeister, and André Brechmann. Simultaneous measurement of program comprehension with fMRI and eye tracking: A case study. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[223] Julia Mock, S Huber, E Klein, and K Moeller. Insights into numerical cognition: Considering eye-fixations in number processing and arithmetic. *Psychological Research*, 80(3):334–359, 2016.

[224] Kenneth H. Rosen. *Discrete mathematics and its applications.* McGraw-Hill, 7th edition, 2012.

[225] Cynthia A Stanich, Michael A Pelch, Elli J Theobald, and Scott Freeman. A new approach to supplementary instruction narrows achievement and affect gaps for underrepresented minorities, first-generation students, and women. *Chemistry Education Research and Practice*, 19(3):846–866, 2018.

[226] Patricia A Tolley, Catherine M Blat, Christopher R McDaniel, Donald B Blackmon, and David C Royster. Enhancing the mathematics skills of students enrolled in introductory engineering courses: Eliminating the gap in incoming academic preparation. *Journal of STEM Education: Innovations and Research*, 13(3), 2012.

[227] John R Reisel, Marissa Jablonski, Hossein Hosseini, and Ethan Munson. Assessment of factors impacting success for incoming college engineering students in a summer bridge program. *Intl. J. of Mathematical Education in Sci. and Tech.*, 43(4):421–433, 2012.

[228] Hammad Ahmad, Zachary Karas, Kimberly Diaz, Amir Kamil, Jean-Baptiste Jeannin, and Westley Weimer. How do we read formal claims? Eye-tracking and the cognition of proofs about algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 208–220. IEEE, 2023.

[229] Nathan R Kuncel, Marcus Credé, and Lisa L Thomas. The validity of self-reported grade point averages, class ranks, and test scores: A meta-analysis and review of the literature. *Review of educational research*, 75(1):63–82, 2005.

[230] Ketrina Yim, Daniel D Garcia, and Sally Ahn. Computer science illustrated: Engaging visual aids for computer science education. In *Proceedings of the 41st ACM technical symposium on computer science education*, pages 465–469, 2010.

[231] Jenessa R Shapiro and Steven L Neuberg. From stereotype threat to stereotype threats: Implications of a multi-threat framework for causes, moderators, mediators, consequences, and interventions. *Personality and Social Psychology Review*, 11(2):107–130, 2007.

[232] Steven J Spencer, Claude M Steele, and Diane M Quinn. Stereotype threat and women's math performance. *Journal of experimental social psychology*, 35(1):4–28, 1999.

[233] Anneli Olsen. The Tobii I-VT fixation filter. *Tobii Technology*, 21:4–19, 2012.

[234] Dario D Salvucci and Joseph H Goldberg. Identifying fixations and saccades in eye-tracking protocols. In *Eye tracking research & applications*, pages 71–78, 2000.

[235] Robert JK Jacob and Keith S Karn. Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. *Mind*, 2(3):4, 2003.

[236] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *International Conference on Program Comprehension*, pages 255–265, 2015.

[237] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *J. of the Royal statistical society: series B*, 57(1):289–300, 1995.

[238] Madeline Endres, Westley Weimer, and Amir Kamil. An analysis of iterative and recursive problem performance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 321–327, 2021.

[239] Irene Polycarpou. Computer science students' difficulties with proofs by induction: an exploratory study. In *Proceedings of the 44th annual Southeast regional conference*, pages 601–606, 2006.

[240] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[241] Johann M Schumann. *Automated theorem proving in software engineering*. Springer Science & Business Media, 2001.

[242] Kelum AA Gamage, Roshan GGR Pradeep, and Erandika K de Silva. Rethinking assessment: The future of examinations in higher education. *Sustainability*, 14(6):3552, 2022.

[243] Rudolf Netzel, Bettina Ohlhausen, Kuno Kurzhals, Robin Woods, Michael Burch, and Daniel Weiskopf. User performance and reading strategies for metro maps: An eye tracking study. *Spatial Cognition & Computation*, 17(1-2):39–64, 2017.

[244] Mary Hegarty, Richard E Mayer, and Carolyn E Green. Comprehension of arithmetic word problems: Evidence from students' eye fixations. *Journal of educational psychology*, 84(1):76, 1992.

[245] Ana Susac, Andreja Bubic, Maja Planinic, Marko Movre, and Marijan Palmovic. Role of diagrams in problem solving: An evaluation of eye-tracking parameters as a measure of visual attention. *Physical Review Physics Education Research*, 15(1):013101, 2019.

[246] Manju Manoharan and Berinderjeet Kaur. Mathematics teachers' perceptions of diagrams. *International Journal of Science and Mathematics Education*, pages 1–23, 2022.

[247] Charles Buckley and Chrissi Nerantzi. Effective use of visual representation in research and teaching within higher education. *International Journal of Management and Applied Research*, 7(3):196–214, 2020.

[248] Daesub Yoon and N Hari Narayanan. Mental imagery in problem solving: An eye tracking study. In *Proceedings of the 2004 symposium on Eye tracking research & applications*, pages 77–84, 2004.

[249] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *International Conference on Software Engineering*, pages 524–536, 2021.

[250] J. Duraes, H. Madeira, J. Castelhano, C. Duarte, and M. Castelo Branco. Wap: Understanding the brain at software debugging. In *International Symposium on Software Reliability Engineering*, pages 87–92, 2016.

[251] Joao Castelhano, Isabel C Duarte, Carlos Ferreira, Joao Duraes, Henrique Madeira, and Miguel Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fMRI study. *Brain imaging and behavior*, 13:623–637, 2019.

[252] Alberto Abadie. Causal inference. In Kimberly Kempf-Leonard, editor, *Encyclopedia of Social Measurement*, pages 259–266. 2005.

[253] Justyna Hobot, Michał Klincewicz, Kristian Sandberg, and Michał Wierzchoń. Causal inferences in repetitive transcranial magnetic stimulation research: challenges and perspectives. *Frontiers in Human Neuroscience*, 14:586448, 2021.

[254] Juha Silvanto and Zaira Cattaneo. Common framework for "virtual lesion" and state-dependent TMS: the facilitatory/suppressive range model of online TMS effects on behavior. *Brain and cognition*, 119:32–38, 2017.

[255] Tomáš Paus. Inferring causality in brain images: a perturbation approach. *Philosophical Trans. Royal Society B: Biological Sciences*, 360(1457):1109–1114, 2005.

[256] Alexander T Sack. Transcranial magnetic stimulation, causal structure–function mapping and networks of functional relevance. *Current opinion in neurobiology*, 16(5):593–599, 2006.

[257] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. "Yours is Better!": Participant response bias in HCI. In *Conference on Human Factors in Computing Systems*, page 1321–1330, 2012.

[258] Hammad Ahmad, Madeline Endres, Kaia Newman, Priscila Santiesteban, Emma Shedden, and Westley Weimer. Causal relationships and programming: A transcranial magnetic stimulation experiment. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 2024.

[259] Simone Rossi, Mark Hallett, Paolo M. Rossini, and Alvaro Pascual-Leone. Safety, ethical considerations, and application guidelines for the use of transcranial magnetic stimulation in clinical practice and research. *Clinical Neurophysiology*, 120(12):2008–2039, 2009.

[260] Eric M Wassermann. Risk and safety of repetitive transcranial magnetic stimulation: report and suggested guidelines from the international workshop on the safety of repetitive transcranial magnetic stimulation, june 5–7, 1996. *Electroencephalography and Clinical Neurophysiology*, 108(1):1–16, 1998.

[261] Aaron Alexander-Bloch, Jay N Giedd, and Ed Bullmore. Imaging structural covariance between human brain regions. *Nature Reviews Neuroscience*, 14(5):322–336, 2013.

[262] Jonathan Peirce, Jeremy R Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, Erik Kastman, and Jonas Kristoffer Lindeløv. Psychopy2: Experiments in behavior made easy. *Behavior research methods*, 51:195–203, 2019.

[263] Michael Peters and Christian Battista. Applications of mental rotation figures of the Shepard and Metzler type and description of a mental rotation stimulus library. *Brain and cognition*, 66(3):260–264, 2008.

[264] So Y. Yoon. *Psychometric properties of the Revised Purdue Spatial Visualization Tests: Visualization of Rotations (the Revised PSVT:R)*. PhD thesis, Purdue University, 2011.

[265] A Suppa, Y-Z Huang, K Funke, MC Ridding, B Cheeran, V Di Lazzaro, U Ziemann, and JC Rothwell. Ten years of theta burst stimulation in humans: established knowledge, unknowns and prospects. *Brain stimulation*, 9(3):323–335, 2016.

[266] Lizbeth Cárdenas-Morales, Dennis A Nowak, Thomas Kammer, Robert C Wolf, and Carlos Schönfeldt-Lecuona. Mechanisms and applications of theta-burst rTMS on the human motor cortex. *Brain topography*, 22:294–306, 2010.

[267] Felix Duecker, Tom A de Graaf, Christianne Jacobs, and Alexander T Sack. Time-and task-dependent non-neural effects of real and sham TMS. *PLoS One*, 8(9):e73813, 2013.

[268] Patricia L. Lockwood, Gian Domenico Iannetti, and Patrick Haggard. Transcranial magnetic stimulation over human secondary somatosensory cortex disrupts perception of pain intensity. *Cortex*, 49(8):2201–2209, 2013.

[269] Neil G. Muggleton, Peggy Postma, Karolina Moutsopoulou, Ian Nimmo-Smith, Anthony Marcel, and Vincent Walsh. TMS over right posterior parietal cortex induces neglect in a scene-based frame of reference. *Neuropsychologia*, 44(7):1222–1229, 2006.

[270] Vincenzo Romei, Micah M. Murray, Lotfi B. Merabet, and Gregor Thut. Occipital transcranial magnetic stimulation has opposing effects on visual and auditory stimulus detection: Implications for multisensory interactions. *Journal of Neuroscience*, 27(43):11465–11472, 2007.

[271] Juha Silvanto, Zaira Cattaneo, Lorella Battelli, and Alvaro Pascual-Leone. Baseline cortical excitability determines whether TMS disrupts or facilitates behavior. *Journal of Neurophysiology*, 99(5):2725–2730, 2008.

[272] JeYoung Jung, Andreas Bungert, Richard Bowtell, and Stephen R Jackson. Vertex stimulation as a control site for transcranial magnetic stimulation: a concurrent TMS/fMRI study. *Brain stimulation*, 9(1):58–64, 2016.

[273] Henrik Foltys, Roland Sparing, Babak Boroojerdi, Timo Krings, Ingo G Meister, Felix M Mottaghy, and Rudolf Töpper. Motor control in simple bimanual movements: a transcranial magnetic stimulation and reaction time study. *Clinical Neurophysiology*, 112(2):265–274, 2001.

[274] Thomas Nyffeler, Pascal Wurtz, Tobias Pflugshaupt, Roman von Wartburg, Mathias Luthi, Christian W. Hess, and René M. Muri. One-hertz transcranial magnetic stimulation over the frontal eye field induces lasting inhibition of saccade triggering. *NeuroReport*, 17(3), 2006.

[275] Barbara Tomasino, Gereon R. Fink, Roland Sparing, Manuel Dafotakis, and Peter H. Weiss. Action verbs and the primary motor cortex: A comparative TMS study of silent reading, frequency judgments, and motor imagery. *Neuropsychologia*, 46(7):1915–1926, 2008.

[276] B Boroojerdi, H Foltys, T Krings, U Spetzger, A Thron, and R Töpper. Localization of the motor hand area using transcranial magnetic stimulation and functional magnetic resonance imaging. *Clinical neurophysiology*, 110(4):699–704, 1999.

[277] Paul Dassonville, Scott M Lewis, Xiao-Hong Zhu, Kamil Ugurbil, Seong-Gi Kim, and James Ashe. The effect of stimulus–response compatibility on cortical motor activation. *Neuroimage*, 13(1):1–14, 2001.

[278] Paul Dassonville, ScottM Lewis, Xiao-Hong Zhu, Kâmil Uğurbil, Seong-Gi Kim, and James Ashe. Effects of movement predictability on cortical motor activation. *Neuroscience research*, 32(1):65–74, 1998.

[279] TA Yousry, UD Schmid, H Alkadhi, D Schmidt, Aurelia Peraud, Andreas Buettner, and P Winkler. Localization of the motor hand area to a knob on the precentral gyrus. A new landmark. *Brain: a journal of neurology*, 120(1):141–157, 1997.

[280] C Lacadie, RK Fulbright, J Arora, R Constable, and X Papademetris. Brodmann areas defined in MNI space using a new tracing tool in BioImage suite. In *Meeting of the Organization for Human Brain Mapping*, volume 771, 2008.

[281] Cheryl M Lacadie, Robert K Fulbright, Nallakkandi Rajeevan, R Todd Constable, and Xenophon Papademetris. More accurate Talairach coordinates for neuroimaging using non-linear registration. *Neuroimage*, 42(2):717–725, 2008.

[282] Ignacio Obeso, Noemí Robles, Elena M Marrón, and Diego Redolar-Ripoll. Dissociating the role of the pre-SMA in response inhibition and switching: a combined online and offline TMS approach. *Frontiers in human neuroscience*, 7:150, 2013.

[283] Dominik Pizem, Lubomira Novakova, Martin Gajdos, and Irena Rektorova. Is the vertex a good control stimulation site? Theta burst stimulation in healthy controls. *Journal of Neural Transmission*, 129(3):319–329, 2022.

[284] Magdalena W. Sliwinska, Sylvia Vitello, and Joseph T. Devlin. Transcranial magnetic stimulation for investigating causal brain-behavioral relationships and their time course. *JoVE*, (89):e51735, Jul 2014.

[285] Joseph E. Gonzales and Corbin A. Cunninghham. The promise of pre registration in psychological research. *American Psychological Association*, 2015.

[286] Yuki Yamada. How to crack pre-registration: Toward transparent and open science. *Frontiers in Psychology*, 9(1831), 2018.

[287] Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Mining software repositories. *IEEE*, 2023.

[288] Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. Pre-registration: Why and how. *J. Society for Consumer Psychology*, December 2020.

[289] Craig M. Bennett, Abigail A. Baird, Michael B. Miller, and George L. Wolford. Neural correlates of interspecies perspective taking in the post-mortem atlantic salmon: An argument for multiple comparisons correction. *Neuroimage*, 47(Suppl 1), 2009.

[290] Anders Eklund, Thomas E. Nichols, and Hans Knutsson. Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences*, 113(28):7900–7905, 2016.

[291] Xinming An and Yiu-Fai Yung. Item response theory: What it is and how you can use the IRT procedure to apply it. *SAS Institute Inc. SAS364-2014*, 10(4):1–14, 2014.

[292] Benjamin Xie, Matthew J Davidson, Min Li, and Amy J Ko. An item response theory evaluation of a language-independent CS1 knowledge assessment. In *Technical Symposium on Computer Science Education*, pages 699–705, 2019.

[293] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. Fitting linear mixed-effects models using lme4. *arXiv preprint arXiv:1406.5823*, 2014.

[294] Julian J Faraway. *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. CRC press, 2016.

[295] Joshua B Gilbert, James S Kim, and Luke W Miratrix. Modeling item-level heterogeneous treatment effects with the explanatory item response model: Leveraging large-scale online assessments to pinpoint the impact of educational interventions. *J. Educational and Behavioral Statistics*, 2023.

[296] Gerhard H Fischer and Ivo W Molenaar. Rasch models: Foundations, recent developments, and applications. 2012.

[297] Sandra Monteiro, Gail M Sullivan, and Teresa M Chan. Generalizability theory made simple (r): an introductory primer to G-studies. *J. Graduate Medical Education*, 11(4):365–370, 2019.

[298] Matthew J. Burke, Peter J. Fried, and Alvaro Pascual-Leone. Chapter 5 - transcranial magnetic stimulation: Neurophysiological and clinical applications. In Mark D'Esposito and Jordan H. Grafman, editors, *The Frontal Lobes*, volume 163 of *Handbook of Clinical Neurology*, pages 73–92, 2019.

[299] Isabel Wagner. Gender and performance in computer science. *ACM Trans. Comput. Educ.*, 16(3), May 2016.

[300] Elisabeth Hertenstein, Elena Waibel, Lukas Frase, Dieter Riemann, Bernd Feige, Michael A. Nitsche, Christoph P. Kaller, and Christoph Nissen. Modulation of creativity by transcranial direct current stimulation. *Brain Stimulation*, 12(5):1213–1221, 2019.

[301] Michal Klichowski and Gregory Kroliczak. Mental shopping calculations: A transcranial magnetic stimulation study. *Frontiers in Psychology*, 11, 2020.

[302] Alexandra G. Poydasheva, Ilya S. Bakulin, Dmitry Yu. Lagoda, Alexei A. Medyntsev, Dmitry O. Sinitsyn, Petr N. Kopnin, Liudmila A. Legostaeva, Natalia A. Suponeva, and M. A. Piradov. Effects of online repetitive transcranial magnetic stimulation on the frequency of insights during anagram solving. In *Advances in Cognitive Research, Artificial Intelligence and Neuroinformatics*, pages 107–113, 2021.

[303] A. Ciricugno, R. J. Slaby, M. Benedek, and Z. Cattaneo. *The Contribution of Non-invasive Brain Stimulation to the Study of the Neural Bases of Creativity and Aesthetic Experience*, pages 163–196. 2023.

[304] Fabiana Ruggiero, Andrea Lavazza, Maurizio Vergari, Alberto Priori, and Roberta Ferrucci. Transcranial direct current stimulation of the left temporal lobe modulates insight. *Creativity Research Journal*, 30(2):143–151, 2018.

[305] Joseph E Cavanaugh and Andrew A Neath. The Akaike information criterion: Background, derivation, properties, application, interpretation, and refinements. *Wiley Interdisciplinary Reviews: Computational Statistics*, 11(3):e1460, 2019.

[306] Hirotugu Akaike. Information theory and an extension of the maximum likelihood principle. *International Symposium on Information Theory*, pages 267–281, 1973.

[307] Anthony Christopher Davison and David Victor Hinkley. *Bootstrap methods and their application.* Number 1. Cambridge university press, 1997.

[308] Giorgia Cona, Giuliana Marino, and Carlo Semenza. TMS of supplementary motor area (SMA) facilitates mental rotation performance: evidence for sequence processing in SMA. *Neuroimage*, 146:770–777, 2017.

[309] Zachary Karas, Andrew Jahn, Westley Weimer, and Yu Huang. Connecting the dots: rethinking the relationship between code and prose writing with functional connectivity. In *Foundations of Software Engineering*, pages 767–779, 2021.

[310] David B Richardson, Ting Ye, and Eric J Tchetgen Tchetgen. Generalized difference-in-differences. *Epidemiology*, 34(2):167–174, 2022.

[311] Claudia Szabo, Nickolas Falkner, Antti Knutas, and Mohsen Dorodchi. Understanding the effects of lecturer intervention on computer science student behaviour. In *ITiCSE conference on working group reports*, pages 105–124, 2018.

[312] Stephen Cooper, Karen Wang, Maya Israni, and Sheryl Sorby. Spatial skills training in introductory computing. In *International Computing Education Research*, pages 13–20, 2015.

[313] Jay Gill, Priyanka P. Shah-Basak, and Roy Hamilton. It's the thought that counts: Examining the task-dependent effects of transcranial direct current stimulation on executive function. *Brain Stimulation*, 8(2):253–259, 2015.

[314] Heng Wang, Mingjian Pan, Changquan Qiu, Zhichao Xue, Xiaorui Wu, and Jingtian Tang. Research on digital cognitive behavior based on transcranial direct current stimulation. In *International Conference on Signal and Image Processing*, pages 1243–1247, 2021.

[315] Wanalee Klomjai, Rose Katz, and Alexandra Lackmy-Vallée. Basic principles of transcranial magnetic stimulation (TMS) and repetitive TMS (rTMS). *Annals of physical and rehabilitation medicine*, 58(4):208–213, 2015.

[316] Michail N Giannakos, John Krogstie, and Nikos Chrisochoides. Reviewing the flipped classroom research: reflections for computer science education. In *Computer Science Education Research Conference*, pages 23–29, 2014.

[317] Maria-Blanca Ibanez, Angela Di-Serio, and Carlos Delgado-Kloos. Gamification for engaging computer science students in learning activities: A case study. *Transactions on Learning Technologies*, 7(3):291–301, 2014.

[318] Stephen Cooper, Karen Wang, Maya Israni, and Sheryl Sorby. Spatial skills training in introductory computing. In *International Computing Education Research*, pages 13–20, 2015.

[319] Henry Edison, Xiaofeng Wang, and Kieran Conboy. Comparing methods for large-scale agile software development: A systematic literature review. *Transactions on Software Engineering*, 48(8):2709–2731, 2021.

[320] Denae Ford, Margaret-Anne Storey, Thomas Zimmermann, Christian Bird, Sonia Jaffe, Chandra Maddila, Jenna L. Butler, Brian Houck, and Nachiappan Nagappan. A tale of two cities: Software developers working from home during the COVID-19 pandemic. *Trans. Softw. Eng. Methodol.*, 31(2), 2021.

[321] Norsaremah Salleh, Emilia Mendes, and John Grundy. Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review. *Transactions on Software Engineering*, 37(4):509–525, 2011.

[322] Laurie Williams and Richard L. Upchurch. In support of student pair-programming. In *Technical Symposium on Computer Science Education*, page 327–331, 2001.

[323] Andrew Begel and Nachiappan Nagappan. Pair programming: What's in it for me? In *Empirical Software Engineering and Measurement*, page 120–128, 2008.

[324] Madeline Endres, Kevin Boehnke, and Westley Weimer. Hashing it out: a survey of programmers' cannabis usage, perception, and motivation. In *International Conference on Software Engineering*, pages 1107–1119, 2022.

[325] Kaia Newman, Madeline Endres, Westley Weimer, and Brittany Johnson. From organizations to individuals: Psychoactive substance use by professional programmers. In *International Conference on Software Engineering*, pages 665–677, 2023.

[326] Alan DeSantis, Seth M Noar, and Elizabeth M Webb. Speeding through the frat house: A qualitative exploration of nonmedical ADHD stimulant use in fraternities. *Journal of Drug Education*, 40(2):157–171, 2010.

[327] Jingyuan E Chen and Gary H Glover. Functional magnetic resonance imaging methods. *Neuropsychology review*, 25:289–313, 2015.

[328] Jianjun Xu, Jiayu He, Jingyan Zhang, Deheng Yang, Jiang Wu, and Xiaoguang Mao. Validating the redundancy assumption for HDL from code clone's perspective. In *Proceedings of the 2023 International Symposium on Physical Design*, pages 247–255, 2023.

[329] Jiang Wu, Zhuo Zhang, Deheng Yang, Xiankai Meng, Jiayu He, Xiaoguang Mao, and Yan Lei. Fault localization for hardware design code with time-aware program spectrum. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 537–544. IEEE, 2022.

[330] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[331] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.