UNIVERSITY OF CALIFORNIA SAN DIEGO

Toward Gamification and Crowdsourcing of Software Verification

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Dimitar Assenov Bounov

2018

DEDICATION

To my family, and to Ginger for their love and patience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Sorin Lerner for his guidance and incredible patience, even in the face of sometimes crazy ideas. I would like to thank Prof. Ranjit Jhala for his teaching in his advanced topics course, and feedback on research.

I am grateful to my other committee members - Prof. William G. Griswold, Prof. Larry Milstein and Prof. Brian Demsky for their time and insightful questions and feedback. A special thank you to Prof. Griswold for his guidance in our gamification work.

I would like to thank my collaborators Alan Leung, Rami Gökhan Kici, Anthony DeRossi and Massimiliano Mennarini, for giving me the honor of working with them and learning from them.

A special thank you to the members of the UCSD PL group (current and graduated). Without your friendship and support the last 7 years would not have been possible. In no particular order, to Allan Leung, Don Jang, Zach Tatlock - for teaching me so much. To Valentin Robert, John Sarracino, Mario Alvarez and Nikki Vazou - thank you for being outstanding friends and officemates for so long. To the broader members of the PL group - thank you for your friendship, jokes, and patiently teaching me all I didn't know.

Thank you to the members of the UCSD-CSE community for being one of the most tightly knit and fun groups of people I've had the pleasure of being a part of - form the ChezBob crew to the Skit groups and the members of CSE TLC, you guys gave this place a spirit. Also to John "Woofs" Sarracino and Arjun "The Train Conductor" Roy - for actually getting me to stick to something and for all the woofs and puns.

I am grateful to the support of Dr. Stephanie Salo - without her insight, I don't think I would have finished this PhD.

Finally, I would like to thank my partner - Ginger Gentile, for your love, humour, support and grounding that helped me so much through the last 3 years.

Chapter 1, in part, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. "Inferring Loop

Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 2, in full, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 3, in full, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 5, in part, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 6, in part, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

VITA

2009	Sc.B, Brown University
2009-2011	Software Engineer, Sun Microsystems
2009-2018	Research Assistant, University of California, San Diego
2013	Research Intern, Microsoft Research
2016	Research Intern, Mozilla Research
2017	Research Intern, Mozilla Research
2018	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Bounov, Dimitar; DeRossi, Anthony; Menarini, Massimiliano; Griswold, William G.; Lerner, Sorin. ”*Inferring Loop Invariants through Gamification*”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018

Bounov, Dimitar; Kici, Rami Gökhan; Lerner, Sorin. ”*Protecting C++ Dynamic Dispatch Through VTable Interleaving*”, 23rd Annual Network and Distributed System Security Symposium, (NDSS), 2016

Leung, Alan; Bounov, Dimitar; Lerner, Sorin. ”*C-to-Verilog translation validation*”, 13th ACM/IEEE International Conference on Formal Methods and Models for Codesign, (MEMOCODE), 2015

ABSTRACT OF THE DISSERTATION

Toward Gamification and Crowdsourcing of Software Verification

by

Dimitar Assenov Bounov

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Sorin Lerner, Chair

Software has become intimately linked with every part of our modern life, from controlling our power grids and water ways, through managing financial transactions and medical records to mediating our personal communications and social life. This increasingly complex web of programs is developed in many different languages, by different actors with varying degrees of quality control and expertise. As a result, bugs proliferate through modern software and cause failures with high human and fiscal costs.

One promising technique proposed to ensure software quality is automated software verification. In this approach, an automated tool tries to prove the software is free from entire classes of bugs. In practice however, these verification tools are not completely ’automated’ -

they still require a significant amount of manual effort, by a verification experts in the form of explicit annotations, carefully crafted hints or template libraries. This necessary manual labor, compounded by the scarcity of verification experts, limits the scalability of these tools to larger bodies of code.

In this dissertation we argue that we can overcome these scalability limitations, by opening up parts of the software verification process to a wider audience, through the use of Gamification and Crowdsourcing. The core insight here is that many parts of the verification problem can be encoded in games, making them more widely accessible.

We provide an empirical evaluation of this idea in the form of a numerical puzzle game encoding one of the common manual verification tasks - annotating loops with invariants. Our game requires only a high-school level understanding of algebra and a love for numerical puzzles, yet our user studies show that it enables non-experts to outperform state-of-the-art verification tools. We further discuss the design and early evaluation of a second game exposing even more aspects of the verification problem to non-experts.

Chapter 1

Introduction

Software is increasingly intertwined in our daily lives: it manages sensitive information like medical records and banking information; it controls physical devices like cars, planes and power plants; and it is the gateway to the wealth of information on the web. Unfortunately, software can also be unreliable and exploitable, as shown by a number of occurrences in recent history. In 2017 a bug in the Apache Struts framework [79] enabled the Equifax Hack [37], leaking personal information of 145 million people. In 2016 a vulnerability in a ‘smart contract’ on the Ethereum block chain allowed the theft of 55M USD [103]. Barely a year later another contract vulnerability allowed for the theft of a further 32M USD [52]. In the process of fixing that vulnerability, *yet another* bug led to the accidental destruction of 300M USD worth of Ethereum tokens [44].

Cyber-physical systems have had their own share of woes. In 2003 a bug in a control room alarm system led to a blackout affecting 10 million people. In 2007 an acceleration bug in Toyota’s firmware [63], caused loss of human life. Additionally, researchers have found multiple vulnerabilities in automotive software [64]. Similar security issues have also been discovered in medical devices [75] potentially putting lives at risk. Perhaps the most famous attack in this space is the Stuxnet worm [66] which itself relied on a record 4 zero-day vulnerabilities, to damage industrial infrastructure.

Finally, to this day, bugs such as stack overflows [81] and use-after-free [5] have plagued everyday software such as web browsers, office suites and even operating system kernels, enabling successful attacks as repeatedly demonstrated at the annual Pwn2Own [113] competition.

As software takes on a more pervasive and critical role in our lives, the human and financial costs of bugs and vulnerabilities are unfortunately only likely to rise.

Both industry and the academic community have put forth a number of mitigations and potential solutions to this problem such as systematic testing [20, 82, 89, 6], dynamic mitigations [86, 101, 100] and software verification [12, 13, 61, 69, 53].

Software verification is a particularly promising approach, as it tries to *prove* properties about a program once and for all, accounting for any possible execution of the program. Furthermore, when successful, Software Verification can eliminate entire classes of bugs, such as "buffer overflows" or "integer overflows".

Over the last two decades, software verification has shown tremendous advances, such as techniques for: model checking realistic code [12, 13]; analyzing large code bases to prove properties about them [109, 114]; automating Hoare logic pre- and post-condition reasoning using satisfiability modulo theories (SMT) solvers [25, 10]; fully verifying systems like compilers [69], database management systems [74], microkernels [61], and web browsers [53].

Despite these advances, even state-of-the-art techniques are limited, and there are cases where even 'fully automated' verification fails, thereby requiring some form of human intervention for the verification to complete. For example, some verification tools require *annotations* to help the prover [10, 35]. Others need expert-written libraries of patterns [32]. Some Machine Learning based techniques need human-specified features [41]. In more extreme cases, the verification itself is a very manual process, for example when interactive theorem provers are used [61, 69].

The required manual effort can limit the size of programs that can be verified. For example, Klein et al. [61] managed to prove the full functional correctness of the seL4 kernel with respect to an executable specification. However, this effort required $\sim$20 person years of effort for less than 10K lines of code (LOC). This is approximately 3 orders of magnitude smaller than the Linux kernel today. In the compiler space LeRoy et al. took 6 person years [57] to build and verify the CompCert C compiler [69] — a foundationally verified C compiler. However, at 100000 LOC CompCert is still at least an order of magnitude smaller than its commercial cousins — LLVM at 4M LOC and GCC at 8M LOC. In terms of supported C/C++ standards, compilation targets and optimization strength [57], CompCert also has a long way to catch up.

Verification tools that do scale to large code bases on the other hand, either do not guarantee the absence of bugs [98], only reason about a limited set of properties [109, 114], or generate false positives which still require manual effort [98, 114]. For example Radar [109] is a static analysis tool that has been successfully run on the whole Linux kernel, but only detects race conditions. Xie and Aiken's Saturn project [114] provides a generic framework for path and flow-sensitive analyses over large C code bases. The project has been successfully applied to the Linux kernel, however 1) proving new types of properties require expert effort in developing new analyses, and 2) the tool still generates false positives which also require user inspection [17]. The commercial Coverity static analyzer [98] also scales to code bases on the scale of Linux and GCC with little to no manual annotations. However, Coverity does not provide soundness guarantees and also produces 20% false positives [11].

In summary, while state-of-the-art automated verification systems have made great strides in recent years, they are still not quite ”automated”. For each new artifact to be verified, or new type of property added to the spec, they require a non-trivial amount of human effort. Additionally, this effort usually requires highly-trained verification experts, of which there are few. This need for highly trained manual labor limits the scalability of automated software verification to the billions of lines of code developed and used today.

While there is a lot of work in trying to scale verification by increasing the automation (see Section 5.1 for details), in this dissertation we focus on a different approach — tackling the need for scarce expert effort, by opening the verification problem to a wider audience through the use of *crowdsourcing* and *gamification*.

Crowdsourcing has been successfully used in academia for tasks such as data set building [97], training set labeling [55, 105] and performing user studies [60]. Crowdsourcing has also been successfully used by businesses such as iStockPhoto [51] and Threadless [16]. Building upon crowdsourcing's success, Gamification uses design elements from games to make participants experience more engaging [106, 107, 108, 67], and sometimes to make a complex problem more approachable[22, 72, 33]. Perhaps the most successful scientific gamification effort to date is the FoldIt [22] project, that allows non-experts to explore the complex problem of protein folding, and even make scientific discoveries [59] in the process.

1.1 Thesis Statement

In this dissertation, we will argue that Gamification can be effectively applied to the domain of Software Verification. The central insight that inspires our and related works is that the core reasoning needed for many human tasks in Software Verification is a basic understanding of high-school level mathematics, logic and a passion for mathematical puzzles.

1.2 Contributions

We demonstrate the feasibility of this approach by providing an empirical evaluation of a game — INVGAME that allows non-experts to perform one of the most common human-centered tasks in automated software verification — synthesizing loop invariants. While prior work has been done to apply Gamification to this domain [72], we add to the literature by evaluating the humans' performance against state-of-the-art automated loop inference and verification tools and demonstrate that non-experts can outperform the state-of-the-art. Furthermore, we explore a point in the game design space that exposes as much of the underlying mathematical symbols as possible. We further discuss the design and some early evaluation of a second game — FLOWGAME , that opens up the entire verification process to non-experts. In summary this work makes the following contributions:

1. Design and evaluation of a mathematical puzzle game INVGAME aimed at inferring loop

invariants.

2. Empirical evidence suggesting non-experts can outperform state-of-the art tools at the task of loop invariant inference.

3. Design and early experience with a more complex follow-up game FLOWGAME , aimed at exposing more of the verification task to non-experts

1.3 Outline of dissertation

The rest of this dissertation is organized as follows. In Chapter 2 we present some necessary background on Software Verification. In Chapter 3 we present the design and gameplay of INVGAME , describe how its results are used to build correct loop invariants, and provide an empirical evaluation against state-of-the-art automated approaches. In Chapter 4 we discuss the design of our second game — FLOWGAME , discuss some early experiments and the challenges this game faces. In Chapter 5 we discuss related work on loop invariant inference and gamification of software verification. We discuss directions for future work in Chapter 6 we and finally in Chapter 7 we summarize and conclude the dissertation.

1.4 Acknowledgments

This chapter is in part adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 2

Background

As background, we first present some standard material on program verification and loop invariants. Consider the code in Figure 2.1, which is a benchmark taken from a Software Verification Competition [1]. The code uses C syntax, and sums up the first `n` integers. The code is straightforward in its meaning, except for the `assume` and `assert` statements, which we describe shortly. In the context of the following discussion, we will assume that a tool called a *program verifier* will try to verify this benchmark.

2.1 Specifications

The first step in program verification is defining what we want to verify about the program. This is achieved through a *program specification*. There are many mechanisms for stating specifications, and here we use a standard and widely used approach: pre- and post-conditions expressed as assume and assert statements. This approach is taken by the benchmarks in the Software Verification Competition, and also by all preeminent general verification tools.

The `assume` statement at the beginning of the code in Figure 2.1 states that the program verifier can assume that `1<=n` holds at the beginning of `foo`, without proving it. It will be the responsibility of the caller of `foo` to establish that the parameter passed to `foo` satisfies `1<=n`.

The `assert` statement at the end of the code states what we want the verifier to prove. In this case, we want the verifier to show that after the loop, the predicate `2*sum == n*(n+1)`

```
void foo(int n) {
  int sum,i;
  assume(1 <= n);
  sum = 0;
  i = 1;
  while (i <= n)
  // invariant: a condition that holds here
  // at each iteration
  {
    sum = sum + i;
    i = i + 1;
  }
  assert(2*sum == n*(n+1));
}
```

Figure 2.1. Code for our running example

holds.

Note that after the loop, the `sum` variable holds the sum of the first `n` integers. This benchmark is therefore asking the verifier to show Gauss's theorem about summing sequences of numbers, which states that the sum of the first n integers (n included) is equal to $n(n+1)/2$.

2.2 Loop Invariants

Reasoning about loops is one of the hardest parts of program verification. The challenge is that the verifier must reason about an unbounded number of loop iterations in a bounded amount of time.

One prominent technique for verifying loops is to use *loop invariants*. A loop invariant is a predicate that holds at the beginning of each iteration of a loop. For example, `i >= 1` and `sum >= 0` are loop invariants of the loop in Figure 2.1. Not all loop invariants are useful for verification, and we'll soon see that the above two invariants are not useful in our example. However, before we can figure out which loop invariants are *useful*, we must first understand how to *establish* that a predicate is a loop invariant.

It is not possible to determine that a predicate is loop invariant by simply testing the program. Tests can tell us that a predicate holds on some runs of the program, but it cannot tell us that the predicate holds on all runs. To establish that a predicate is a loop invariant, we must establish that the predicate holds at the beginning of each loop iteration, *for all runs of the program*.

To show that a predicate is a loop invariant, the standard technique involves proving some simpler properties, which together imply that the predicate is loop invariant. For example, in our code, given a predicate $\mathscr{I}$, the following two conditions guarantee that $\mathscr{I}$ is a loop invariant:

1. **[I-ENTRY] $\mathscr{I}$ holds on entry to the loop:** If the code from our example starts executing with the assumption that `1 <= n`, then $\mathscr{I}$ will hold at the beginning of the loop. This can be encoded as the verification of the straight-line code shown on row 1 of Figure 2.1.

2. **[I-PRESERVE] $\mathscr{I}$ is preserved by the loop:** If an iteration of the loop starts executing in a state where $\mathscr{I}$ holds, then $\mathscr{I}$ will also hold at the end of the iteration. This can be encoded as the verification of the straight-line code shown on row 2 of Figure 2.1. Note that in this code, the `assume` statement also encodes the fact that, since an iteration of the loop is about to run, we know that the `while` condition in the original code, `i <= n`, must also hold.

Given a candidate loop invariant, there are known techniques for automatically checking the above two conditions. Since these conditions only require reasoning about straight-line code, they are much easier to establish than properties of programs with loops. Typically, one encodes the straight-line code as some formula that gets sent to a certain kind of theorem prover called an SMT solver [25]. As a result, given a candidate predicate, there are known techniques to automatically determine if the predicate is a loop invariant on all runs of the program.

2.3 Verification Invariants

The main reason for having loop invariants is to establish properties that hold *after* the loop. However, in general a loop invariant is not guaranteed to be useful for this purpose. For

Table 2.1. Straight-line code to verify for each condition

1: [I-ENTRY]	assume(1 <= n);
	sum = 0;
	i = 1;
	assert($\mathscr{I}$);

2: [I-PRESERVE]	assume($\mathscr{I}$ && i <= n);
	sum = sum + i;
	i = i + 1;
	assert($\mathscr{I}$);

3: [I-IMPLIES]	assume($\mathscr{I}$ && i > n);
	assert(2*sum == n*(n+1));

example, `true` is a loop invariant for any loop, but it does not help in proving asserts after the loop.

To be useful for verification, the loop invariant must satisfy an additional condition:

3. **[I-IMPLIES] $\mathscr{I}$ implies the post-loop assert:** If $\mathscr{I}$ holds at the end of the last iteration of the loop, then the assert after the loop will hold. This can be encoded as the verification of the straight-line code shown in row 3 of Figure 2.1. Note that in this code, the `assume` statement encodes the fact that, since we are done running the loop, we know that the `while` condition in the original code must be false, which gives us `i > n`.

We call a loop invariant that satisfies the above property a *verification invariant*. In other words: whereas a loop invariant holds at each iteration of a loop, a verification invariant additionally implies the post-loop assert.

Verification. Say we have found a verification invariant for a given loop. In this case, the post-loop assert immediately follows from I-ENTRY, I-PRESERVE, and I-IMPLIES. Thus, finding a verification invariant is the key to proving post-loop asserts, and as such they are used by many verification tools.

Some tools require the user to explicitly provide the verification invariant. However, this requires a significant amount of effort for the programmer, and typically also requires some kind of expertise in formal verification. As a result, tools and techniques have been developed to automatically infer invariants.

As we will soon explain, automatically coming up with verification invariants is challenging. The tool must not only find a property that holds at each iteration of the loop, but is strong enough to show the post-loop assert.

Note that in our running example, the post-loop assert is *not* a loop invariant. Some readers might want to try to figure out the invariant by looking at the code before we give the invariant away.

Verification Invariant for our Example. The verification invariant for our example is:

```
2*sum == i*(i-1) && i <= n+1
```

The above invariant for our running example was not inferred by any of the five preeminent verification systems we tried. There are four main reasons why this invariant (and invariants in general) are difficult to infer automatically.

First, the expression `i-1` does not appear syntactically anywhere in the program (or in any assert or assume). In general, program verifiers must come up with invariants that are often quite removed from the actual statements, expressions, asserts and assumes in the program.

Second, the expression `2*sum == i*(i-1)` is not a common pattern. Some tools use patterns, for example $A = B * C$, to guess possible invariants. But the invariant in this case does not fit any of the common patterns encoded in preeminent verification tools. In general, program verifiers must come up with invariants that don't belong to predefined patterns.

Third, the verifier has to figure out the `i <= n+1` conjunct, which seems orthogonal to the original assert. In this case, the additional conjunct is somewhat straightforward to infer, because it comes from the loop termination condition and the single increment of `i` in the loop. However, in general, candidate loop invariants often need to be strengthened with additional conjuncts so that verifiers can establish that they are verification invariants.

Fourth, the space of possible invariants is very large. In this example, if we consider just +, *, equality and inequality, there are over 10^{14} possible predicates whose number of symbols is equal to or smaller than the final verification invariant.

Finally, its worth noting that automatic inference of verification invariants for arbitrary programs and specifications is itself an undecidable problem, as follows from Rice's theorem [85]. Due to the above four considerations, automatic inference of verification invariants even for the more restricted set of programs and specifications that people care about 'in practice', this task is still extremely difficult.

2.4 Acknowledgments

This chapter is in full adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. "Inferring Loop Invariants through Gamification", Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 3

InvGame

3.1 Overview

In this dissertation, we focus on the particular problem of *inferring loop invariants*, an essential part of program verification that is hard to fully automate. Loop invariants are properties that hold at each iteration of a loop. They are needed to verify properties of loops, and as such are required in any realistic program verification effort. Despite decades of work, inferring good loop invariants is still a challenging and active area of research, and often times invariants need to be provided manually as annotations.

In this chapter, we will show how to leverage *gamification* and *crowdsourcing* to infer correct loop invariants that leading automated tools cannot. Our approach follows the high-level idea explored in prior citizen science projects, including Foldit [22] and, more closely related to our research, VeriGames [28]: turn the program verification task into a puzzle game that players can solve online through gameplay. The game must be designed in such a way that the players can solve the tasks without having domain expertise.

In particular, we have designed a puzzle game called INVGAME . Our game displays rows of numbers, and players must find patterns in these numbers. Without players knowing it, the rows in the game correspond to values of variables at each iteration of a loop in a program. Therefore, unbeknownst to the player, finding patterns in these numbers corresponds to finding candidate loop invariants. As a result, by simply playing the game, players provide us with candidate loop invariants, without being aware of it, and without program verification expertise. INVGAME then checks these candidates with a solver to determine which are indeed loop invariants.

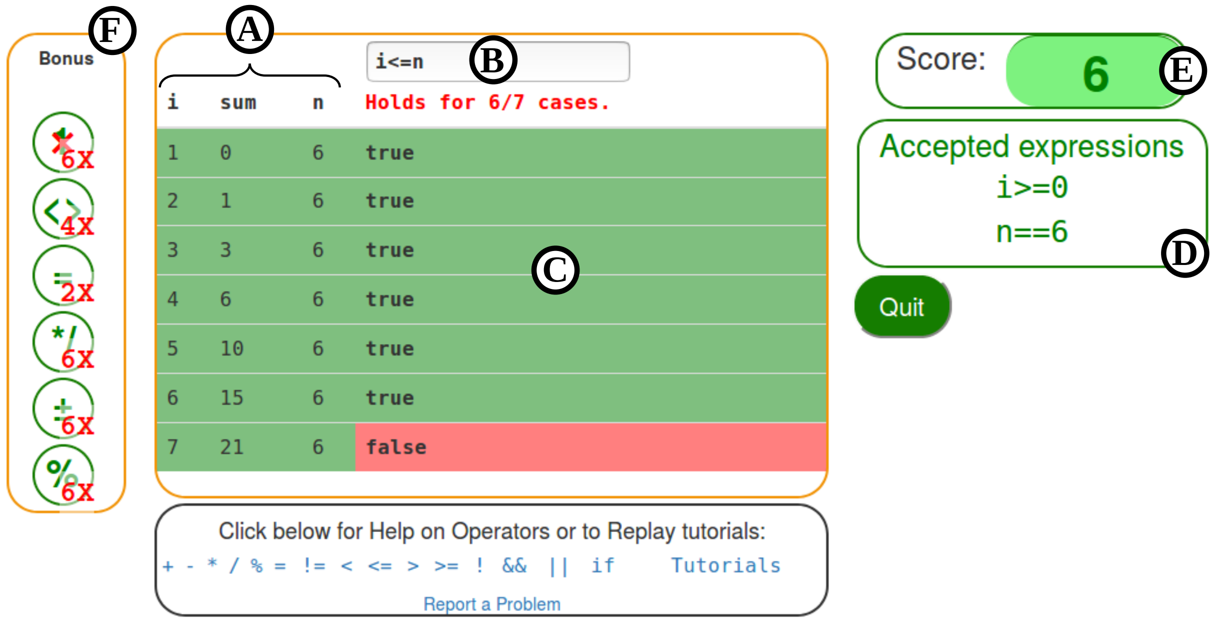
This division of labor between humans and computers is beneficial because invariants are easier to for a tool to check than to generate. We let humans do the creative work of coming up with potential invariants, and we use automated tools to do the tedious work of proving that the invariant really holds.

While there has been prior work on inferring loop invariants through crowdsourcing and gamification [72], our work distinguishes itself in two ways (for more details on prior art see Section 5). First, we have chosen a point in the design space that exposes numbers and math symbols directly to players in a minimalist game, thus leveraging human mathematical intuition. Second, we present a thorough empirical evaluation against state-of-the-art automated tools, and show how our gamification approach can solve problems that automated tools alone cannot.

We envision our gamification approach being used on the problems that automated tools cannot verify. To evaluate our game in this setting, we collected a large set of verification benchmarks from the literature and our own experiments, and ran four state-of-the-art preeminent verification tools on these benchmarks, without giving the tools any human-generated annotations. Of these benchmarks, all but 14 were solved by state-of-the-art tools. The remaining 14 benchmarks, which could *not* be verified by state-of-the-art tools, are precisely the benchmarks we tried to solve using our gamification approach. In particular, our game, played by Mechanical Turk users, was able to solve 10 of these 14.

3.2 Game Design

We now describe our game, INVGAME , by showing how it allows us to verify the running example from Figure 2.1. Figure 3.1 shows a screenshot of INVGAME on a level that corresponds to our running example.



i<=n

Holds for 6/7 cases.

i	sum	n	
1	0	6	true
2	1	6	true
3	3	6	true
4	6	6	true
5	10	6	true
6	15	6	true
7	21	6	false

Score: 6

Accepted expressions
i>=0
n==6

Click below for Help on Operators or to Replay tutorials:
+ - * / % = != < <= > >= ! && || if Tutorials
Report a Problem

Figure 3.1. INVGAME play screen

The area labeled (A) in the screenshot is a *data table*. This table shows rows of numbers, with columns labeled at the top. The player does not know this, but this table corresponds to the values of variables from a particular run of our example. In particular, the table displays the values of `i`, `sum` and `n` at the beginning of each loop iteration when the `foo` function from Figure 2.1 is called with `n = 6`.

The area labeled (B) in the screenshot is a *predicate input box*. Players enter boolean predicates in this box, with the goal of finding a predicate that evaluates to true for all rows.

The area labeled (C), which appears immediately below the input box, is the *result column*, which displays the result of evaluating the entered predicate on each row. Results in this column are displayed as “true” or “false”, with “true” displayed in green, and “false” displayed in red. For example, in the screenshot of Figure 3.1, the player has entered `i<=n`. On the first six rows, this predicate evaluates to true, whereas it evaluates to false on the last row.

The result column updates automatically as soon as the player stops entering characters in the input box. More precisely, when the player hasn’t entered a character in more than 500ms, the game tries to parse the text in the input box, and if it parses as a boolean expression, it evaluates the predicate on each row of the table, and displays the results in the result column.

In the screenshot of Figure 3.1, not all rows are green (true). Let's assume at this point the player enters the predicate `i<=n+1` (for example by simply adding "`+1`" at the end of the text box). The result column immediately updates to all green (true). At this point, the player is told that they can press "enter" to submit the predicate. If the player presses "enter" at this point (i.e., when all rows are green), four things happen.

First, the predicate in the text box is added to area Ⓓ on the screen, which contains a list of accepted expressions.

Second, the score in area Ⓔ is updated. Each predicate is worth 1 point multiplied by a set of "power-up" multipliers. Area Ⓕ displays the currently available multipliers. We discuss how the multipliers work in the next section.

Third, the multipliers in area Ⓕ are updated to new power-ups that will be in effect for the next entered predicate.

Fourth (and finally), INVGAME records the entered predicate in a back-end server as a candidate loop invariant. Note that all rows being green does not necessarily mean that the entered predicate is an invariant, let alone a verification invariant. We will discuss later what INVGAME does with these candidates invariants, but at the very least it checks if some subset of the predicates entered by the player is a verification invariant for the level. INVGAME does this by checking, on the fly, conditions I-ENTRY, I-PRESERVE and I-IMPLIES.

If the player finds a verification invariant for the current level, the player has solved the level, and can go to the next level (which is generated from a different loop to verify). Also, if the player has not solved the level after entering a certain number of invariants, the player is advanced to the next level.

Note that INVGAME does not give the player any hint of the post-loop assert, or the code in the loop. As such, players are "blind" to the final goal of the invariant; they are just looking for patterns in what they see. While there are possibilities for adding such information to the game, our current design provides a more adversarial setting for getting results. Even in this more adversarial setting, we show that players can solve levels that leading automated tools cannot.

3.2.1 Scoring

One of the biggest challenges in developing a scoring strategy is that INVGAME does not know the final solution for a level. Thus, it is hard to devise a scoring mechanism that directly incentivizes getting closer a solution.

We address this problem by instead incentivizing *diversity*. We achieve this through "power-ups" that provide bonus points for entering certain kinds of predicates. The bonus points vary over time, and are bigger for symbols that the player has not entered in a while.

In particular, each accepted predicate earns one point, multiplied by the "power-ups" that apply to the predicate. The power-ups are shown in area Ⓕ. In order from top to bottom these power-ups apply in the following situations: the first applies if the predicate does not use any constants like 1, or 0; the second applies if the predicate uses inequality; the third applies if the predicate uses equality; the fourth applies if the predicate uses multiplication or division; the fifth applies if the predicate uses addition or subtraction; the sixth applies if the predicate uses the modulo operator.

Each power-up has an associated multiplier that varies over time. The multiplier is shown next to each power-up in area Ⓕ. For example, the top-most power-up in area Ⓕ has multiplier 6X. At the beginning of a level, all multipliers are set to 2X. When a predicate is entered, the multipliers of all power-ups that apply are reset to 2X and the multipliers of all power-ups that do not apply are incremented by 2 (the sequence is: 2X, 4X, 6X, 8X, etc). As a result, the longer a power-up has not been used, the more valuable it becomes.

Power-ups compose, so that for example if two power-ups apply to a given predicate, with values 6X and 4X, then the score for that predicate is the base score of 1, multiplied by 6 and then by 4, giving 24.

Power-ups serve two purposes. First, they make the game more fun. In preliminary experiments with players, we often observed players show outward enjoyment from trying to hit multiple power-ups at once. The gratification of hitting a huge-scoring predicate was very apparent, something we also noticed when we as authors played the game. Second, power-ups incentivize players to enter a diversity of predicates. For example, if a player has not entered an inequality in a while, the inequality multiplier becomes large, incentivizing the player to enter an inequality.

3.2.2 Column Ordering

We found through preliminary trials that one important consideration is the order in which columns are displayed. In particular, certain column orderings made it more likely for players to discover certain invariants. For example, consider the screenshot in Figure 3.1, and recall the invariant in this case: `2*sum == i*(i-1) && i <= n+1`. The hardest part of this invariant is `2*sum == i*(i-1)`, which relates `sum` and `i`. This relationship is easier to see if the `sum` and `i` columns are displayed right next to each other, as in Figure 3.1. If the `sum` and `i` columns were further apart (for example by having the `n` column in between), then the pattern between `i` and `sum` would be harder to see, because the intervening columns would be cognitively distracting. The situation would be even worse if there were *two* intervening columns, containing unrelated numbers.

To address this problem, INVGAME serves the same level with different column orderings. In general, for n columns there are $n!$ possible orderings, which is quite large. To make this number more manageable, we take advantage of the fact that proximity between pairs of variables is the most important consideration. To this end, we define a notion of *adjacent-completeness*: we say that a set of orderings is *adjacent-complete* if all pairs of variables are adjacent in at least one of the orderings.

For 3 variables a,b,c, there are a total of 6 possible orderings, but only two orderings are needed for adjacent-completeness: abc and acb. For 4 variables a, b, c, d, there are a total of 24 possible orderings, but surprisingly only two orderings are needed to be adjacent-completeness: $abcd$ and $cadb$. In general, adjacent-completeness provides a much smaller number of orderings to consider, while still giving us the benefit of proximity. INVGAME uses a pre-computed table of adjacent-complete orderings, and when a level is served, the least seen ordering for that level is served.

3.2.3 Gamification Features

Our approach uses gamification, the idea of adding gaming elements to a task. INVGAME has four gaming elements: (1) a scoring mechanism, (2) power-ups that guide players to higher scores, (3) a level structure, (4) and quick-paced rewards for accomplishments (finding invariants/finishing levels). While we didn't perform a randomized controlled study to measure the effects of these gamification elements, preliminary experiments with players during the design phase showed that scoring and power-ups significantly improved the enjoyment and engagement of players.

3.3 Back-end Verification Checking

Our backend incorporates the Z3 SMT solver and has two main functions — (1) an on-line check for tautologies and trivially weaker expressions and (2) an on-line attempt to find a sound subset of the candidate invariants. The first use case is used to prevent trivial cheating, while the latter use case tries to verify a level given the current candidate invariants from the user.

3.3.1 Preventing Trivial Cheating

The default gameplay modality — accepting any predicates that satisfy the set of green rows — allows for two trivial ways of cheating: (1) entering tautologies and (2) entering an unbounded sequence of weaker expressions. In both cases there is an infinite number of expressions readily available to players, that allow them to get any score without contributing useful information.

For example, a player may enter any of the infinite sequence of tautologies of the shape `x=x`, `x+1=x+1`, `x+2=x+2`, etc. Additionally given a variable `x` for which all values are less than

1000, a player can also enter any of the infinite sequence of increasingly weaker expressions `x<1000`, `x<1001`, `x<1002`, etc.

To prevent this the game uses the backend SMT solver to check for every submitted predicate `I` whether (1) `I` is a tautology and (2) whether `I` is implied by any of the currently discovered candidate invariants by the user.

Note that due to incompleteness of the underlying decision theories Z3 uses, it is possible for users to build tautologies that Z3 cannot recognize as such. In our experiments we did not observe such behavior from players, however some experts playing the game recognized that they could do this. Furthermore, in our experiments we did not observe the tautology and implication checking to cause any significant issues with the game's responsiveness. All checks were performed with a relatively small timeout of 5-10 seconds in the backend.

3.3.2 Computing Sound Subset of Invariants

A predicate that makes all rows true (green) is not necessarily an invariant, let alone a verification invariant. For example, in Figure 3.1, `n==6` would make all rows true, but it is not an invariant of the program. In other words, predicates entered by players are based on a particular run of the program, and might not be true in general.

Thus, we need a back-end solver that checks if the entered predicate is a verification invariant. This is straightforward for a given predicate, but there is a significant challenge in how to handle multiple predicates. For example, in Figure 3.1, the player might have entered `2*sum == i*(i-1)` and `i <= n+1` individually, but also other predicates in between, one of them being `n==6`, which is *not* a loop invariant. If we simply take the conjunction of all these predicates, it will not be a loop invariant (because of `n==6`), even though the player has in fact found the key invariants. Therefore, the INVGAME back-end needs to check if the conjunction of some *subset* of the entered predicates is an invariant. Furthermore, to enable collaborative solving, the back-end would need to consider the predicates from *all* players, which can lead to hundreds of predicates. Given 100 predicates there are 2^{100} possible subsets, which is for too many to test in practice.

To address this problem, we use *predicate abstraction*, a widely investigated technique for doing program analysis [8]. In particular, given a set of predicates that are candidate loop invariants, the back-end first uses a theorem prover to prune out the ones that are not implied by the `assume` statements at the beginning of the function. For the remaining predicates, our back-end uses a fixpoint computation, iteratively reducing the set of candidate invariants. In particular, it first assumes that the remaining predicates all hold at the beginning of the loop. Then for each remaining predicate p the back-end asks a theorem prover whether p holds after the loop (under the assumption that all predicates hold at the beginning of the loop). The predicates p that pass the test are kept, and ones that don't are pruned, thus giving us a smaller set of remaining predicates. The process is repeated until the set of predicates does not change anymore, reaching a fixed point. This process issues at most $O(n^2)$ queries to the underlying solver, where n is the number of candidate invariants. This follows trivially from the fact that at every iteration at most $O(n)$ individual queries are issued, and there are no more than $O(n)$ iterations since at each iteration at least one of the n predicates is removed. Note that we can improve performance by (1) batching the $O(n)$ queries at each iteration into a single large query, (2) caching information on invariant soundness between player attempts. Interestingly enough, the theory on program analysis tells us that upon termination the remaining set will be the maximal set of predicates whose conjunction is a loop invariant, modulo solver timeouts.

We run the solver in real-time, but with a short time-out, and leave the rest to an offline process. Predicates for an individual player are done quickly, in most cases at interactive speeds. However, checking the predicates of all players collectively (to check if a conjunction of predicates from different players solves a level) must often be left to an offline process.

3.4 Evaluation

We want to evaluate InvGame along five dimensions, each leading to a research question: *(1) Comparison against Leading Tools*: Can InvGame solve benchmarks that leading automated tools cannot? *(2) Player Skill*: What skills are needed to play InvGame effectively? *(3) Solution Cost*: How much does it cost (time, money) to get a solution for a benchmark using InvGame ? *(4) Player Creativity*: How creative are players at coming up with new semantically interesting predicates? *(5) Player Enjoyment*: How fun is InvGame ? Before exploring each of these questions in a separate subsection, we first start by describing our experimental setup.

3.4.1 Experimental Setup

We envision our gamification approach being used on problems that automated tools cannot verify on their own. To evaluate InvGame in this setting, we collected a set of 243 verification benchmarks, made up of benchmarks from the literature and test cases for our system. Each benchmark is a function with preconditions, some loops and an assert after the loops. Functions range in size from 7 to 64 lines of code. Although these functions might seem small, it's important to realize that we have included benchmarks from recent prominent papers on invariant generation [30, 41], and also benchmarks from recent Software Verification Competitions [1].

We removed from this set of benchmarks those that use language features that InvGame doesn't handle, in particular doubly nested loops, arrays, the heap and disjunctive invariants (we discuss limitations of our approach in Section 6.1). This left us with 66 benchmarks. We ran four automated state-of-the-art verification tools on these benchmarks, namely: Daikon [32], CPAChecker [14], InvGen [47] and ICE-DT [41]. Daikon is a dynamic statistical invariant inference engine that generates invariants from a fixed set of templates. CPAChecker is a configurable static analysis tool based on abstract interpretation. InvGen is an automatic linear arithmetic invariant generator. Finally, ICE-DT is a state-of-the art invariant generator based on

Table 3.1. Benchmarks not solved by other tools

Name	**Invariant**	**Solved**
gauss-1	`2*sum==i*(i-1) && i<=n+1`	✓
gauss-2	`2*s==i*j && j==i-1 && i<=n+1`	✓
sqrt	`su==(a+1)*(a+1) && t==2*a+1`	✓
nl-eq-1	`i == 2*k*j`	✓
nl-eq-2	`i == k*j*l`	✓
nl-eq-3	`i == 10+k*j`	✓
nl-eq-4	`i == l+k*j`	✓
nl-ineq-1	`i*j <= k`	✓
nl-ineq-2	`i*j <= k`	✓
nl-ineq-3	`i*i <= k`	✓
nl-ineq-4	`i <= j*k`	✗
prod-bin	`z+x*y == a*b`	✗
cube-1	`i*(i+1)==2*a && c==i*i*i`	✗
cube-2	`z==6*(n+1) && y==3*n*(n+1)+1 && x==n*n*n`	✗

counter-example guided machine learning.

We consider a benchmark as *solved by leading tools* if any of these four tools solved it. This left us with 14 benchmarks that were *unsolved by leading tools*. These 14 benchmarks, shown in Table 3.1, are the ones we aimed to solve using INVGAME . In general, these benchmarks are challenging for automated tools because of the non-linear equalities and inequalities.

Here is the provenance of the 14 benchmarks that have resisted automated checking: `gauss-1` is from the Software Verification Competition suite [1]; `sqrt` is from the ICE-DT [41] benchmark suite; `gauss-2` is an elaborated version of `gauss-1`; `cube-1` is based on a clever technique for computing cubes [99]; `cube-2` and `prod-bin` are from an online repository of polynomial benchmarks [4]; the remaining 7 are from our test suite.

We use Mechanical Turk to run our experiment, as a set of human intelligence tasks (HITs). One HIT consists of playing at least two levels (benchmarks) of the game, but we let the players play more if they want. We paid participants because Mechanical Turk is a paying platform. Per our IRB protocol, we aimed to pay our players at a rate of about $10/hr. Participants were paid even if they didn't solve a level, and we didn't connect pay to score. Because we did not want users to purposely slow down to get paid more, we paid players by the level rather than by the hour. Preliminary trials showed that \$0.75/level would lead a pay rate of \$10/hr. However, players in experiments were quicker than in trials (avg of 109s/level), yielding an average pay rate of \$24.77/hr.

INVGAME dynamically serves the level (benchmark) that the current player has played the least. Although a production system would run a level until it was solved, our goal was to better understand how players interact with INVGAME . To this end, we ran experiments until we had at least 15 unique players for each level. In total, our data set includes 685 plays and 67 unique players, with each level played between 36 and 72 times. Some levels were also solved many times.

3.4.2 Comparison against Leading Tools

Table 3.1 shows the benchmark name, the verification invariant, and whether our gamification approach was able to find the invariant using Mechanical Turk users. In short, our gamification approach worked on 10 out of the 14 benchmarks.

Individual vs. Collective Solving. One interesting question is whether solutions came from single plays or collections of plays. We define a *play* as a player playing a level once. We define an *individual solving play* as a play of a level that *by itself* finds the *entire* required invariant. In all cases but two, the solutions in our experiments came from individual solving plays. However, there can be cases where there is no individual solving play (because each player only finds part of the invariant), but putting the invariants together from all plays solves the benchmark. We call such solutions *collective solutions*; these solutions can only happen when the verification invariant has a conjunct, so in 5 of our 14 benchmarks. In our experiments, collective solutions happened *only twice*, for `sqrt` and `gauss-2` (for `sqrt`, an individual solution was also found). For *collective* solutions, because of the way our solver works, it is difficult to determine who contributed to solving the level. As a result, in all of the following discussions where we need

Table 3.2. Player math and computer science skill rating scales

Math 1	Middle school math	Prog 1	No experience
Math 2	High school math	Prog 2	Beginner
Math 3	College level math	Prog 3	Intermediate
Math 4	Masters level math	Prog 4	Advanced
Math 5	PhD level math	Prog 5	Professional

to attribute the solution to a player, we only consider individual solving plays (and call them solving plays).

3.4.3 Player Skill

To better understand the relationship between a player's prior math/programming skills and their ability to solve levels, we asked players to rate themselves on a scale from 1 to 5 on math expertise and programming expertise. We use the term "math skill" and "programming skill" to refer to the math/programming skill rating that the player gave themselves. The skill ratings for math and programming shown in the survey are displayed in Table 3.2.

Bearing in mind that these are self-reported metrics, Figures 3.2(a) and (b) show the number of players at each math and programming skill, respectively. The majority of our players have not taken a math course beyond high-school or college and have either no programming experience, or are at best novice to intermediate. It's also worth noting that we have no players that only took math at the middle-school level and that we see a wider spread of skill levels in programming experience than math. Finally, we had no expert programmers (i.e. players who self-identified as having programming skill 5 in Table 3.2) in our study. Since verification is an advanced and specialized topic, we assume this also means that none of our players had verification expertise.

We define the math/programming skill of a *play* as the math/programming skill of the player who played the play. Figures 3.2(c) and (d) show the number of solving plays (recall, these are individual solving plays) categorized by the skill of the solving play — math in (c)

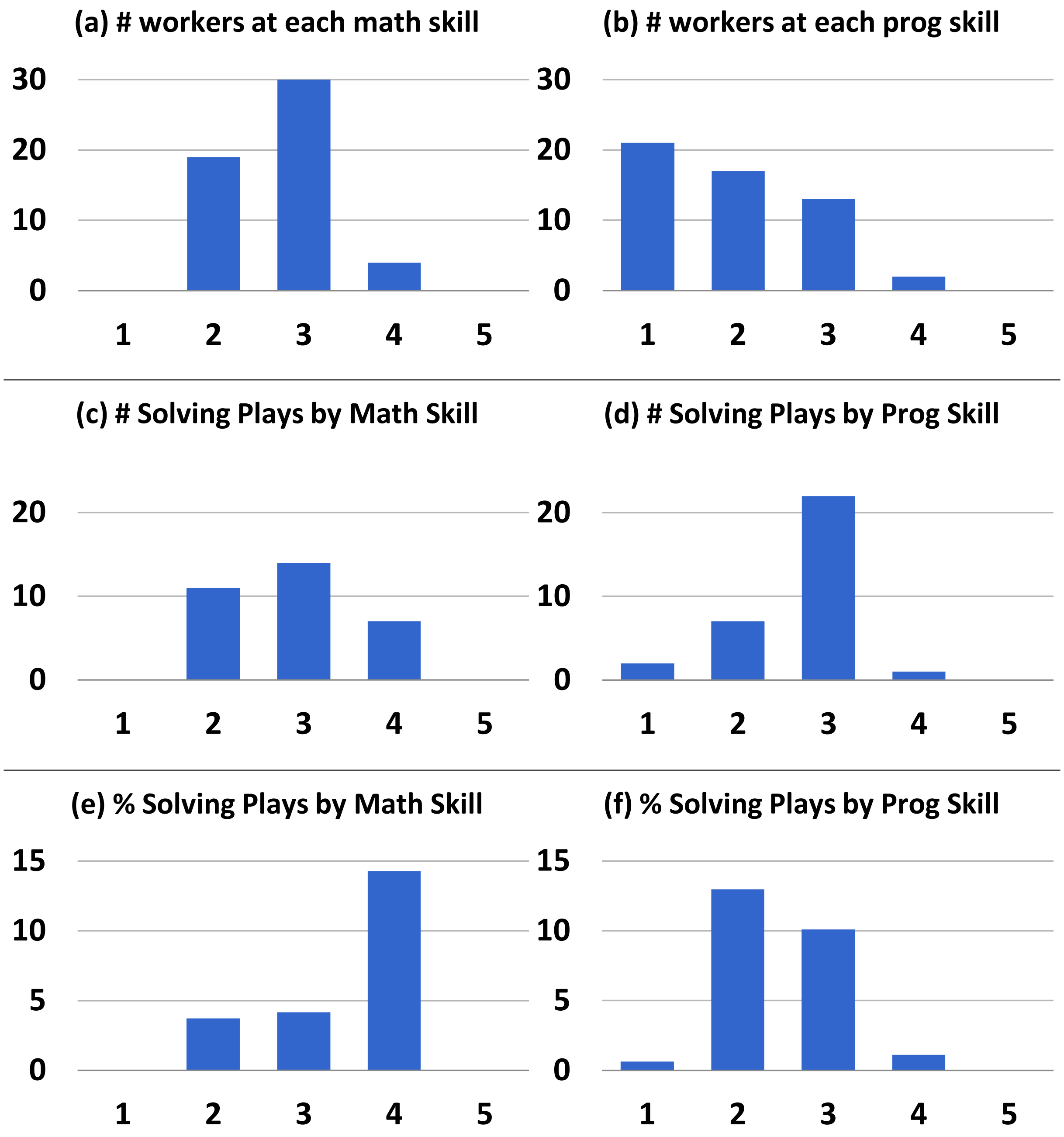


Figure 3.2. Number of players by math and programming skill

and programming in (d). This shows us that in absolute numbers, plays at programming skill 3 contribute the most.

To better understand the performance of each skill rating, Figures 3.2(e) and (f) show the number of solving plays at a given skill divided by the total number of plays at that skill — math in (e) and programming in (f). For example, the bar at math skill 3 in Figure 3.2(e) tells us that about 4% of plays at math skill 3 are solving plays. This gives us a sense of how productive plays are at math skill 3.

Looking at Figures 3.2(e) and (f), if we ignore programming skill 4 — which is difficult to judge because it has so few plays, as seen in Figure 3.2(b) — there is in general a trend, both in math and programming skill, that higher skill ratings lead to higher productivity.

In addition to looking individually at math and programming skill, we also looked at their combination, in particular: (1) the average of math and programming skill (2) the maximum of math and programming skill. Although we don't show all the charts here, one crucial observation emerged from examining the maximum of math and programming skill. To be more precise, we define the *mp-max* skill to be the maximum of the math skill and the programming skill. We noticed that *all* solving plays had an *mp-max* skill of 3 or above, in other words a skill of 3 or above in math *or* in programming. Furthermore, this was not because we lacked plays at lower *mp-max* skills: 35% of plays have an *mp-max* skill strictly less than 3, in other words a skill below 3 in *both* math and programming. These 35% of plays did not find any solutions. In short, INVGAME does not enable players with low math *and* low programming skill to find loop invariants: instead the most effective players are those with at least a score of 3 in math or programming.

Difficulty of Benchmarks. Having focused on the effectiveness of players over *all* benchmarks, we now want to better understand what makes a benchmark difficult and this difficulty relates to what skill is required to solve it.

One clear indication of the difficulty is that some benchmarks were not solved. Looking more closely at the four benchmarks that were not solved by INVGAME , we found that for `cube-1`, players found `c==i*i*i` many times, but not `i*(i+1)=2*a`; and for `cube-2` players found `6*(n+1)` and `x==n*n*n` but not `y==3*n*(n+1)+1`. From this we conclude that more complex polynomial relationships are harder for humans to see.

Focusing on levels that *were* solved, let's first consider `gauss-1`, which was only solved once in a preliminary trial with online Mechanical Turk workers. The player who solved `gauss-1` had a math skill of 2 and programming skill of 3, which is average for our population. This occurrence resembles cases observed in Foldit [22], where some of the biggest contributions were made by a small number of players with no formal training in the matter, but with a strong game intuition.

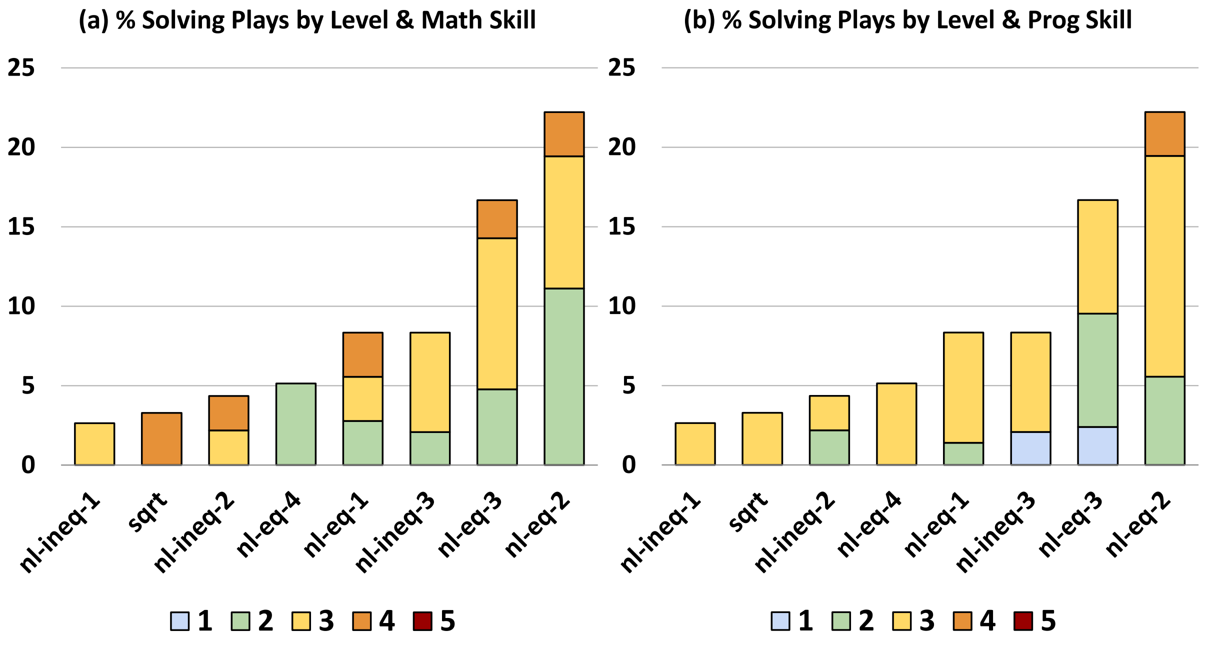


Figure 3.3. Percentage of solving plays (broken down by experience)

For the remaining benchmarks that were solved, a proxy measure for difficulty of a level might be the number of solving plays for that level. The more frequently a level is solved, the easier the level might be. Because there is variation in the number of times each level was played, we normalize the number of solving plays for a level by the total number of plays for that level. This gives us, for each level, the percentage of plays that were solving plays. Figure 3.3(a) and (b) show this percentage for all solved levels — except for `gauss-2`, because it was solved by multiple plays, and `gauss-1` because as we mentioned earlier we don't include the data for that benchmark here. For example, we can see that for `nl-eq-2` about 22% of plays were solving plays, and for `nl-ineq-1` about 3% of plays were solving plays. The levels are ordered from hardest at the left (the least number of solutions) to easiest at the right (most number of solutions).

Furthermore, we divided each bar into stacked bars based on math and programming skill of the solving plays. Figure 3.3(a) shows the chart divided by math skill, and Figure 3.3(b) shows the chart divided by programming skill.

Plotting the data in this way gives us a sense of how much each skill contributed to the solving plays for each level. We can notice, for example, that programming skill 3 appears on all benchmarks, meaning that if we only keep programming skill 3 plays, we still solve the same number of benchmarks.

We also expect that the hardest levels might require more skill to solve. Thus, moving from right to left in the bar chart, we would expect the bars to contain larger proportions of high skills. While not perfectly observed in Figure 3.3, we do see signs of this pattern. For example, scanning from right to left in the chart organized by math skill – 3.3(a) – we see that math skill 2 (green) disappears: the hardest three benchmarks are only solved by math skill 3 and 4. Scanning from right to left in the chart organized by programming skill – 3.3(b) – we see that programming skills 1 and 2 shrink and then disappear.

3.4.4 Solution Cost

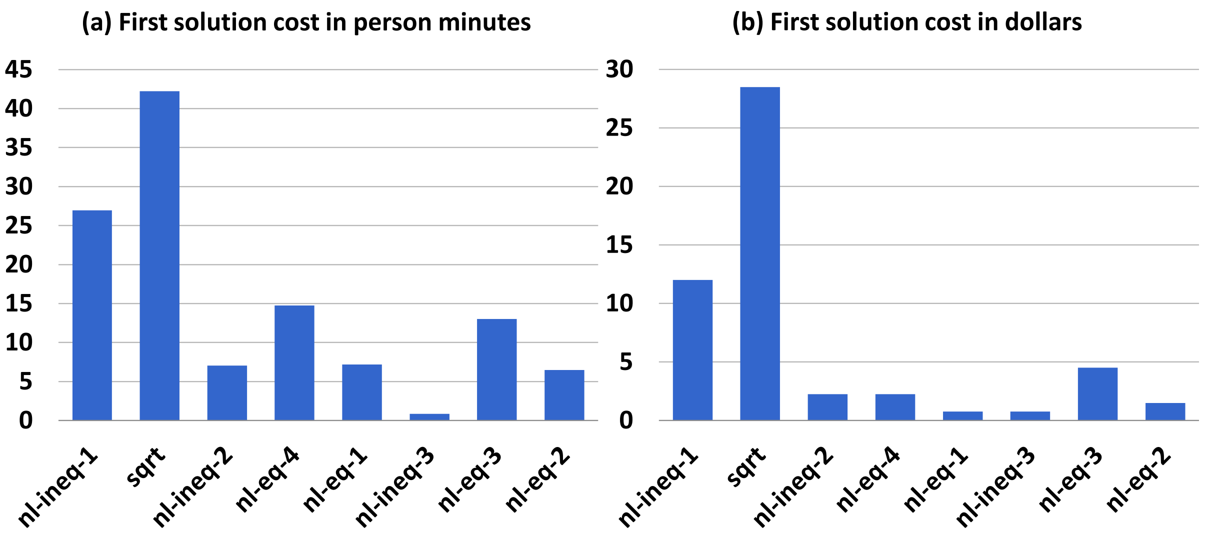


Figure 3.4. Cost of first solution in total player time and money

In practice, if we were trying to solve verification problems with INVGAME , we would have InvGame serve a given level until it was verified. Because all of our data has time-stamps, we can simulate this scenario from our data. In particular, given a level, we can measure the cost to get the first solution for that level, both in minutes of game play for that level, and in terms of money spent on that level. Figure 3.4(a) shows the first solution cost in minutes of gameplay for each level, and Figure 3.4(b) shows the first solution cost in dollars for each level. The benchmarks are ordered in the same way as in Figure 3.3. As expected, we notice a broad trend which is that benchmarks that were measured as harder according to Figure 3.3 tend to cost more to solve.

On average the first solution required 15 minutes of player time and cost \$6.5. Also, although we don't show a chart with the number of unique players to the first solution, on average, the first solution required about 8 plays and 4 unique players.

3.4.5 Player Creativity

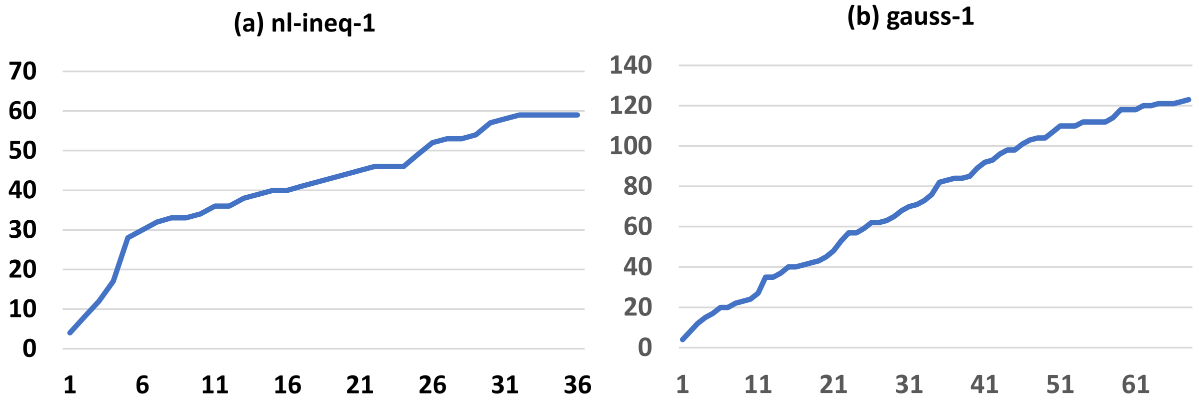


Figure 3.5. Cumulative # of semantically new predicates vs # of plays

A key assumption of our game is that players are good at coming up with new predicates, in the hope that they will eventually hit upon a verification invariant. To get more detail on player creativity, we plot, over time, the cumulative number of predicates entered by players that were *semantically new*, meaning that they were not implied by any of the previously entered predicates. There are 14 such plots, two of which are shown in Figure 3.5(a) and (b).

To gain insight on whether these trends had flattened by the end of our experiments –

signaling an exhaustion of what players are able to invent for accepted predicates – we computed the fits of two trend lines, one on the assumption that new predicates were continuing to come (a linear regression), and one that they were not (a logarithmic regression). Our intuition is that, given enough runs, we would see a leveling off in the discovery of semantically new predicates. This would mean that the logarithmic regression would be a clearly better fit. However, we are *not yet* seeing this for our benchmarks: the linear regression is a better fit (i.e., had a better R^2 value) in 12 of the 14 benchmarks. Furthermore, the R^2 values of *all* regressions, linear or logarithmic, were uniformly high, in the range 0.86 to 0.98 — with $R^2 = 1$ indicating a perfect fit. The most logarithmic plot is shown in Figure 3.5(a). This tells us that, despite having between 36 and 72 plays for each benchmark, we have not yet seen enough runs to detect a clear leveling off of semantically new predicates. This suggests that we have not yet exhausted the creativity of our players in terms of coming up with semantically new predicates. Besides being indicative of the creative potential of our players, it provides hope that further gameplay could solve more of our four remaining unsolved benchmarks.

3.4.6 Player Enjoyment

We asked players to rate the game from 1 to 5 on how much fun they had. The average rating for fun was 4.36 out of 5. Also, we received several emails from users asking us when more HITs would be available. One user even referred to us as their favorite HIT. Note that our experiments do not show that participants are willing to play the game just for fun, without being paid. Still, even with players being paid, the rest of our data analysis is valid with regard to (1) how humans play INVGAME , (2) the solving power of humans vs. automated tools (3) the relation of player skill to solving potential.

The balance of intrinsic vs. extrinsic motivation (e.g. monetary payment) in game design is an interesting topic of study [87]. Certainly, extrinsic motivation is not inconsistent with the idea of gamification, in that some citizen science projects use both [22, 104]. Recent research also suggests that hybrid schemes of motivation (mixing intrinsic and extrinsic) are worth studying in terms of player engagement, retainment, and effectiveness [87, 104, 115].

In addition to the “fun” rating, we also asked players to rate the game from 1 to 5 on how challenging they found it. The average rating was 4.06, suggesting users found this task fairly difficult. The fact that players found the game challenging and yet fun might suggest that INVGAME is a “Sudoku” style puzzle game, where players enjoy being engrossed in a mathematically challenging environment.

3.5 Design Insights

We believe these were several important design insights that led to our approach being successful.

Less abstraction has benefits. In INVGAME we chose a point in the design space that exposes the math in a minimalist style, without providing a gamification story around it. As we discuss in our related work, this is in contrast to other games like Xylem [72], which provide a much more polished and abstracted veneer on the game. Our minimalist user interface scales better with content size: our unencumbered table can convey a lot of information at once. Furthermore, we believe INVGAME was successful at solving problems in large part because of the lack of abstraction over mathematical concepts: we are directly leveraging the raw mathematical ability of players.

The diversity of human cognition is powerful. Players in INVGAME don’t see code or asserts. Yet they are still able to come up with useful invariants. In fact, if players played perfectly, always finding the strongest invariants given the traces they saw, they would often miss the invariants we needed. In this sense, *the diversity of human cognition is extremely powerful* as it allows the crowd to infer expressions without overfitting. Crowdsourcing, along with our scoring incentive of power-ups, together harness this diversity of cognition.

While looking at predicates entered by players, two cases stood out as exemplifying the diversity of human cognition.

First, a player found `su == (a+1)*(a+1) && su==t+a*a` for `sqrt`, and this solved the `sqrt` level. At first we were surprised, because this looks very different from the solution we expected from Table 3.1, namely `su==(a+1)*(a+1) && t==2*a+1`. However, upon closer examination, the player's predicate is semantically equivalent to ours, even though syntactically different. The fact that a player found such a different expression of the same invariant, which required us to think twice about how it could actually solve the level, is a testament to the diversity in cognitive approaches and pattern recognition that different players bring to the table.

Second, in `nl-ineq-1`, many players found a compelling candidate invariant that we did not expect: `(i+1)*j==k`. This candidate invariant is stronger than (i.e., implies) the needed verification invariant, namely `i*j<=k`. The candidate invariant `(i+1)*j==k`, despite its compelling nature (upon first inspection we mistook it for another correct solution), is actually *not* a valid invariant, even though it held on the data shown in the game. What is interesting is that if all humans had played the game perfectly, finding the best possible invariant for the data shown, they would have only found the incorrect stronger `(i+1)*j==k`, which would have prevented them from entering the correct one, `i*j<=k`. In this sense, the diversity of human ability to see different patterns is crucial, as often the strongest conclusion given the limited data shown is not the one that generalizes well.

Law of Proximity. We found that column proximity affected the ability of players to see certain patterns: it was easier to discover patterns in closer columns. This insight is captured by the *Law of Proximity*, one of the Gestalt Laws of Grouping, from the Gestalt Psychology [110, 111]. The law of proximity states that objects that are close together appear to form groups. This law has a long history of being used in HCI design, including for web page design, menu design, and form design [110, 111]. In our setting, columns are perceived as grouped based on their proximity, which then affects the kinds of relations that players see.

Reducing the Two Gulfs by Changing the Task. One way of assessing the cognitive burden of a task is to consider the two well-known "gulfs": (1) *gulf of execution* [80]: the difficulty in understanding what actions should be taken to achieve a given goal; (2) *gulf of evaluation* [80]:

the difficulty in understanding the state of a system, and how the system state evolves. When a human does program verification the traditional way, using loop invariants and a back-end theorem prover, both gulfs are large: it's hard to determine what invariant to use by looking at the program (gulf of execution); and it's also hard to predict how the back-end theorem prover will respond to a given invariant (gulf of evaluation). By transforming the task into one where humans look at run-time data, instead of programs, and try to come up with predicates that hold on this data, we significantly reduce *both* gulfs: it's easy to understand how to enter predicates that make all rows green to win points (gulf of execution); and it's easy to understand that the system just evaluates the predicates entered to either true (green) or false (red) on the given data (gulf of evaluation).

This reduction in cognitive burden, through a reduction in the two gulfs, explains in theoretical terms why players with no verification expertise can play INVGAME . However, these choices have also robbed the player of *fidelity* with respect to the original verification task: the player does not see the program, or the asserts, or any feedback from the theorem prover. This seems to indicate an *increase* in the gulf between our game and the *original verification* task. How is the game still successful at verification? This fidelity is in fact not always needed; and when not needed, it can be cognitively burdensome, *especially* for non-experts. The ultimate insight is that a low fidelity proxy which reduces cognitive burden and also maintains enough connections to the original task can allow non-experts to achieve expert tasks. Also, from the perspective of game design, our choice of proxy provides another benefit: INVGAME 's frequent positive feedback for entering novel predicates is likely more rewarding and motivating than the frequent feedback on incorrect verification invariants.

Some of the above design insights, individually, can certainly be connected to prior work (e.g.: some VeriGames projects change the task [72], Foldit removes a lot of abstraction [22]). However, our work also provides empirical evidence that the above design insights can be incorporated together into a system that makes players effective at loop invariant inference compared to state-of-the-art automated tools.

3.6 Acknowledgments

This chapter, in full, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 4

FLOWGAME

4.1 Overview

With INVGAME we decided to hide the underlying program structure from the player and to focus their attention on the mathematical patterns in run-time traces. Furthermore, we asked players to solve a simpler problem — satisfying a limited set of runtime traces — as a high-fidelity proxy for the underlying task of finding sound loop invariants.

This decision had several benefits — it allowed for a game that does not require understanding the underlying programming language or the mechanism of verification. Furthermore, it allowed for immediate visual feedback, as the game became only loosely tied to the underlying SMT solver, which has the potential to time out on some queries. Finally, it allowed for more frequent positive feedback, as it is easier to discover true predicates for a limited set of traces than to finding inductive invariants. Candidate predicates over traces can also be discovered in any order. If the game required players to find invariants that the backend solver can immediately show to be sound, then for some levels users would be forced to discover invariants in a specific order, and some invariants would only be allowed grouped together.

The INVGAME design, while simplifying the problem, held back a lot of information that would have been useful to players to solve the underlying verification task quicker (or tackle more complex tasks). Namely, there are 2 types of information that would be beneficial — 1) the structure of the underlying program and 2) the counterexamples provided by the underlying solver.

4.1.1 Exposing Program Structure

During the evaluation of InvGame we noted 2 particular cases where hiding the underlying program structure from players presented a serious obstacle to verifying a level.

The first case is benchmarks that require implication invariants. In our experience the precedent necessary for those implications frequently appears in the branch expression of conditionals in the program, or in the loop condition itself. For example, we found that in 38 out of the 39 benchmarks in our benchmarks set that involve implications, the precedent can be found in the program text, usually in a branch condition in or before the loop, the loop condition or in the postcondition. We experimented with simple heuristics for selecting a precedent as mentioned in Section 6.1.2. Our experiments showed that we can split traces based on candidate predicates and successfully use Daikon or InvGame to infer the antecedent for implications, and in many cases solve implicative levels. We do not further report on those results, as it seemed that once we empowered Daikon with syntactically inferred precedents there were not many levels left on which to evaluate how human players can add value on top of automated tools.

The second case is benchmarks where knowing the goal of the verification (i.e. the required post-condition) significantly aids in finding the right invariants. While there are many patterns, and consequently valid invariants on which a user can spend their time and effort, not all of them are required to prove a given post-condition. Knowing for example what variables are in the post condition, or what kind of relation is sought after (e.g. equality vs inequality) can be a valuable hint for the user as to where to direct their attention.

Furthermore, even the syntactic shape of the post-condition can be a valuable hint for a player. For example consider the benchmark in figure 2.1, coming from the ICE-DT benchmark suite. The postcondition `2*sum == n*(n+1)` is very similar to the required invariant `2*sum == i*(i-1)`. In 27% of benchmarks we found that the post-condition expression appears in the required invariant either unchanged, or same modulo variable substitution.

4.1.2 Exposing Verification Counterexamples

A second piece of information that is useful to finding inductive verification invariants are the counterexamples provided by the underlying solver. Exposing counterexamples to players is one mechanism to explain to players the gap between their current candidate invariants and the yet-unknown correct invariants. In section 6.1.4 we discussed early experiments in trying to display pre-condition and inductiveness counterexamples. In those experiments, even experts found gameplay with counterexamples in INVGAME confusing.

Explaining counterexamples to players (especially inductive ones) requires teaching non-experts intuition very similar to the intuition experts build while performing Hoare-style logical reasoning. It is a significant challenge to incorporate such a level of complexity in a game design, while keeping it accessible to a wide audience.

4.2 FLOWGAME Design

As observed in previous sections extending INVGAME with program information and counterexamples made the original game more confusing. As a result, we decided to design a different game — FLOWGAME — that exposes the essential parts of the underlying program and communicates verification failures faithfully to the player. In FLOWGAME , unlike in INVGAME , the game goal is precisely aligned with the verification goal. There are three basic ideas that drive the design of FLOWGAME :

1. Interactive verification can be viewed as a turn based game with the solver as the adversary. An expert's 'play' is a suggested constraint on the adversary in the form of an invariant, whereas the adversary's 'play' is a concrete counterexample to the player's constraints.

2. The verification game can be visualized over a stylized control-flow graph (CFG). The CFG allows for an engaging and clear visualization of counterexamples, and the flow of counterexamples across a graph resembles an existing category of games called 'tower

defense games'.

3. The semantics of the underlying program, the specification primitives (`assume` and `assert` statements) and even the intuition of inductiveness can be described using just 4 visual elements in the game. In particular, a loop invariant can be viewed as a sequence of an `assume` and `assert` that share the same expression.

The above ideas drove us to build a game that initially at least resembles a turn-based tower defense game, where the player controls only a pair of towers (corresponding to the loop header in the underlying solver). The game goal is to put constraints on a cunning adversary that can spawns any forces he chooses, subject to the constraints imposed by the player.

4.2.1 Overview

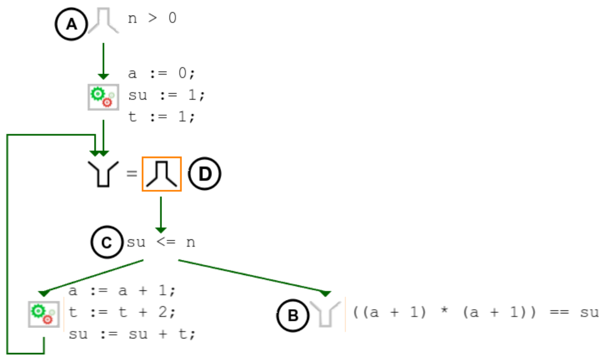
We use the running example shown in Figure 4.1 to describe the InvGame layout and gameplay. The code in Figure 4.1a, and for the remainder of this chapter is presented in the Boogie [68] verification language. Boogie is a low-level imperative verification language with built-in support for expressing pre and post conditions as well as loop invariants. Both INVGAME and FLOWGAME use Boogie as their backend representation for levels. You can find more information on the syntax and semantics of Boogie in [68].

In figure 4.1a we show an example program in Boogie that computes successive squares, and in figure 4.1b the corresponding FLOWGAME level for that program. A level shows a stylized control-flow graph (CFG), with the minimum syntactic information need to convey the level semantics. Basic blocks and certain expressions and statements are represented by the different icons on screen. Icons in FLOWGAME are connected by green lines showing the possible paths between basic blocks in the CFG. So for example the program in fig. 4.1a contains a single loop, and correspondingly there is a single loop formed by the green lines in fig. 4.1b

The game screen in figure 4.1b displays verification and control flow relevant expressions. For example, the icon labeled Ⓐ with expression `n>0` in fig. 4.1b corresponds to the `assume`

```
1    var a,t,su,n: int;
2    assume (n > 0);
3    a := 0;
4    su := 1;
5    t := 1;
6
7    while (su <= n)
8    begin
9        a := a + 1;
10       t := t + 2;
11       su := su + t;
12   end
13   assert(su == (a+1)*(a+1));
```

(a) Source code



```
n > 0

a := 0;
su := 1;
t := 1;

su <= n

a := a + 1;
t := t + 2;
su := su + t;

((a + 1) * (a + 1)) == su
```

(b) FLOWGAME game screen

Figure 4.1. FLOWGAME example — square

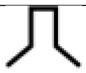
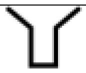
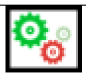
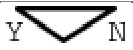
statement on line 2 of fig 4.1a. The icon labeled Ⓑ with corresponding text `(n+1)*(n+1)==su` at the bottom right of fig. 4.1b correspond to the `assert` statement on line 13 in fig. 4.1a. Finally, the icon labeled Ⓒ with associated text `su<=n` in the middle of figure 4.1b corresponds to the loop condition on line 7.

The game is centered around reasoning about solver counterexamples, visualized as variable assignments flowing between icons along green arrow paths. For example in Figure 4.2a, a counterexample containing the assignment `n=1` is shown at the top.

There are 4 types of icons on a FLOWGAME screen, defined in table 4.1. The first icon represents a source for counterexample values. It is depicted by a funnel opening downwards, with an associated expression. The game semantics of a source icon is that it can emit *any* variable assignment that satisfies the accompanying expression. This game semantics precisely mirrors the semantics of the `assume` statement in Boogie.

The second icon represent a sink for counterexample values. It is depicted by a funnel opening upwards with an associated expression. The game semantics of a sink icon is that it can 'eat' *any* example variable assignments that satisfies the accompanying expression. If an example variable assignment reaches a sink node and does not satisfy the node's expression, it

Table 4.1. FLOWGAME icons

Icon	Name	Code Equivalent	Description
	Source	`assume <expr>`	Source of counterexamples values
	Sink	`assert <expr>`	Sink for counterexamples values
	Transformer	`x:= ...;` `y:=...;`	Sequence of assignments
	Branch	`if <expr> {` `...` `} else {` `...` `}`	Branch point for counterexample values

results in an 'error' that the player must fix. This game semantics closely mirrors the semantics of the `assert` statement in Boogie.

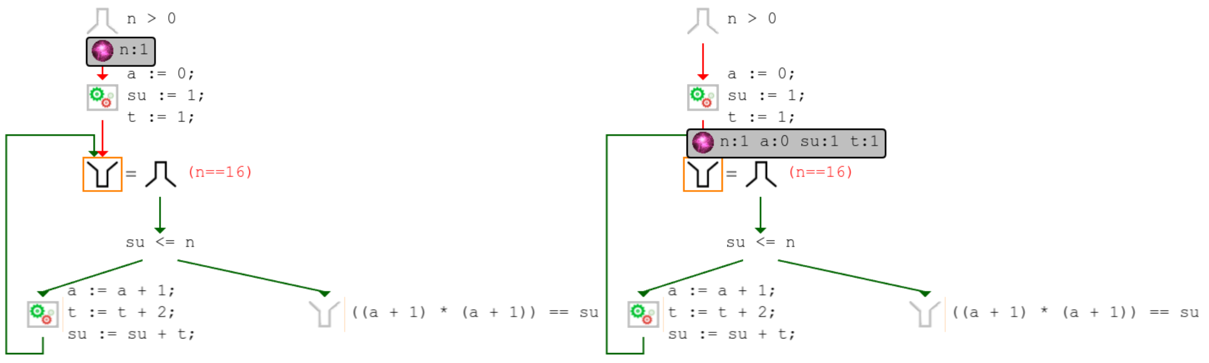
The third icon represents a sequence of assignments mutating the program state. It is depicted by a square gearbox, the gears signifying that this block transforms the variable assignments flowing through it. The fourth icon represents a branch node, and has an associated conditional expression. Its in-game semantics are that for every example assignments that flows through it, the associated conditional expression is evaluated, and depending on the outcome, the assignment flows left if conditional expression is true and right otherwise. This icon corresponds to a branch in Boogie.

Finally, in Figure 4.1b the source/sink pair labeled Ⓓ are connected by an equal sign. This visually shows that the particular source and sink share a single expression. A linked source/sink pair corresponds to a loop invariant and is the only part of the game that a user interacts with. By attempting to add conjunctions to the linked source/sink pair players are trying to constrain the loop invariant. The idea of a linked source/sink pair corresponds to the fact that loop invariants can be expressed as a sequence of an `assume I;` and an `assert I;` statements that share a common expression `I`. This explanation is convenient for us as we don't need a separate visual element with which to explain invariants.

The game mechanic described so far draws inspiration from tower defense games: 'bad' values flow along fixed paths, and it is the player's goal to interact with the map until no more bad values can flow.

4.2.2 Game Play

Solver counterexamples are concrete traces along a particular path through the program. They are visualized as objects flowing along the program control flow graph, that contain concrete variable assignments. In figure 4.2a we show a concrete counterexample flowing from the start of the program to the loop header.



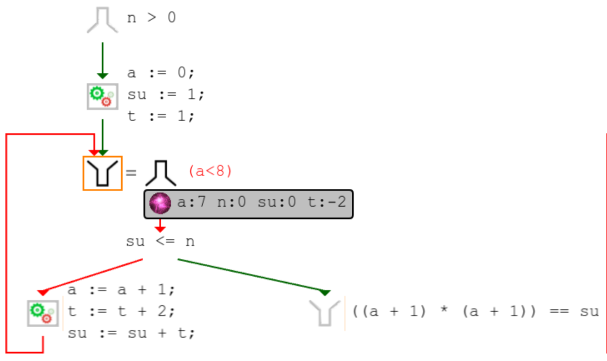
(a) Before assignments **(b)** After assignments

Figure 4.2. FLOWGAME — precondition counterexample

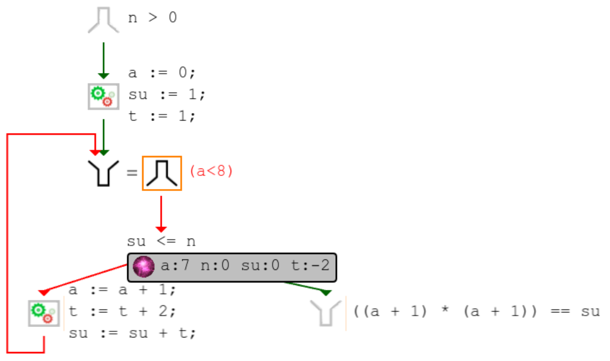
The concrete counterexample is displayed as a gray box containing the concrete variable values. For example in figure 4.2a the counterexample shows that right at the beginning of the program `a` has a value of `0` and `n` has a value of `1`. Players can move the gray counterexample box back and forth using the keyboard. As the counterexample box moves across transformer boxes for example, players can see the values change. Figure 4.2b shows the game after the player has moved the counterexample past the first transformer box, and shows the effect of its assignments reflected in the counterexample — `su` and `t` now appear in the counterexample box.

A player interacts with a game by changing the expression associated with the loop invariant. For example, in figure 4.2a this is the red `n==0` expression next to the Invariant node. The counterexample displayed in figures 4.2a and 4.2b is a pre-condition counterexample — i.e.

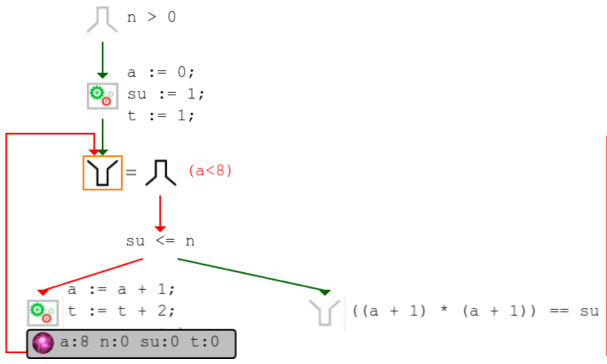
it signifies that the current candidate invariant (n==0) is too strict — it is not implied by the loop's precondition (n>0).



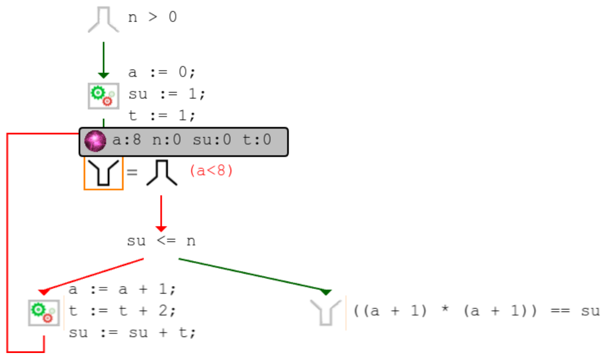
(a) At start of loop



(b) After branch



(c) After loop body



(d) At end of loop

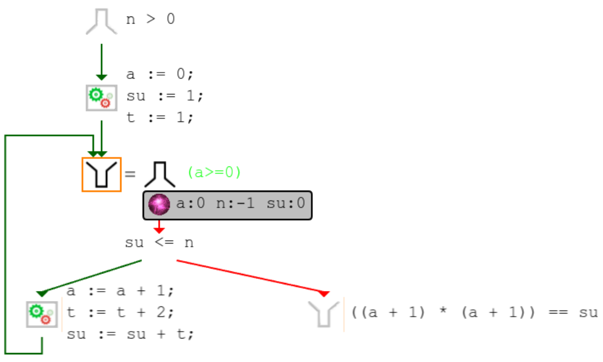
Figure 4.3. FlowGame — inductive counterexample

Next a player may try a<8 as an invariant, as shown in figure 4.3. a<8 is not inductive, and thus the four panels in figure 4.3 show a counterexample that spans a single loop iteration. In fig. 4.3a we see that if we start with a program state where a==8 && n == 0 && su ==0 then we will go inside the loop (fig. 4.3b), after the loop body a will become 9 (fig. 4.3c) and finally that state makes it back to the loop header (fig. 4.3d), but it does not satisfy the loop invariant.

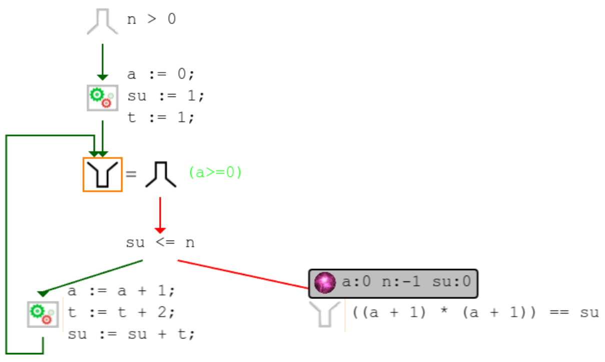
The counterexample in fig. 4.3 may look a little strange to the astute reader, as the values it contains do not constitute a reachable program state. There is no concrete program execution where at some point a==8 and su==0. This discrepancy is as a result of the fact that the loop invariant is the precise and complete summary of the program state at the loop header. Since the candidate invariant in the example (a<8) does not relate a and su in any way, the solver assumes that it can treat a and su independently. This also speaks to the difficulty of creating inductive invariants — they must preserve themselves across iterations, without assuming any additional information.

As users find more sound invariants (e.g. (a+1)*(a+1)==su), the counterexample values they observe become closer and closer to the actual reachable values.

Finally, a player may attempt to write a>=0 as an invariant. a>=0 is actually an inductive invariant. However, as shown in fig 4.4 this invariant is not strong enough to guarantee the safety of the program, so a counterexample is still generated. The counterexample in fig. 4.4 shows that the environment a==0 && n==-1 && su==0 satisfies the loop invariant a>=0(fig. 4.4a) but it flows to the assert after the loop(fig. 4.4b), and violates it.



(a) At start of loop



(b) At post-loop assert

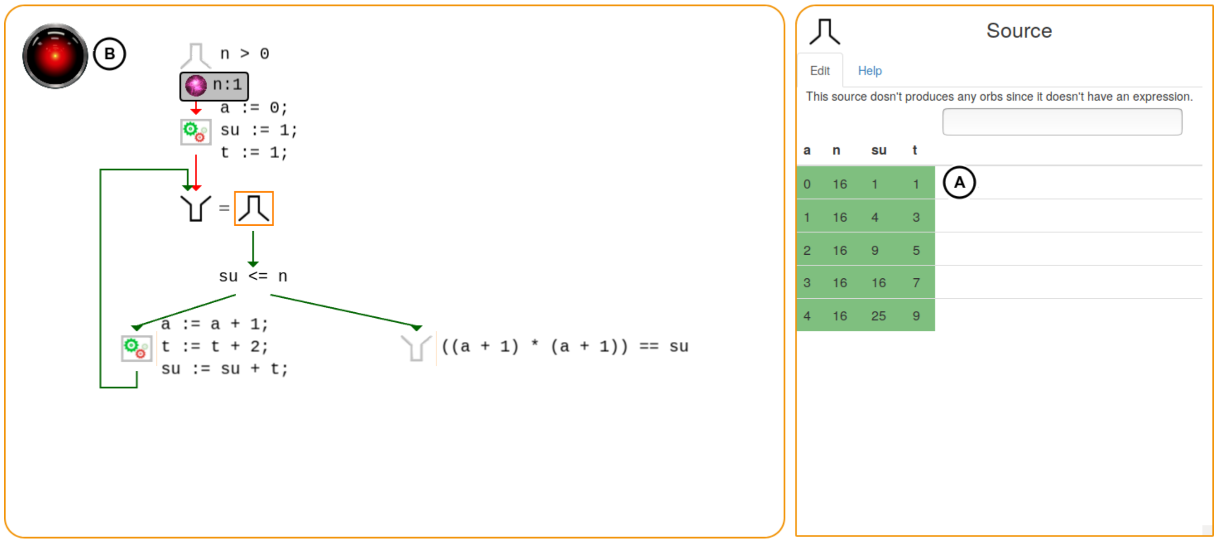
Figure 4.4. FLOWGAME — postcondition counterexample

A player's goal throughout this process is to find a set of invariants such that no more counterexamples are shown.

4.2.3 Early Design Evolution of FlowGame

Since our early design trials, the game as seen in Figure 4.4 evolved to the form shown in Figure 4.5. In the later version of the game there are two major extensions both motivated by user observations in our initial trials.

Need for positive examples. In early design trials we asked an expert to use FLOWGAME to solve the benchmark presented in figure 4.1b. Our expert player had a difficulty solving the level, and later remarked that ‘the UI explains how things *could* go wrong, but not how things work when they go right. In other words, the current game does not give the user intuition about the normal operation of the function since it only provides counterexamples. INVGAME on the other hand provides precisely an example of a loop operates under normal condition using runtime traces. As a result we decided to combine the two games as shown in figure 4.5



```
n > 0
n:1
a := 0;
su := 1;
t := 1;
=
su <= n
a := a + 1;
t := t + 2;
su := su + t;
((a + 1) * (a + 1)) == su
```

This source doesn't produces any orbs since it doesn't have an expression.

a	n	su	t
0	16	1	1
1	16	4	3
2	16	9	5
3	16	16	7
4	16	25	9

Figure 4.5. FLOWGAME and INVGAME combined

In the combined game dynamic a candidate invariant has to first satisfy the limited traces displayed labeled Ⓐ in Figure 4.5. After passing this first filter, the invariant is submitted to the backend to attempt verification, and upon failure a counter example is displayed. An invariant is only accepted if it is sound (i.e. only post-condition counterexamples are returned.) and it is displayed in the green box on the lower right corner of fig. 4.5.

Need for an adversary. In an early design trial a non-expert was confused as to who comes up with the counterexample variable assignments, and found it annoying that without explanation the game always foiled their attempts. Her verbal feedback led to the insight that the solver has a meaningful place in the game world as an actual adversary. As a result we added as an adversary the ‘evil computer’ (see the icon labeled Ⓑ in Figure 4.5). Having an adversary made explaining the game dynamics much simpler — the *cunning* adversary would always find an edge case variable assignment to ‘break’ a user expression, should one exist. Thus making it the user’s goal to outsmart it, and find expression for which *there is no* breaking assignment. This metaphor allows us to encode the idea of ‘soundness’ of an expression into the gameplay. During our later experiments, two of our users noted that they enjoyed playing against an embodied adversary.

4.3 Evaluation

We performed a preliminary empirical evaluation with a small number (n=12) of verification experts and non-experts, that provided useful information on future directions of research In our evaluation we seek information on (1) player effectiveness at verification, (2) sources of confusion and (3) player enjoyment and engagement with our game. While we report statistics on user’s effectiveness (e.g. number of solved levels) and self-reported perception of the game (in Section 4.3.2), the primary focus of our evaluation is on qualitative observations derived from user interviews and the authors’ in-person observations (presented in Section 4.3.3).

Performing a full formal evaluation of how well FLOWGAME enables non-experts to perform verification requires (1) a larger population sample, (2) addressing design issues in our current prototype and (3) an actual playable tutorial. In our current experiments we provided an informal verbal tutorial to all players. The subjective nature of verbal tutorial, and potential influence of factors such as the tutor’s body language, energy level, variations on phrasing can impact the player’s understanding and performance, as further discussed in section 4.3.4.

4.3.1 Experiment Setup

Each user experiment consisted of the following 4 steps:

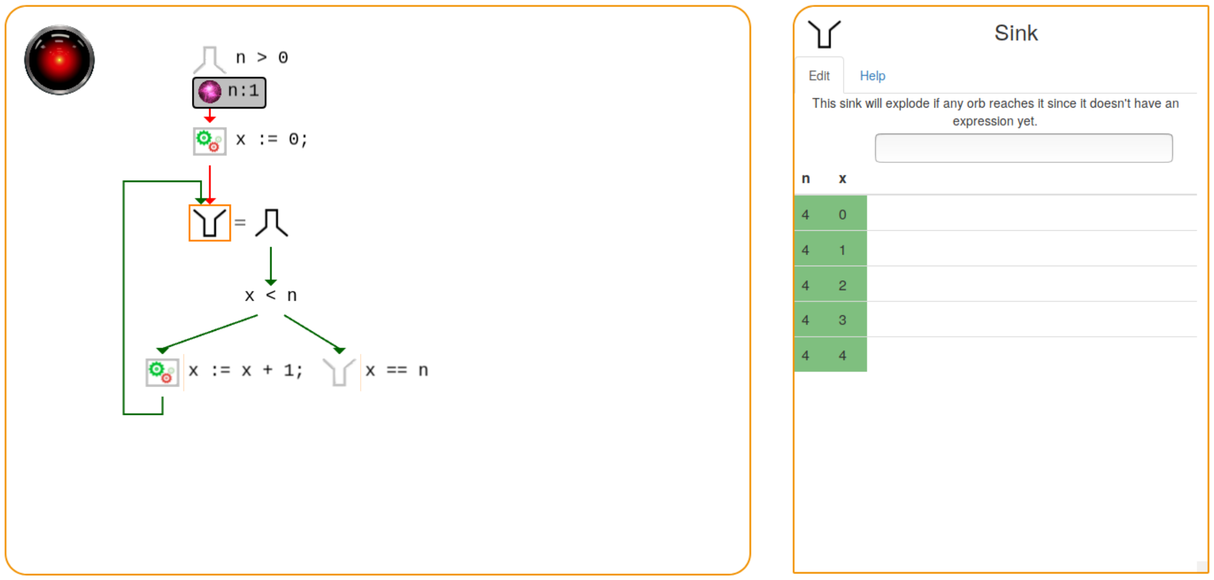
1. **Background questions**. In order to estimate the player’s previous experience, we asked them 3 background questions:

(a) What is the highest educational level at which you have taken a mathematics course? (E.g. high school, BS, MS, PhD)

(b) What is the highest educational level at which you have taken a computer science course? (E.g. high school, BS, MS, PhD)

(c) What is the order of magnitude of the largest code base you have worked on (if any)? In what language(s)? (E.g. ∼10K LOC, Python)

2. **Tutorial**. Users were given an informal verbal tutorial. They were presented with the game level shown in Figure 4.6.



```
n > 0
n:1
x := 0;
=
x < n
x := x + 1;
x == n
```

This sink will explode if any orb reaches it since it doesn't have an expression yet.

n	x
4	0
4	1
4	2
4	3
4	4

Figure 4.6. FLOWGAME tutorial level

To introduce the game world, the metaphor of cities and roads was used. The 4 different types of icons were described as cities, and the connecting green arrows as the roads between the cities. The orbs were likened to trucks carrying values between cities. Next the semantics of each city and the player's mode of interaction with the game was explained. Finally, the gameplay was illustrated with 3 different expressions, each causing a different type of counterexample.

First players were shown the result of entering `n<5` in the level shown in Figure 4.6, which resulted in a precondition counterexample. Next players were shown the invariant `n>=0` which is an example of a sound invariant, that generates a postcondition counterexample. Afterwards players were shown the `x<5` invariant that generates an inductive counterexample. Finally, they were shown the correct solution `x<=n`.

3. **Gameplay**. Users were asked to play as many levels as they felt like. They were allowed to ask questions, and were encouraged to think out loud. If a user was stuck for long time at a given level, they were offered hints. In most cases the hints only directed their attention to parts of the game interface, without specifying variables or expressions for them to try out. However, in a limited number of cases, parts of the relevant variables, or even parts of the correct invariant were pointed out in a hint.

   Throughout this phase notes were taken on every invariant attempted by users as well as their more relevant verbal points.

4. **Final Survey**. After subjects finished playing they were asked to rate how fun and how challenging they found the game on a scale of 1 to 5. Furthermore, they were asked to describe what they liked, disliked, and found confusing.

User Demographics

The study consisted of 12 subjects, 6 considered experts and 6 non-experts. A summary of the users previous experience as estimated by the preliminary questions can be found in Table 4.2. We considered users actively involved in research on Programming Languages and/or Software Verification to be experts in this field. In our study this involved 4 PhD and 2 BS students, all majoring in Computer Science.

Our 6 non-expert subject included 3 players with no prior experience in Computer Science and non-STEM related degrees, and 3 players with STEM degrees outside of computer science.

Table 4.2. FLOWGAME user study subjects

Subject	Math Class	CS Class	CS Experience
Non-Expert 1	2	0	None
Non-Expert 2	3	3	<10 LOC Excel in CS101
Non-Expert 3	3	3	<50 LOC Perl
Non-Expert 4	3	0	None
Non-Expert 5	3	0	<10LOC Python
Non-Expert 6	3	3	<1000 LoC
Expert 1	3	3	∼10K LoC Haskell
Expert 2	3	5	<100K C/C++
Expert 3	4	3	∼60K LoC Python
Expert 4	3	5	∼1M LOC C/C++
Expert 5	3	5	∼100K LOC Scala,JS,Haskell
Expert 6	3	5	∼10K LoC C/C++

Benchmarks

The benchmarks used in our study are shown in Table 4.3 along with their provenance and expected solution. Benchmarks with provenance ”ICE-DT” are selected from Garg et al.’s benchmarks from their paper on learning invariants using decision trees [41]. Benchmarks with provenance ”Dilig” were selected from Dilig et al.’s benchmarks accompanying their paper on learning invariants through abductive inference [30]. Benchmarks with provenance SV-Comp originate from the 2016 edition of the ”Competition on Software Verification (SB-COMP)” [1].

The benchmarks were ordered to provide a gradual difficulty slope, as well as slowly introduce more complex invariants.

Specifically the tutorial provides one of the simplest scenarios (variable incremented by 1 up to a bound) that allows us to showcase all 3 types of counterexamples.

The first level is a small variation on the tutorial, allowing players time to get used to the game and interface.

The second level introduces a multi-variable invariant for the first time and a non-inequality comparison operator.

The 3rd level is the first level requiring multiple conjuncts. Furthermore, there is an

Table 4.3. FLOWGAME user study benchmarks

Benchmark	Provenance	Expected Solution
Tutorial	ICE-DT	`x<=n`
1	ICE-DT	`x>=0`
2	SV-Comp	`x == y`
3	Dilig	`i >= 0 && sum >= 0`
4	SV-Comp	`x+y == n && x >= 0`
5	ICE-DT	`s == y*j && j <= x`
6	ICE-DT	`su == (a+1)*(a+1) && t == 2*a+1`
7	SV-Comp	`2 * sum == i * (i - 1) && i <= n + 1`
8	InvGen test suite	`2 * s == i * j && j == i-1 && i <= n + 1`
9	InvGen test suite	`i * (i+1) == 2*a && c == i*i*i`
10	Dilig	`(y<=0) ==> x<0`
11	InvGen test suite	`i == 2*k*j`

explicit dependency between conjuncts- `sum>=0` is not sound without `i>=0`. As discussed in Section 4.3.3 this dependency confused some players.

The 4th and 5th level introduce more binary arithmetic operators. The 6th, 7th and 8th level involve complex non-linear invariants corresponding to common high-school mathematical knowledge — notably the expansion of $(a+1)^2$ polynomial as well as the Gauss summation formula. The 9th level builds upon these with an invariant involving a polynomial of degree 3. The 10th level introduces a tricky implicative invariant, that confused even the authors. Finally, the 11th level continues with non-linear invariants, in case players made it thus far.

4.3.2 Quantitative Observations

Non-expert players completed between 1 an 5 levels (3 levels completed on average) during our experiments. Expert players completed between 5 and 11 levels (8 levels completed on average). Player's completion times per level can be found in Table 4.4. All expert players completed more levels than any of our non-expert players. Furthermore, for the levels completed by both Experts and Non-Experts (levels 1 through 5), experts are on average 3.7 times faster than non-experts. As expected the Expert players' knowledge allows them to recognize the underlying problem encoded in each game level and solve each task faster than a Non-Expert. Another indication of their understanding is the language they used while talking about the game. While throughout the tutorial the authors were careful to stick to the metaphors of ‘cities’ and ‘trucks’ moving throughout, Expert players quickly began referring to trucks as ‘counterexamples’ and to certain cities as pre- and post-conditions.

Table 4.4. Completed level timings

	Level Completion Time(s)										
Subject	1	2	3	4	5	6	7	8	9	10	11
Non-Expert 1	459										
Non-Expert 2	161	1113									
Non-Expert 3	684	272	607								
Non-Expert 4	511	52	816								
Non-Expert 5	154	505	342	821							
Non-Expert 6	274	171	137	1146	1278						
Non-Expert Avg	374	423	475	983	1278						
Expert 1	63	47	150	469	583						
Expert 2	152	31	735	1179	854						
Expert 3	83	39	110	185	450	996					
Expert 4	105	65	201	165	264	214	210	126	368	705	
Expert 5	66	60	82	61	84	368	190	71	469	197	102
Expert 6	44	25	181	132	176	395	233	303	254	320	133
Expert Avg	86	45	243	365	402	493	211	167	364	407	117

The subjects’ rating of how ‘fun’ and how ‘challenging’ they found the game are reported in Table 4.5. On average players scored their enjoyment of the game as 2.5 on a scale of 1 to 5. There is a significant difference between the reported enjoyment of Non-Experts (1.75 average) and the reported enjoyment of Experts (3.25). Furthermore, our Non-Expert subjects without experience in the STEM field (Non-Expert subjects 1,2 and 4) report the lowest enjoyment (1 average) across all. This confirms our intuition that our game is not enjoyable by people without experience and interest in mathematics or a related technical field.

While it is encouraging that two of our Non-Expert commented after the game, that they would like to play it again, and one of them even requested to play additional levels after the experiment had finished, the relatively low enjoyment scores indicate that a significant amount of effort would be required to make the game enjoyable to a wider audience. Furthermore, both a Non-Expert and an Expert player mentioned in their post-game comments that the game needs more visual stimuli to be more fun. Another player commented on the need for more 'small victories' to maintain their engagement. These comments further indicate the need for more game elements to increase intrinsic motivation.

Both non-expert players and expert players rated the game's difficulty as 4.6 on average, on a scale of 1-5. This indicates that all found it to be fairly difficult. This observation correlates with explicit comments made by players in the post interview (e.g. ”It was a challenging game”, ”steep learning curve”) as well as informal indications of the high cognitive load on players discussed in Section 4.3.3 (e.g. requests for paper scratch space).

Table 4.5. Player enjoyment and perceived difficulty

Subject	Like	Challenging
Non-Expert 1	1	5
Non-Expert 2	1	4
Non-Expert 3	1	5
Non-Expert 4	1	5
Non-Expert 5	3	5
Non-Expert 6	3.5	4
Expert 1	2	5
Expert 2	3.5	5
Expert 3	3	4
Expert 4	4	4
Expert 5	4	4
Expert 6	3	5

In Table 4.6 we present statistics on the expressions that players attempted to enter. Candidate expressions can be broken down into 4 categories depending on the possible outcomes for each expression (the 4 categories correspond to columns 36 in in Table 4.6):

1. **Green Rows Prevented (GRP)**. The candidate expression is prevented by the green rows - i.e. it is refuted by example traces

2. **Pre-Ctrex**. The candidate expression results in a counterexample to the precondition VC. In game terms — an orb traveling from the topmost source icon to the loop header is generated.

3. **Ind-Ctrex**. The candidate expression results in a counterexample to the inductiveness VC. In game terms — an orb traveling along the loop back edge is generated.

4. **Sound**. The candidate expression is sound, and is conjoined to the current invariant. If the invariant is strong enough to imply the postcondition the level is won, otherwise a counterexample to the post-condition VC is shown (in game terms — an orb traveling to the final sink is shown).

During our study subjects attempted 350 candidate invariants across all levels. Almost 50% of attempted candidate invariants were sound. Roughly 23% were prevented by green rows, 10% were overfitted resulting in a precondition counterexample and the remaining 17% resulted in an inductive counterexample.

We can compare side-by-side the breakdown of candidate expressions between experts and non-experts in Figure 4.7. A larger percentage of expert written invariants (55.6%) were sound compared to non-experts (35.5%). This matches our expectation that experts should perform better at our game. The increase of the share of sound expression for experts compared to non-experts comes at the expense of invariants prevented by green rows or precondition counterexample. Expert player expressions were prevented by green rows in 19.8% of cases compared to 32.7% of cases for non-expert players, and resulted in precondition counterexamples in only 7.4% of cases when compared to the 17.8% of cases for non-expert players.

4.3.3 Qualitative Observations

During our experiments participants were asked to think out loud. At the end of each trial they were also asked to provide feedback on things they liked about the game, did not like, and things they found confusing (more details on their responses can be found in Table A.2 in

Table 4.6. Attempted invariants breakdown

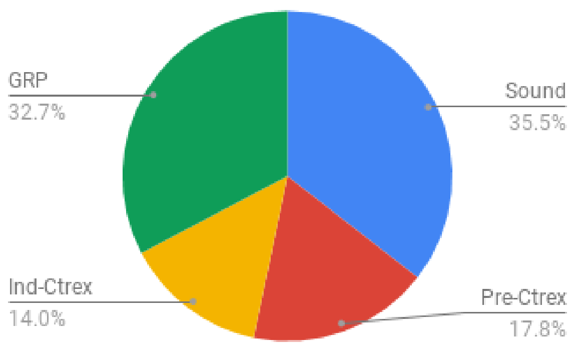
name	Total Invs Tried	GRP	Pre-Ctrex	Ind-Ctrex	Sound
Non-Expert 1	5	3	0	0	2
Non-Expert 2	24	11	5	1	7
Non-Expert 3	21	10	1	3	7
Non-Expert 4	14	4	3	2	5
Non-Expert 5	19	3	8	1	7
Non-Expert 6	24	4	2	8	10
Non-Expert Total	107	35	19	15	38
Non-Expert Ave	17.833	5.833	3.167	2.5	6.333
Expert 1	36	14	1	9	12
Expert 2	47	5	11	10	21
Expert 3	42	7	1	12	22
Expert 4	23	1	0	1	21
Expert 5	45	11	2	3	29
Expert 6	50	10	3	7	30
Expert Total	243	48	18	42	135
Expert Ave	40.5	8	3	7	22.5
All Total	350	83	37	57	173
All Ave	29.167	6.917	3.083	4.75	14.417

Appendix A.2). From the above two sources we synthesize the more common issues players faced when interacting with the game.

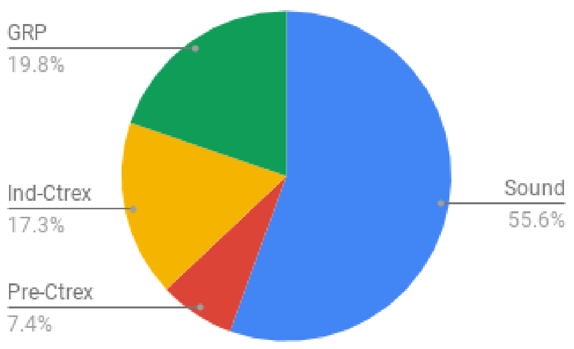
Steep learning curve/complex tutorial

Four out of the 12 participants noted in their feedback that the game had too steep a learning curve, or that the tutorial was too complex. Additionally, in 3 out of the 12 players we noticed they forgot simple concepts from the tutorial, in the first 2-3 levels immediately following. For example, we noted players forgetting that they have to click on the icons to uncover more information, or asking about what the green row's purpose was less than 5 minutes after they had been told. Finally, one of our non-expert players mentioned they were intimidated by the number of operators available to them from the start. They wished the tutorial and initial levels would start with a more restricted set of operators, to give them time to get used to them.

These observations indicate that there is too much material being packed in the relatively



(a) Non-expert expressions



(b) Expert expressions

Figure 4.7. Candidate expressions breakdown

short (∼15 minutes) verbal tutorial. These observations indicate that great care should be taken in designing the official game tutorial with a gentler pace of learning in mind. One potential technique for achieving this would be designing the tutorial with interspersed mini-games, that gradually introduce parts of the entire game, and allow people practice time to learn each concept individually.

Lack of historical context

Four out of the 12 participants noted during gameplay and in the followup questions that they lost their train of thought, or wished they could see their previously attempted ‘bad’ expressions. We observed one player using already discovered sound invariants as a form of an ‘undo’ button to remove the counterexample associated with the currently attempted invariant. These observations suggest that players would benefit from seeing their unsuccessful attempts as well as the accepted sound invariants.

Invariant Dependencies

In 9 out of the 12 user trials we observed players entering a correct invariant, and the game rejecting it because a necessary companion invariant was missing. For example, in Level 3, the required correct invariant is `sum >= 0 && i>=0`. Many players entered `sum>=0` which was rejected, as our backend could not establish its inductiveness without the other conjunct `i>=0`.

This behavior was observed both across expert and non-expert players. Upon entering the other conjunct `i>=0` expert players were especially surprised that the level was not solved, and some had not even noticed that `sum>=0` was rejected, and wondered where it had gone. Two expert players explicitly called out this behavior as confusing in their followup comments.

One idea to improve this scenario is to display the rejected expressions as suggested in the previous section, but also retry all rejected expressions that have not been categorically rejected by the backend (i.e. expressions not found to be overfitted) with every new expression attempted by the user. In this mode, from the user's perspective, it will be sometimes possible for an expression from the 'rejected' category, to sometimes magically move to the accepted category at a later point, thanks to some newly entered expressions.

Guidance from the Game

Several observations indicate that the game does not do enough to guide users to the correct solution. One of our non-experts explicitly noted that the game needs to provide more guidance, while one of our experts noted that it's too easy for "their flow to be broken" and that the game needs to provide more 'hand-holding animations'.

Since our goal is to eventually discover solutions that are not known a priori its not possible to guide users perfectly, however players encountered several cases that could be handled better.

Constant/Variable substitution. In 4 of the 12 cases we observed players entering the correct invariant modulo overfitted constants substituted for a variable. This scenario is usually observed when the green rows are overfitted and don't showcase a variable taking on different values. For example in Level 4 one of the expected invariants is `x+y=n`, however the traces for that level only show values for `n=4`. The overfitted trace frequently misled players into entering `x+y=4` instead of the sound `x+y=n`. One way to deal with this problem would be to utilize fuzzing to attempt to diversify traces and showcase differing values of `n` where possible. Another approach is to identify cases where players are shown only a single value for a variable, and under the hood also try any expressions involving the value with the variable substituted instead. Note that the same issue was observed in the original InvGame experiments, and the same ideas apply there as well.

Expression Evaluation. In 3 of the 12 cases we observed a player confused as to which of the multiple conjuncts of their candidate expression was being violated. We also observed that one of our players was confused by why a given counterexample moved in a particular direction through a branch. Both of these problem cases are caused by the game forcing players to substitute concrete values from a counterexample in symbolic expressions and evaluate those expressions in their head. A better game design would do the substitution and evaluation visually thus simplifying the player's work.

In the case of branch expressions, the game can be extended to highlight the variables in the counterexample that are involved in the computations of the current branch expression, and furthermore the partially evaluated branch expression can be shown. For example, for a counterexample involving `x=5, n=6` and a branch expression `x+1<=n` the partially evaluated `6<=6` can be shown with appropriate colors indicating whether it is true or false.

Similarly, when a counterexample reaches a sink, the violated conjunct and the participating variables in the counterexample box can all be highlighted appropriately. The partially evaluated conjunct after substitution can also be shown side-by-side to ease player's cognitive load.

The same idea can also be applied to counterexamples passing through transformer icons, with modified variables highlighted.

Precondition propagation. In 3 of the 12 cases, a player was confused whether expressions shown in the topmost sink of (corresponding to loop preconditions) apply automatically to the source/sink pair corresponding to the loop header invariant. One of our experts explicitly propagated those for several levels, even though they were not necessary for the solutions. As preconditions often involve conjuncts with variables that remain constant throughout the body of the loop, it is a simple task to extend the game to propagate these conjuncts to the loop header.

Miscellaneous

Finally, we observed several less common or less severe issues. In 3 of the 12 cases we noted players wishing for scratch space to work out a level. While some proposed improvements in previous paragraphs (especially more in-game expression evaluation) will reduce the player's cognitive load, and thus the need for scratch space, adding annotation capabilities is an interesting research direction, especially when combined with multiplayer modes.

Six of our subjects mentioned that the counterexample orb box was overlapping underlying text in confusing ways. Three players were also confused by the lack of labeling on outgoing edges from a branch (which direction corresponds to True and which to False). Another 3 subjects indicated that the game needs more visual appeal to foster player's enjoyment. This feedback indicates that the visual layout and effects of the game needs significant work to improve clarity and enjoyment.

Finally, in 5 of the 12 cases players complained about the metaphors used during the tutorial. One of the Non-Experts and 2 of the Experts found the metaphors to be confusing, while another 2 of the experts found them to be cumbersome and unnecessary. This indicates that more thought needs to be put into more appropriate game metaphors.

Summary

In summary, our experiments indicate that the FLOWGAME interface provides enough information for experts to be effective at verifying single-loop levels. While this is uninteresting as a result, it is an important sanity test, as the alternative — even experts being unable to solve a level, would indicate fundamental issues with the approach.

In terms of player effectiveness and enjoyment we see a clear trend with more exposure to programming being associated with better enjoyment and effectiveness. Non-experts with non-STEM experience reported the lowest enjoyment scores (1 on average) and solved the fewest levels (less than 3). Non-experts with STEM experience (outside computer science) solved more levels (up to the 5-th level) and reported higher enjoyment scores (2.5 on average) also corroborated by qualitative observations. Finally, experts solved all levels and reported the highest enjoyment score (3.25 on average). These results suggest that FLOWGAME is appropriate for people with some code literacy and mathematical inclination. As a result, we believe students in the STEM fields, and computer science undergraduate students would make a good target audience for FLOWGAME .

While some non-experts with experience in the STEM fields indicated they are enjoying the game, on average the low enjoyment score and user frustrations indicate the game is still a long way from being an enjoyable experience. Direct user feedback indicates that users find the game too complex and confusing. User feedback also hints that this is in part due to the lack of a good tutorial with a gradual learning curve. Additionally, our experiments suggest other improvements that would lower the cognitive load such as more historical context, and more visual assistance in evaluating expressions.

4.3.4 Threats to Validity

In our experiments looking we sought larger design issues with our game. There are several aspects of our experimental setup that may adversely affect our observations on which issues are the most pertinent. We present those below, along with notes on how to alleviate them in our future work.

1. **Small user sample size.** Due to the small number of subjects (`n=12`) it is possible that some issues were not observed, or the perceived severity of some issues is skewed by chance.

2. **User sample bias.** While we attempted to include both experts and non-experts in our study, there are still some biases in our demographics that may skew our results. For example most of our subjects are either pursuing a graduate degree, or involved in some form of research. On the other hand, since the target audience of our game is likely to be people with interest in STEM fields this particular bias may not be as crucial. It would

still be beneficial to perform a larger study with a wider demographics.

3. **Informal Verbal Tutorial.** While every attempt was made to maintain the same metaphors and same example expressions throughout the tutorial, it was still an informal effort. Variations in tutorial speed, body language, tutor energy levels do impact subjects perception. Furthermore, depending on how comfortable subjects felt asking questions, some may have gotten a better understanding. To control for these effects a formal online tutorial needs to be developed.

4. **Questions during gameplay.** Due to the absence of a formal tutorial and the early stage of the game subjects were allowed to ask questions throughout the experiment. Depending on how comfortable a subject felt asking questions, a subject may have gotten more information compared to others, and thus had performed better. In the presence of a polished formal tutorial, questions during gameplay can be disallowed.

5. **Hints.** Throughout the tutorial 31 total hints were given to players. Hints may impact perceived player enjoyment (one Non-Expert mentioned in their feedback that they enjoyed receiving hints) and may mask design problems. To control for this, experiments without hints need to be performed. As a large percentage of our hints also indicate design issues in the game, we report them here.

   Every effort was made to give hints that only point players at parts of the interface, rather than suggesting particular variables, operators or expressions. 21 of the 31 hints were focused on the tutorial or game UI/UX rather than suggesting concrete expressions to try. We consider these hints to be substitutes to missing game features. We summarize the breakdown of those 21 hints in Table 4.7. For completeness the remaining 10 hints that directly suggest expressions are listed in Table A.3 in Appendix A.2.

   The first hint in Table 4.7 refers to cases where we asked a player to step through a given counterexample, and explained to them why the values change in a given way and why the

Table 4.7. UI/UX hints breakdown

Hint	Count	Missing Feature
Verbally Step through Counterexample	4	Expression evaluation
Prompt User to Step Through Counterexample	3	Expression evaluation
Look at Green Rows	5	Visual Hint
Repeat Part of Tutorial	2	Better tutorial
User Green Rows as Calculator	2	Better tutorial
Re-write without division	2	Re-write in backend
Re-try an earlier expression	2	Automatic Re-try
Use a variable instead of a constant	1	Fix-up in backend

counterexample violates a given conjunct. We believe that the need for such hints would be reduced by adding visual evaluation of expressions and highlighting of violated conjuncts as described in Section 4.3.3. The second hint refers to just asking people to step through a counterexample without a verbal explanation. We believe the need for this hint will be reduced by visual evaluation and by automatically stepping through a counterexample upon its first appearance (this was explicitly requested by one of our expert players in their comments). In 5 of the cases where hints were needed, simply asking the player to look at the green rows was enough for them to see a pattern. The need for such hints can be reduced through visual hints highlighting of the green rows after a certain number of tries, and by more emphasis on the green rows during the tutorial. Finally, the last 3 types of hints (covering in total 5 of the hint instances) can all be handled transparently to the user in the backend.

Chapter 5

Related Work

While automated loop invariant inference is a very difficult problem there has been a rich body of research on this topic, due to its importance. We describe some of the work on automated invariant inference in Section 5.1. Next in Section 5.2 we discuss some more closely related work on applying Gamification to Software Verification, inspired by the success of Gamification and Crowdsourcing in other complex domains such as protein folding citefoldit. Finally in Section 5.3 we discuss recent efforts on the use of Machine Learning techniques for invariant inference. While Machine Learning approaches to invariant inference can also be seen as an instance of automated inference, we discuss them separately due to the close ties they have with gamification work.

5.1 Automated Invariant Inference

The field of automatic invariant inference has a rich history of research, of which we attempt here to summarize at least the main directions.

One of the primary techniques used for invariant inference has been Abstract Interpretation [24, 73, 23, 78, 56]. Work in this domain tends to restrict the shape of invariants with the choice of abstract domain. For example, the polyhedral [24, 73] and octagon [78] domains fix invariants to be a conjunction of linear inequalities. Predicate abstraction work [43, **?**, 36, 34, 58] takes as its abstract domain a finite set of atomic predicates over program variables and thus restricts invariants to be a boolean combination of the atomic predicates from the input set. Some works in this group such as Houdini [34] and Liquid Fixpoint [58] infer only conjunctions over atomic predicates, while others such as C2Bp [8] infer boolean formula involving both disjunction and conjunctions over predicates from the input sets. Both Houdini and Liquid Fixpoint instantiate the predicates from a set of expert driven templates. Unlike Houdini, Liquid Fixpoint also mines the program source for additional predicates.

Counterexample-guided abstraction refinement(CEGAR) [50, 9] approaches build upon predicate abstraction by allowing a lazy refinement of the set of atomic predicates, deriving new predicates from counterexample traces. Techniques using Craig-interpolation [77, 49, 76] further improve upon CEGAR inference by extracting a smaller set of locally-relevant atomic predicates from counterexample traces.

Another line of automated inference work [21, 45, 48] translates the verification conditions for invariant soundness and program correctness into non-linear constraints over program variables, that are passed on to a separate constraint solver. Thanks to the precise encoding, any solution to the systems of constraints is a sound invariant by construction. All 3 techniques are restricted to only inferring linear inequalities.

Work by Dilig et al. [31] used logical abduction for invariant inference. Their technique was limited to boolean combinations of inequalities from Presburger arithmetic as their technique relies on the underlying theory admitting quantifier elimination.

Daikon [32] and DDEC [95] both perform black-box invariant inference using runtime traces. Daikon instantiates statistically likely invariants from a limited set of templates, using logical implication to suppress redundant candidate invariants. DDEC infers linear equalities over liver variables by solving systems of linear equations built from runtime traces, using standard null-space computation.

In summary, traditional automated invariant inference work has made great strides in inferring invariants from various domains, but tends to be limited in the shape of invariants it infers either by the choice of abstract domain, or set of invariant templates used.

5.2 Gamification of Software Verification

Inspired by the success of projects such as Foldit [22] several lines of work have tried to apply gamification to the problem of Software Verification. One of the main differences between our work and these verification games is the extent of the empirical evaluation: we evaluate our system on a well-known set of benchmarks, showing that our system can outperform state-of-the-art automated tools. We can split existing work on gamification for Software Verification into games that expose math as part of the game play and those that conceal it.

Games that expose math. The closest approach to ours is Xylem [72, 70], a game where players are botanists trying to identify patterns in the numbers of growing plants. Like INVGAME , Xylem relies on players finding candidate invariants from run-time data. However, due to its focus on casual gaming, Xylem explores a different point in the design space than ours. Xylem is a touch-based game that uses a graphics-rich UI with many on-screen abstractions, including plants, flowers, and petals. Since these abstractions use space, Xylem is limited in the amount of data it can show onscreen, which in turn might hinder the ability of players to see patterns. For example, Xylem players can only see 2 data rows at a time, having to scroll to see other rows. In contrast, INVGAME tries to increase the ability of humans to spot patterns by: (1) showing the data matrix all at once (2) re-ordering columns. Xylem also restricts some aspects of predicate building, such as not allowing arbitrary numeric constants. In contrast, INVGAME has an unrestricted way for players to enter predicates, and provides immediate feedback on which rows are true/false, for quick refinement. Finally, although Xylem was played by many users, there is no systematic evaluation on a known benchmark set, or comparison to state-of-the-art tools.

Monster Proof [26] is another verification game that exposes math symbols to the player. However, Monster Proof is much closer in concept to a *proof assistant* [102, 2, 3], where the user is constructing a detailed proof using rules to manipulate expressions. In contrast, INVGAME uses humans to recognize candidate invariants, and automated solvers to perform much of the manipulations done by players in Monster Proof. To the extent of our knowledge, there is no published evaluation of Monster Proof.

Games that conceal math. There are several verification games part of the VeriGames project [26] that conceal underlying mathematical concepts, including Binary Fission [33], Circuit Bot/-Dynamkr [26], Pipe-Jam/Flow-Jam/Paradox [28, 26], StormBound [26] and Ghost Space/Hyper Space [112, 26].

In Binary Fission [33], players build preconditions by composing primitive predicates generated by Daikon. The user is never exposed to the predicates and instead builds a decision tree out of black-box filters, attempting to sort opaque positive and negative examples from runtime values. In our own work on INVGAME we evaluate on benchmarks involving predicates that Daikon cannot infer, which makes them difficult to solve by Binary Fission. Binary Fission was also evaluated only on loop-free programs, whereas in our own work the goal is to verify loops, which are one of the biggest challenges in program verification. The authors in [33] ask users to find preconditions for 7 functions in a piece of avionics software, and note that out of the 398 clauses collected by users 16 (4%) are sound preconditions. Furthermore for all 7 functions the preconditions synthesized from players were overfitted - i.e. they omitted valid function inputs. An interesting part of Binary Fission's design is the emphasis on the players sense of community, fostered through online chat and community events.

In Circuit Bot [26] and Dynamkr [26] players help refine pointer analysis, by adding edges to a constraint graph inferred by a points-to analysis. In both game the underlying mathematical and programming structures are obscured by the intermediate presentation of graphs and the metaphor of 'energy' associated with parts of the graph. While in Circuit Bot players interact at a fine-grained level with the graph, by adding individual edges, in the follow-up game Dynamkr player instead guide an autosolver by choosing different heuristics, and thus work on much larger graph instances. To the extend of our knowledge, we have not found a published evaluation of what levels players were able to solve, or how well they compare to the automated state of the art.

FlowJam [26] and Pradox [26] is a pair of constraint solving games asking players to solve instances of MaxSAT problems encoding verification problems. While in the earlier FlowJam game players were tasked with individually toggling variable assignments, in Paradox players are guiding general solving strategies such as DPLL [38] and GSat [88] over very large SAT instances. The authors remark on the difficulty of making levels derived from real code 'fun', and report on one experiment on a piece of code from Hadoop, showing that developer effort for verification starting from a version of the code annotated from gameplay is significantly reduced compared to the unannotated version. A related earlier effort - FunSat [27] also aims to leverage human intelligence and specifically visual and spacial recognition, to solve SAT instances, by representing clauses as visual shapes. To the extent of our knowledge of the literature, we have not found an evaluation of FunSAT's performance.

PipeJam[29], is another game in which players solve constraints by working over a graph-like map. In PipeJam players are presented with a network of pipes and a set of balls of 2 different widths, and given control over the widths of some pipes/balls. Constraints in PipeJam are derived from type flows in the program, and the widths correspond to security properties encoded in the type system. In PipeJam the underlying program structure and math are again hidden from the player. An interesting aspect of PipeJam is the encoding of inter-procedural information in the shape of inter-level dependencies, which further motivates a 'world' map of all levels corresponding to the underlying program's call-graph.

In Ghost Map [26] and Hyper Space [26] players manipulate an abstracted control-flow graph to perform counterexample-guided abstraction refinement under the hood. In these games underlying program state, instructions and verification mathematics are again hidden from the player. The authors note that the time needed for the underlying solver to provide feedback became a problem in Ghost Map's gameplay and required adding more gameplay features in Hyper Space.

Finally Bjorner et al. [15] propose to crowdsource solving Constrained Horn Clause, by allowing the crowd to perform logical abductions. In their tool users were presented with raw Horn clauses, without any counterexamples or runtime traces. They authors recruited 58 participants from colleagues and friends, of which 21 submitted solutions and 2 solved the majority of benchmarks at a rate that suggest that the 2 top users had built automation to use other solvers. While strictly speaking not a gamification approach, this work is an interesting exploration of crowdsourcing for verification, especially in using humans to augment automated logical solvers.

5.3 Machine Learning Invariant Generation

Another closely related line of work uses machine learning for invariant generation. A large body of work [19, 62, 65, 94, 92, 93] has focused on inferring decision trees over primitive predicates, using positive and negative example program states obtained from an underlying solver. Garg et al. [40] argue that invariant learning approaches based on only positive and negative examples are incomplete since in the domain of inductive invariants the mechanized teacher (in this case the underlying solver) sometimes returns *inductive* counterexamples consisting of a pair of states, for which it is not known a priori whether they should be permitted or omitted. In such cases, the teacher cannot provide correct feedback to the learner, and thus the learning process get stuck.

Garg et al. [41] later propose decision tree learning work that accounts for inductive counterexamples. Learning from inductive counterexamples is also leveraged for inferring invariants over linear data structures [39], and was used in a general framework for automatically instantiating invariant inference procedures from invariant checking procedures [90]. The insight that inductive counterexamples make invariant inference a different learning domain compared to traditional machine learning is something that we also encountered during the design of our first game as mentioned in Section 6.1.4 and informed parts of the design of our second game as discussed in Section .

Finally while most works described so far inferred decision trees over expressions involving fixed features, such as linear invariants over variables in scope, recent work by Padhi et al. [83] extends this work by attempting to also learn the predicate features through counterexample guided synthesis.

We believe crowdsourced gamification and machine learning based approaches are complimentary lines of research, as data gathered from the best players in a crowdsourced setting would be a useful training set for machine learning approaches. Furthermore, as discussed in Section 6.3, we believe machine learning approaches along with other traditional automated inference can play an important role in scaling human effort in future games.

5.4 Acknowledgements

This chapter, in part, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 6

Future directions

We identify several interesting avenues for future work both in the setting of INVGAME (Section 6.1) and FLOWGAME (Section 6.2). Some avenues for work are cross-cutting — for example multiplayer gameplay has interesting applications in both INVGAME and FLOWGAME . We discuss such cross-cutting directions separately for each game.

6.1 InvGame

There are several research challenges to scaling INVGAME to larger pieces of software.

6.1.1 Arrays and the Heap

Our current version of INVGAME handles only arithmetic invariants. In the future we plan to investigate ways of incorporating arrays and the heap, for example simple numerical arrays, multi-dimensional arrays, linked-lists, and object-oriented structures. Doing so entails two challenges: (1) how to display these structures (2) how to enable players to express predicates over these structures.

For displaying, we can take inspiration from systems like Python tutor [46] which visualize the memory state of programs in various languages, mostly for educational purposes. Another source for inspiration is work done in the verification oriented game Xylem [71] on displaying one dimensional numerical arrays. Compared to those settings however, INVGAME requires a much denser visual representation. Python tutor needs to display only one state of a data structure at a time and Xylem displays at most 2 different states of a data structure. In INVGAME however, we would need to display many more different data structure states on screen, given the higher number of rows. It is a significant research challenge to find a representation that is dense enough so that humans can see patterns. If the entire heap is not viewable, some hierarchical representation might be needed that unfolds interactively.

Another approach here is to only show the relevant parts of a data structure or array. One potential marker for relevance is whether the part of the data structure was modified or read during the current evaluation.

For expressing candidate predicates over these more complex structures, we would also need to enrich the language of predicates. For example, to express that all elements of an array are zero, or that array is sorted, one would need universal quantification. In order to express properties on disjoints parts of the heap, one might need operations from separation logic [84].

It is an open question whether exposing these complex predicates to the user would be too confusing. Since separation logic involves a spacial component (splitting heaps into disjoint rejoins), it may be the case that extensive use of shapes, colors and visual metaphors may leverage the power of the human visual cortex to lower the cognitive load associated with reasoning about separating conjunctions. Another approach would be to attempt to hide separation logic entirely and use heuristics within the backend solver to split the heap.

6.1.2 Handling Implication Invariants

Some benchmarks require verification invariants involving implications of the form `A => B`. One promising approach for adding such invariants to INVGAME involves splitter predicates [91]: if we can guess `A`, then the level along with its data can be split into two sub-levels, one for `A` and one for $\neg$`A`. Although guessing a correct `A` is challenging, we found in 38 out of 39 cases that `A` appears in the program text (usually in a branch), suggesting that there are promising syntactic heuristics for finding `A`.

We explored splitting our traces based on these predicates, asking players to solve the two separate levels, and finally combining the results from the two levels in our backend using the precedent inferred from the program text. Early experiments suggest that INVGAME players are effective at solving implicative benchmarks split using syntactically inferred predicates. However, we also find that this approach is not actually leveraging the full potential of non-experts intellect. We also noted that adding the same syntactic pre-pass on top of Daikon also greatly expanded Daikon's solving ability. In an experiment using splitter predicates and Daikon as an invariant oracle instead of INVGAME , on a subset of 33 benchmarks involving implications, we found that Daikon is sufficient 75% of the time. Thus while syntactically inferred splitter predicates are a promising idea for handle implications when inferring invariants from runtime traces, it is not necessarily an idea that enables human players to use their full potential compared to automated tools.

Another related idea we did not have a chance to explore, would be to provide precedent candidates as *suggestions* to the player. Deciding when to display such suggestions (always? after a number of failed invariants?) is an open problem. Deciding how to display these suggestions is also an interesting problem — we could directly suggest an implication in the text box, or offer players a 'cutting' tool that lets them split the green rows in multiple groups and solve each as a sub-level. In essence the latter approach would allow a player's to pick a cut they think is promising (i.e. to select a splitter predicate), rather than INVGAME picking splitter predicate for players based on our own heuristics.

6.1.3 Trace Generation

Scaling INVGAME beyond limited benchmarks requires an automated way of generating run-time traces for new programs. Generating traces with good coverage requires suitable test inputs, which is an active research area [7]. While it is audacious to think we would be able to generate traces with good coverage for *any* program, we believe that for many programs we can leverage techniques such as random testing [20] and symbolic execution based testing [89, 18, 42] to get good coverage.

Another challenge beyond coverage in our domain is the *quality* of the obtained traces. Observations during the design phase suggest that people perform better when faced with traces that contain smaller, preferably non-negative numbers. Dealing with such numbers seems to lower the cognitive burden in finding patterns. Symbolic testing techniques can be adapted to produce higher quality traces by adding additional constraints for size and sign on the runtime traces, that can be iteratively relaxed. Similarly, for fuzz based trace generation, the prior distribution for test inputs can be (to some extent) biased towards human-friendly values. For example in our own fuzz tracing we biased our input values to positive numbers less than 10, which was sufficient to produce simple traces for our own evaluation.

Finally, as discussed in Section 6.2.4 the FLOWGAME visual layout allows alternative game modes aimed at discovering test inputs that achieve coverage, or even a particular logical goal (e.g. satisfy a given expression after the loop).

6.1.4 Exposing Verification Counterexamples

The goal of the INVGAME currently is not precisely aligned with the verification goal as discussed in section 3.5. One approach for bridging this gap is by exposing verification counterexamples in the INVGAME UI. As discussed in chapter 2 there are 3 VC that an invariant needs to satisfy. In the context of INVGAME , counterexamples to the 3 VC can be translated to 3 different kinds of ‘rows’.

Violations of the I-ENTRY VC arise when the candidate invariant is overfitted, and represent possible variable assignments at the loop header. As a result, these can be shown in the game UI as additional green rows, that the player must satisfy.

Violation of the I-IMPLIES VC arise when the candidate invariant is not strong enough, and represent variable assignments that should not be possible at the loop header, as they would lead to a program crash. Assuming that the program is correct, the player should be able to find an invariant that excludes those. One approach to representing those in the INVGAME setting would be as ‘red rows’ — i.e. rows that *at least one* of the candidate invariants must make false(e.g. see row Ⓒ in Figure 6.1). Note that the different behavior of green rows (all invariants must make them same color) vs red rows (at least one invariant must make them one color) may be confusing, and require additional teaching in the tutorial, or other hint mechanisms.

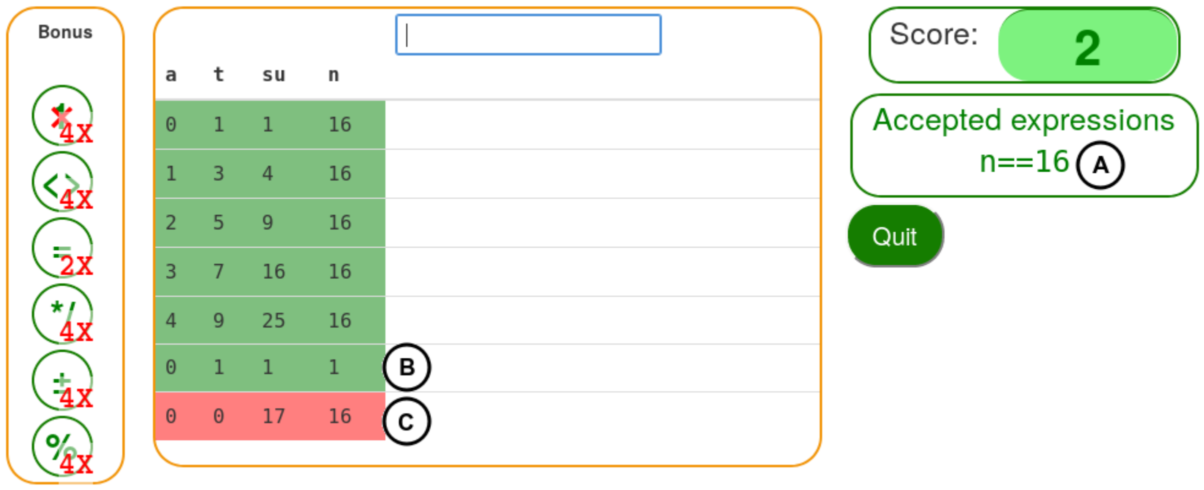
Finally, violations of the I-PRESERVE VC arise when the candidate invariant is not inductive. Counterexamples to this VC are particularly challenging, as they correspond to not one, but *two rows* — the variable assignment at the start of the loop that satisfies the invariant, and the variable assignment at the end of the loop iteration that violates it. Furthermore, we have no way a priori of knowing whether the first row is an inadmissible variable assignment(red row), or both rows are admissible assignments (both green rows). Garg et al. [40] observed a similar challenge when trying to apply traditional ML techniques to learn loop invariants using the solver as an oracle.

We attempted to model these in early mock-ups as shown in Figure 6.1. In Figure 6.1 we show the state of the game after the invariant `n=16` was just accepted (labeled Ⓐ). This invariant resulted in both a precondition and postcondition counterexamples. The precondition counterexample resulted in the new green row (labeled Ⓑ), which future invariants must satisfy. That row makes it clear that `n` is not fixed to be a constant. The postcondition counterexample resulted in the new red row Ⓒ, which shows an invalid program state that the invariant `n=16` does not prevent.

To handle inductive counterexamples we also extended the game to show two rows, with an attached ”switch” allowing players to choose which row coloring to solve for in these rows. In this gameplay mode, depending on the direction of the switch a player would either have to make both rows green, or just solve for the first row being red.

In our experiments, we found that even when limited to just the pre- and post- condition counterexamples the game quickly got confusing. For example the positive example Ⓑ in Figure 6.1 shows the case where `n=1`. While this is a valid counterexample, a singular green row does not convey any new intuition about the pattern that is sought. Even experts were confused, and sought to use disjunction to handle singular green rows separately. One possible direction is to provide several green rows as a positive counterexample, corresponding to feasible consecutive loop iterations. This approach re-enforces existing patterns for players (through the several consecutive rows) while still rejecting the current overfitted invariant.

Furthermore, we observed that red rows are not as useful early on to a player. At the start of the game most players wanted to build up a set of expressions covering the patterns that they see. With postcondition counterexample, every expression players entered early on added a new red row that distracted them from the task of seeing patterns, and provided negative feedback. In future work we believe postcondition counterexamples should be provided only after a user has built up some invariants, and care should be taken to limit the number of red rows presented.



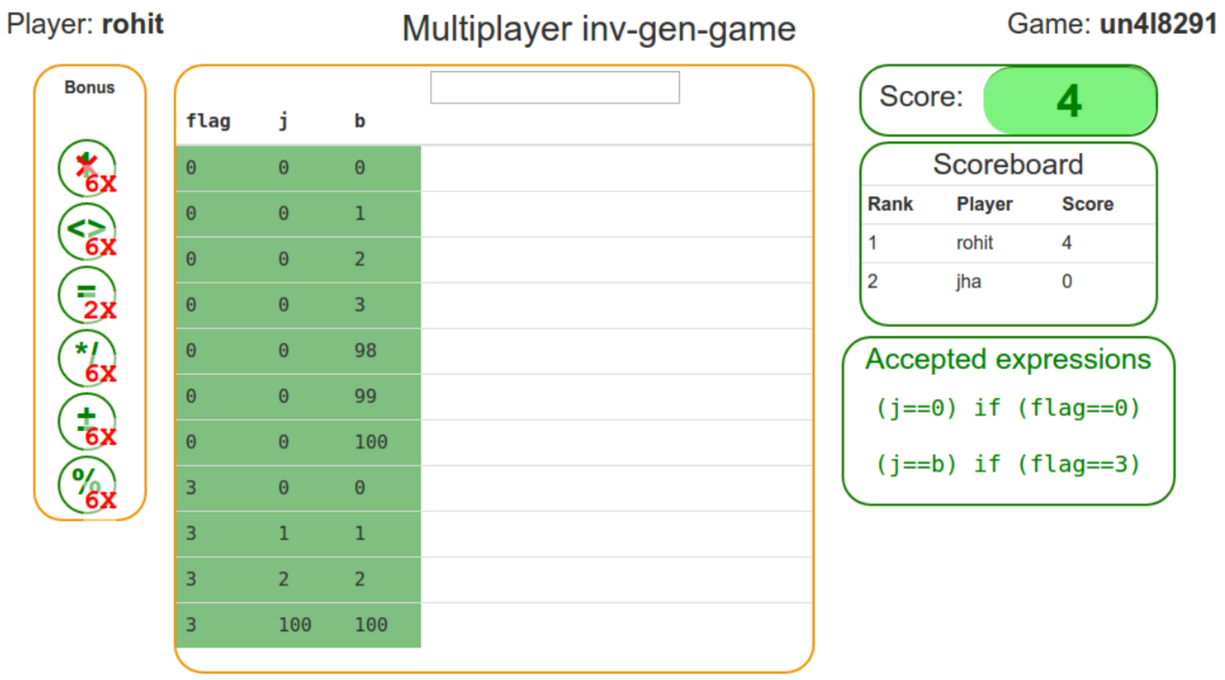
a	t	su	n
0	1	1	16
1	3	4	16
2	5	9	16
3	7	16	16
4	9	25	16
0	1	1	1
0	0	17	16

Figure 6.1. INVGAME prototype with counterexample red and green rows

Finally, the idea of giving people the option to choose how to solve an inductive counterexample using switches proved to be even more confusing to (expert) players and as such was not evaluated further. Making an informed decision whether an inductive counterexample pair is unfeasible or should be rejected, seems difficult without more knowledge of the underlying program.

6.1.5 Multiplayer

Jha [54] explored a multiplayer player version of INVGAME involving two players solving the same level in real time. Figure 6.2 shows the screen as seen by one of the players.



Player: **rohit**

Multiplayer inv-gen-game

Game: **un4l8291**

flag	j	b
0	0	0
0	0	1
0	0	2
0	0	3
0	0	98
0	0	99
0	0	100
3	0	0
3	1	1
3	2	2
3	100	100

Score: 4

Scoreboard

Rank	Player	Score
1	rohit	4
2	jha	0

Accepted expressions

```
(j==0) if (flag==0)
(j==b) if (flag==3)
```

Figure 6.2. InvGame multiplayer prototype

In this game mode, each player sees in real time the invariants discovered by their opponent. Furthermore, if a player A discovers an invariant I_1 that is strictly stronger (i.e. implies) an invariant I_2 previously discovered by player B, then player A receives a bonus. Additionally, players are shown a leader board for each level. In their experiments Jha found that players enjoyed the multiplayer version more than the single player version.

While this result is encouraging, biasing users to always seek the strongest expressions does not always result in sound invariants. One future refinement of this gameplay we would like to explore is to only broadcast the sound expressions found by users. This refinement would benefit the player base in two ways: (1) prevent individual players from being blocked from entering a correct invariant due to an overfitted expression broadcast by another player and (2) since only sound invariants are being broadcast, players will only learn beneficial patterns from their peers. This extension relies on the backend's ability to perform predicate abstraction on-line, and therefore may not always be able to provide real-time feedback to players.

Such multiplayer modality can also be combined with positive and negative counterexamples. For example after several players have competed on a level, and can not discover any new invariants, a new set of positive and negative counterexamples can be shown to them, visualized in the game world as the next *round* of the level. Upon the new counterexamples being shown, any unsound invariants disqualified by the counterexamples are removed, with only the sound invariants from the previous round retained.

6.2 FlowGame

Our experience from our experiments with FlowGame suggest several avenues for future work.

6.2.1 Immediate Design Issues

As outlined in the previous Section 4.3.3 our experiments indicate that the game is still confusing for players. User feedback has suggested many potential avenues for managing cognitive load and improving gameplay experience. We believe that the following suggestions are most pertinent to a successful gameplay and follow-up evaluation:

1. **Game tutorial with gradual learning curve.** Since the game requires teaching new players a significant amount of material, trying to fit it all in a single continuous tutorial would not be preferable. In future work we would like to build a tutorial that teaches a single concept at a time (e.g. precondition counterexamples, inductive counterexamples) intertwined with playable mini-levels that reinforce learning and maintain user engagement. As part of this work we plan on exploring better in-game metaphors for the various game elements, in response to user feedback during our experiments.

2. **Evaluation Visualization.** Providing visual evaluation of branching expressions, sink predicates and right-hand side expressions in assignment would greatly reduce the amount of math players need to do in their head to understand the behavior of game elements. Furthermore, adding small enhancement such as highlighting the variables that change values when stepping across an assignment, or the individual failing conjunct in a larger

expression would also help focus player's attention on the most pertinent information at any point.

3. **Historical Context.** The game should display the list of expressions that were tried but rejected. Upon discovering a new expression, the game should transparently re-try old expression to account for the case were a user tried a correct expression but it was rejected due to another missing expression. It would be an interesting experiment to see if showing the unsuccessful invariants tried by *other* users would be also helpful. Additionally, the game should provide a way to undo the currently attempted invariant, and go to the previous attempt.

4. **Additional user guidance from the game.** The game should identify cases where the green rows involve variables that have the same value (e.g. `n` is 7 in all rows) and automatically re-try any expression involving that value with the corresponding variable substituted. (e.g. when the user tries the expression `x+y=7` the game should transparently re-try `x+y=n`).

   The game needs to also re-write expressions involving divisions by a constant into equivalent expressions involving multiplication. As we observed during our experiments, integer division, even in cases when the division is guaranteed to be reminder free, can cause the underlying SMT solver to time out. For example one of our expert players needed multiple hints to re-write expressions such as `sum==i*(i+1)/2` into the equivalent form `2*sum==i*(i+1)`. With plan on adding syntactic heuristics for re-writing some common cases of expressions involving division.

   Finally, another future direction is to explore more advanced guidance that focuses a player's attention on particular variables. During our experiments we observed that the best performing experts in the game focused on single variables, or pairs of variables that were the least constrained. For example, an expert confronted with the level seen in Figure 4.1b, first entered the expression `(a+1)*(a+1)==su`. After observing a counterexample, they verbally reasoned that the counterexample was only possible because of the lack of constraint between `a` and `t`. This focused their attention on `a` and `t` allowing them to see the missing expression `t=2*a+1`. We believe that some of this expert behavior can be stimulated in non-experts by the game. Specifically we plan on exploring a heuristic hint approach, that computes a ‘constraint’ score for individual variables, and pairs of variables, based on the number of constraints that involve them and the types of those constraints (e.g. equalities vs. inequalities). The game can use the ‘constraint’ score to suggest which variables a user should focus on.

5. **Propagating preconditions.** In 3 of our experiments users observed that some preconditions can be immediately propagated as expressions in the middle source/sink pair, when they involve variables not modified in the program. This propagation can be easily provided by the game automatically based on the result of a standard dataflow analysis.

We believe that addressing the above improvements are necessary before FLOWGAME is ready for a wider evaluation.

6.2.2 Complex Data Types and Heaps

Similarly, to INVGAME , heaps and more complex data-types such as records, sets, maps are challenging in the FLOWGAME setting due to the need to present them in a limited visual space. For example, if a counterexample involves an array of any non-trivial length we cannot simply fit it within the current gray counter-example box. Finding succinct representations, or providing flexibility in how much of a complex data structure is displayed is a challenging problem. Furthermore, while handling scalar values requires only a high-school level understanding of algebra, complex data-structures require more specialized knowledge. Finding a good visualization that makes data structures intuitive, and building a tutorial that explains them well is another challenge in this setting.

Similarly, to InvGen we can draw upon the work done by Logas et al. [71] in the

Xylem verification game, where plant roots are used as a visual metaphor for representing one-dimensional numeric arrays. An additional inspiration would be the work done on Python tutor in representing the heap [46]. Similarly, to our planned INVGAME future work, a hierarchical representation of the heap that allows for interactive unfolding would reduce the cognitive load and help a player focus on what is important. One possible approach is to choose what parts of the heap are shown is to only the parts of an array/data structure that are read or modified in a counterexample traces. It remains as future work to evaluate whether this results in a succinct enough representation, while showing enough information for a player to complete a level.

6.2.3 Players as the Solver

Encoding heaps in the underlying solver requires using more complex theories (such as the theory of arrays [96]) which can result in the underlying solver returning `unknown` or timing out more often. The same problem can also be encountered with simpler logics such as integers or bit-vectors, when more complex non-linear invariants are used. This incompleteness of the underlying solver is likely to become an obstacle as we attempt to scale FLOWGAME to real world code.

A promising direction for future work is enabling players to take the role of the Solver in FLOWGAME . Specifically we plan on exploring a multiplayer mode of FLOWGAME , where players can take the role of the Solver and construct their own counterexamples to other player's invariants.

In this gameplay our backend solver becomes a 'first pass' at verification. If the solver can show a set of invariants sound, they are accepted and made immutable. If however the player times out or returns unknown for some invariants, those are 'conditionally' accepted, and marked as open to challenge by other players. At a later time another player can construct a concrete counterexample from any source in the game, to challenge the conditionally accepted invariants. The new counterexample can be validated against the expression associated with the corresponding source directly through substitution and evaluation — there is no need to invoke the SMT solver. The concrete counterexample is then interpreted through the CFG, hopefully reaching the disputed invariants and disproving them. Incentives such as map control, score, or even extrinsic motivations such as bounties would motivate players to find counterexamples for invariants open to challenge. Note that such gameplay can only disprove candidate invariants, not prove their soundness.

Using players as solvers in a competitive multiplayer setting is a promising research direction, and adds a new dimension to the current FLOWGAME gameplay.

6.2.4 Input Generation in FLOWGAME

Allowing players to craft their own counterexamples in FLOWGAME can also be used to create interesting test inputs. To evaluate this idea, we first need to modify game levels to remove the linked Source/Sink pair that corresponds to loop headers. In the thus modified level, the only Source is the function entry point. In this setting we can ask players to craft concrete counterexamples (corresponding to function arguments) that reach a particular icon. This task corresponds to finding function inputs that reach a particular program point.

Furthermore, by inserting additional game branches and sinks, we can server more interesting queries. For example with a carefully inserted branch and sink, we can ask a player to find inputs, such that on the `n`-th iteration of a loop, a particular condition holds.

6.2.5 Moving beyond benchmarks

FLOWGAME has so far only been evaluated on known benchmarks from literature and software verification competitions. Moving to verification ‘in the wild’ holds several challenges. First there is the need to handle more complex data structures, already mentioned in a previous section. Next is the fact that unlike benchmarks, real software lacks function pre and post condition annotations, and often holds multiple nested loops. In FLOWGAME ’s setting, this means that the soundness of invariants for a loop may depend on discovering enough invariants for an earlier loop, potentially in a completely different function! What’s worse, as more invariants are discovered, it is possible that new paths are found to a function, invalidating some of the earlier invariants found for it. In this setting, it is no longer enough to play a level derived from a single loop just once — instead gameplay must support awarding people for discovering sound loops when partial information about the program is available and re-visiting a loop multiple times, as more information is discovered about the program.

One promising approach here is leveraging different multiplayer modes. For example, one possible multiplayer mechanics is a turn-based game, where finding invariants for a given loop establishes a player's 'claim' over that loop. For example if a player A discovers an invariant `i>0` for a given loop `L`, but at a later time a player B's work leads to the verifier discovering a new path to L, along which it is possible that `i=0` then that new path is presented back to A as a challenge to their claim over the loop `L`. Player A can resolve this by finding a weaker invariant such as `i>=0`, and both players are awarded for their work. The 'map ownership' dynamic here is inspired by real time and turn-based strategy games.

6.2.6 Teaching Verification

Another interesting avenue for future work is evaluating FLOWGAME 's potential for a teaching aid for verification. We believe that the following properties of FLOWGAME make it suitable for the task:

1. Hoare-style reasoning maps closely to the gameplay
2. Counter-examples are mapped onto source-code
3. Runtime traces can server as optional hints for players

Since the game-play faithfully displays VC-gen failures it can be used as a tool for teaching software verification to CS students. Furthermore, the fact that counterexamples are mapped directly onto program variables, in a fashion similar to reversible debuggers reduces the user's cognitive load. The ability to place subject in both the shoes of the solver (by allowing them to craft counterexamples) and of the invariant oracle, allows a teacher to craft exercises both aiming at crafting inductive invariants and understanding why a given expression is not a good invariant.

6.3 Gamification and Machine Learning

As discussed in our related work (Section 5.3) there has been a significant amount of promising results in applying Machine Learning techniques [41, 83, 90] to the domain of loop invariant generation. Combining gamification with machine learning opens up an interesting research direction. At a minimum, player interactions from both INVGAME and FLOWGAME can be a useful sources of training data for machine learning work.

Another idea here is to leverage players intelligence more directly in cases where machine learning approaches get stuck. For example, Garg et al. and Padhi et al.'s work [41, 83] both involve learning decision trees over linear inequalities of features. In Garg et al.'s tool ICE-DT these invariants are the variables in scope, and additional expert provided expressions. In Padhi et al.'s Pie/InvGen tools the features are also learned through a procedure that exhaustively searches the space of expressions for features that split the space of accumulated counterexamples. Both approaches rely on the presence of high quality features for the learning to be successful. One potential direction here is to study whether useful features can be mined from the candidate expressions players find while using INVGAME or FLOWGAME .

Another research idea involves integrating machine learning approaches into the gameplay itself, as an on-screen avatar that assists a player, by suggesting likely expressions when a player gets stuck. This idea is also interesting from the point of game design, as such an avatar can learn from the user's behavior, and thus become specialized to each player. We believe such a personalized in-game character would improve player's engagement in the game. Furthermore, an ML-avatar would reduce the player's effort, as it can instantiate the 'easy' invariants, leaving the more interesting and more difficult work for the human.

6.4 Acknowledgements

This chapter, in part, is adapted from material as it appears in Bounov, Dimitar; DeRossi, Anthony; Mennarini, Massimiliano; Griswold, William G.; Lerner, Sorin. “Inferring Loop Invariants through Gamification”, Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), 2018. The dissertation author was the primary investigator and author on this paper.

Chapter 7

Conclusion

Software Verification is a promising technique for ensuring software quality that unfortunately requires a large amount of manual expert effort. This need for expert effort is an impediment to scaling Software Verification to larger bodies of code. In this dissertation we argue that Gamification and Crowdsourcing can reduce the need for expert effort by opening up some tasks in Software Verification to the wider public. In support of this thesis we present INVGAME — a numeric puzzle game allowing non-experts to infer candidate loop invariants. Our game collects candidate invariants from multiple users, and using predicate abstraction distills a sound loop invariant from the crowd's gameplay. We further evaluate INVGAME in Section 4.3 on standard benchmarks, and compare with state-of-the-art loop invariant inference tools. In our evaluation we observe that INVGAME enables non-expert players to perform on-par with state-of-the-art tools, while enjoying their experience.

We identify three limitations of INVGAME — namely (1) lack of syntactic information about the underlying program and (2) lack of feedback from the solver on why candidate invariants are insufficient and (3) misalignment of game goal and verification goal. In Chapter 4 we present the design and early experience with the follow-up game FLOWGAME , that aims to address these limitations by exposing program structure and solver counterexamples. In FLOWGAME players are exposed to the full verification problem and perform a form of Hoare-style logical reasoning. Our experiments indicate that while the game is accessible and enjoyable to some non-experts with interest in STEM, and provides enough information for experts to be highly effective at solving, it still incurs a high cognitive load on players. Feedback from our players point to several directions to improve FLOWGAME including (1) visual evaluation of expressions, (2) a gradual tutorial intermixed with gameplay and (3) more contextual information for users (e.g. rejected expressions).

We conclude by outlining several directions for future work. First, in both games one of the major obstacles to handling real-world applications is the lack of an in-game representation of arrays and complex data-structures. Such visualizations are non-trivial due to the need to find simple visual metaphors and identify the minimum parts of the data-structure needed by users, to avoid clutter.

A second research direction is leveraging multiplayer modes in both games. Beyond simply increasing player engagement multiplayer gameplay has the potential to serve the underlying verification needs. Exposing sound expression discovered by peers in INVGAME is a promising approach to focus user attention and prevent redundant work. Allowing users to challenge candidate invariants too complex for a solver to discharge is a promising idea in FLOWGAME to overcome incompleteness in the underlying SMT solver.

Finally, we discuss potential applications of the FLOWGAME interface beyond gamified verification. Alternative modes where players construct concrete counterexamples instead of expressions provide a platform to explore gamified test input generation. FLOWGAME itself can also serve as a platform for teaching verification to students.

In conclusion, we hope that the artifacts evaluated in this dissertation, and our evaluation, showing non-experts' ability to perform on-par with state of the art automated tools serve as motivation for future work on Gamification in the domain of Software Verification.

Appendix A

FLOWGAME User Study Data

A.1 Sound Invariants

In Table A.1 we present the sound invariants found for each level, by each of our subjects during our experiments. The sound invariants are mostly the solutions expected by the authors, with 2 notable exceptions.

1. On level 5 Expert 2 used the implicative invariant $j >= x ==> x == j$ in place of the more commonly found $j <= x$
2. On level 6 Expert 6 used the equivalent expression $(t+1)*(t+1) == su*4$ in place of the more commonly found $(a+1)*(a+1) == su$.

These two instances are similar to two cases in the original INVGAME evaluation of players finding original solutions that surprised even the authors. This creativity exemplifies the power of diversity of human cognition.

Table A.1. FLOWGAME User Study Sound Invariants

Level	Player	Invariants
1	Non-Expert 1	$x < 101, x >= 0$
1	Non-Expert 2	$x >= 0$
1	Non-Expert 3	$x <= 100 \&\& x >= 0, x <= 100$

1	Non-Expert 4	$x >= 0 \&\& x <= 100$
1	Non-Expert 5	$x >= 0$
1	Non-Expert 6	$x <= 100, x >= 0$
1	Expert 1	$x >= 0$
1	Expert 2	$x <= 100 \&\& x >= 0, x <= 100$
1	Expert 3	$x <= 100, x >= 0$
1	Expert 4	$x >= 0$
1	Expert 5	$x >= 0$
1	Expert 6	$x >= 0$
2	Non-Expert 2	$x+1024 >= y-1025, y >= x, y+1 >= x-1, x == y, y+1 >= x, x+1 >= y-1$
2	Non-Expert 3	$x == y$
2	Non-Expert 4	$y == x$
2	Non-Expert 5	$x == y$
2	Non-Expert 6	$y == x$
2	Expert 1	$x == y$
2	Expert 2	$y == x$
2	Expert 3	$x == y$
2	Expert 4	$x == y$
2	Expert 5	$x == y$
2	Expert 6	$x == y$
3	Non-Expert 3	$n >= i, n >= i \&\& sum >= 0 \&\& i >= 0$
3	Non-Expert 4	$n >= 0, sum >= 0 \&\& i >= 0$
3	Non-Expert 5	$sum >= 0 \&\& i >= 0$
3	Non-Expert 6	$i >= 0 \&\& sum >= 0$

3	Expert 1	$n >= 0 \&\& sum >= 0 \&\& i >= 0$
3	Expert 2	$i <= n \&\& sum >= 0 \&\& i >= 0, i <= n$
3	Expert 3	$sum >= 0, i <= n, i >= 0$
3	Expert 4	$i <= n, i >= 0, sum >= 0$
3	Expert 5	$n >= 0, i >= 0, sum >= 0$
3	Expert 6	$i <= n, i >= 0, sum >= 0$
4	Non-Expert 5	$y >= 0 \&\& x + y == n, y >= 0, x >= 0$
4	Non-Expert 6	$x + y == n, n >= 0 \&\& x >= 0, y >= 0, n >= x$
4	Expert 1	$y >= 0, n - x == y, n > 0, y >= 0 \&\& n > 0 \&\& x >= 0$
4	Expert 2	$x >= 0, n > 0, x + y == n, n > 0 \&\& x >= 0, x > 0 ==> n > 0, x < 0 ==> y == n$
4	Expert 3	$x <= n, n > 0, x >= 0, n - x == y, y >= 0$
4	Expert 4	$x >= 0 \&\& y >= 0 \&\& x + y == n$
4	Expert 5	$x >= 0, x + y == n$
4	Expert 6	$y + x == n, x <= n, x >= 0$
5	Non-Expert 6	$j * y == s, j >= 0, x == 0 ==> s >= 0, j <= x$
5	Expert 1	$s * y >= 0, j >= 0 \&\& j <= x, s == j * y$
5	Expert 2	$x >= 0, y > 0 ==> s >= 0, s == j * y, j >= x ==> x == j, j >= 0$
5	Expert 3	$x >= 0, j <= x, j * y == s$
5	Expert 4	$s == j * y, j <= x, x >= 0, j >= 0$
5	Expert 5	$s == j * y, j <= x$
5	Expert 6	$j >= 0, x >= j, s == j * y$
6	Expert 3	$(a + 1) * (a + 1) == su, su >= 1, (a + 1) * (a + 1) <= su, n > 0, t >= 1, a >= 0, 2 * a + 1 == t, a < su$

6	Expert 4	$a >= 0 \&\& t == 2*a+1, su == (a+1)*(a+1)$
6	Expert 5	$(a+1)*(a+1) >= su, t >= 1, t == 2*a+1, n > 0, n >= 0, su == (a+1)*(a+1)$
6	Expert 6	$(t+1)*(t+1) == su*4, t == 2*a+1$
7	Expert 4	$i <= n+1, sum*2 == (i-1)*i$
7	Expert 5	$2*sum == i*(i-1), i <= n+1$
7	Expert 6	$2*sum == (i-1)*i, i > 0, i <= n+1, sum >= 0$
8	Expert 4	$i <= n+1 \&\& s*2 == i*(i-1))$
8	Expert 5	$2*s == i*(i-1), i <= n+1$
8	Expert 6	$j >= 0, s >= i-1, j <= n+1, i*(i-1) == s*2, i <= n+1, i >= 1$
9	Expert 4	$a*2 == i*(i+1), c == i*i*i, i <= 10$
9	Expert 5	$c == i*i*i, 2*a == i*(i+1), a >= 0, i >= 0, c >= 0$
9	Expert 6	$i*(i+1) == a*2, c == i*i*i$
10	Expert 4	$y > 0 ==> y > x, x >= 0 ==> y > 0, y > 0 ==> y >= x$
10	Expert 5	$x >= 0 ==> y > 0$
10	Expert 6	$x < 0 \| \| y > 0$
11	Expert 5	$j <= k, k >= 0, i == 2*k*j, j >= 0, i >= 0$
11	Expert 6	$j >= 0, j*k*2 == i, i >= 0, j <= k$

A.2 User Feedback

In Section 4.3.3 we summarize the key patterns emerging in user's explicit feedback and the authors observations during experiments. As some of the patterns motivation is based on the author's interpretation of the subjects feedback, for completeness we provide the relevant subjects feedback below.

Table A.2. Player feedback

Player	Fun	Challenging	Detailed Comments
Non-Expert 1	1	5	**Liked:** **Disliked:** - game doesn't prompt players what to do - game doesn't show the list of possible operators/expressions **Confused by:** **Other Comments:** -Too many options (for operators) in the beginning -Too many metaphors (referring to cities/trucks), confusing metaphors -Wish for a drag-and-drop interface for discovering possible expressions
Non-Expert 2	1	4	**Liked:** -It was a challenging game **Disliked:** -not enough eye-candy -steep learning curve -levels look the same **Confused by:** - thought one has to find the answer all at once as a single conjunct **Other Comments:** -would not play again

Non-Expert 3	1	5	**Liked:** -having to solve for expressions **Disliked:** -wished they had scratch space for comments -naked math made it intimidating **Confused by:** - expressions from top source didn't directly apply to intermediate source/sink **Other Comments:**
Non-Expert 4	1	5	**Liked:** **Disliked:** -tutorial was very long **Confused by:** -branches were very confusing **Other Comments:**

Non-Expert 5	3	5	**Liked:** -"good brain challenge" -liked having hints (referring to verbal hints given by author) **Disliked:** **Confused by:** -objective of the game was confusing **Other Comments:** - Asked to play another level after experiment finished
Non-Expert 6	3.5	4	**Liked:** - Logic part **Disliked:** - Previous wrong attempts are not shown (no history context) - Orbs covers up other stuff on screen **Confused by:** - didnt realize full value of green rows initially - variable names switching from level to level was confusing - direction of orbs through branches was confusing. **Other Comments:** - Honestly would want to play again" - "The more I do it, the easier it gets."

Expert 1	2	5	**Liked:** - Builtin calculator (though they didnt use it as much) - Felt the green rows are sometimes useful - especially for pattern recognition - Assignments order **Disliked:** -orbs overlaying/obstructing other text -game needs more graphics to be more fun **Confused by:** -Multiple cities metaphor was confusing -Confused whether second source satisfies the constraints of the first source **Other comments:** - Game not very fun, however it is engaging - Player wanted to gradually build a solution, which was not always possible - Once they got used to all tools at their disposal, felt levels got easier

Expert 2	3.5	5	**Liked:** -Animation of bubble going through assignment rows **Disliked:** - Cant undo the current expression and counterexample - Game seems too complex, too steep a learning curve **Confused by:** -Confused by role of green rows. Initially thought of them as a separate problem **Other Comments:**
Expert 3	3	4	**Liked:** **Disliked:** - Really easy for my flow to be broken. - Easy to get lost/forget where I am - Want values for counterexamples to appear in green row table, in order to reduce the amount of computation done by player **Confused by:** **Other Comments:** - ”I want more hand-holding animations when an orbs appears” - Hard for someone with short attention span

Expert 4	4	4	**Liked:** - Visuals - red/green path distinction **Disliked:** - would like counterexamples added as additional green rows **Confused by:** - metaphors where unnecessary/cumbersome. **Other Comments:** - would continue playing

Expert 5	4	4	**Liked:** - game provides a ton of info - Visualization of a trace - Visualization of control flow - Sense of playing against an adversary - This is the first game (in the proof game series) that feels like a game" **Disliked:** - correct expressions can get rejected early on due to missing other expressions - Lack of pen&paper/scratch space - Metaphor for cities/roads **Confused by:** **Other Comments:**

Expert 6	3	5	**Liked:** -Combination of green rows and paths -Characters in the game (adversary - HAL) -Explanation metaphor - roads, cities, grinder cities **Disliked:** - ”not enough small victories to keep me going” **Confused by:** -Under what conditions are the invariants ”going green” (i.e. accepted) **Other Comments:** - Feel proud of finishing these levels

A.3 Hints

As described in Section 4.3.4 hints provided to the users are a threat to the generality of the design issues outlined during our experiments. Hints were split into 2 categories - (1) hints that don't refer to any particular expression or variable, and just substitute missing UI/UX features and (2) hints that directly mention variables, operators or expressions. The former part of those hints (21 hints) is presented in Table 4.7 in Section 4.3.4. The latter part (10 hints) are presented below.

Table A.3. Hints mentioning concrete variables, operators or expressions

Level	Subject	Hint
3	Non-Expert 3	In response to subject question, told them implication is not needed.
1	Non-Expert 4	User entered invariant `x>=0 || x<=100`. I asked them if they meant `&&`.
2	Non-Expert 4	Asked subject if they can write something to make sum positive.
3	Non-Expert 5	Told subject the missing invariant was `x>=0`.
3	Non-Expert 6	Asked subject to look at sum of `x` and `y` in green rows.
2	Expert 2	Asked subject ”Why do you think `sum<=6` is inductive?”.
5	Experts 1 and 2	Asked subjects to look at `t` in green rows.
5	Expert 3	Asked subject to consider other operators than inequalities.
5	Expert 3	Asked subject to consider if the postcondition holds on just the last iteration or every iteration.

Bibliography

[1] Competition on software verification (sv-comp) benchmarks. https://sv-comp.sosy-lab.org/2016/, 2016.

[2] The hol interactive theorem prover. https://hol-theorem-prover.org/, 2017.

[3] Isabelle. https://www.cl.cam.ac.uk/research/hvg/Isabelle/, 2017.

[4] Some programs that need polynomial invariants in order to be verified. http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html, 2017.

[5] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit. *Black Hat USA*, 2007.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[7] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[8] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213, 2001.

[9] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[11] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[13] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[14] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.

[15] Nikolaj Bjørner, Dejan Jovanovic, Tancrède Lepoint, Philipp Rümmer, and Martin Schäf. Abduction by non-experts. In Thomas Eiter, David Sands, Geoff Sutcliffe, and Andrei Voronkov, editors, *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017*, volume 1 of *Kalpa Publications in Computing*. EasyChair, 2017.

[16] Daren C Brabham. Moving the crowd at threadless: Motivations for participation in a crowdsourcing application. *Information, Communication & Society*, 13(8):1122–1145, 2010.

[17] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 325–338, 2008.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[19] Yu-Fang Chen and Bow-Yaw Wang. Learning boolean functions incrementally. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2012.

[20] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.

[21] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[22] Seth Cooper, Adrien Treuille, Janos Barbero, Andrew Leaver-Fay, Kathleen Tuite, Firas Khatib, Alex Cho Snyder, Michael Beenen, David Salesin, David Baker, and Zoran Popovic. The challenge of designing scientific discovery games. In *International Conference on the Foundations of Digital Games, FDG '10, Pacific Grove, CA, USA, June 19-21, 2010*, pages 40–47, 2010.

[23] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282, 1979.

[24] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[26] Drew Dean, Sean Gaurino, Leonard Eusebi, Andrew Keplinger, Tim Pavlik, Ronald Watro, Aaron Cammarata, John Murray, Kelly McLaughlin, John Cheng, et al. Lessons learned in game development for crowdsourced software formal verification. *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, 2015.

[27] Andrew DeOrio and Valeria Bertacco. Human computing for eda. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 621–622. IEEE, 2009.

[28] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popovic. Verification games: making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 42–49, 2012.

[29] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popovic. Verification games: making verification fun. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 42–49. ACM, 2012.

[30] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 443–456, 2013.

[31] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 443–456. ACM, 2013.

[32] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 449–458, 2000.

[33] Daniel Fava, Dan Shapiro, Joseph C. Osborn, Martin Schäf, and E. James Whitehead Jr. Crowdsourcing program preconditions via a classification game. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1086–1096, 2016.

[34] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[35] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[36] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 191–202. ACM, 2002.

[37] Forbes. How hackers broke equifax: Exploiting a patchable vulnerability. https://www.forbes.com/sites/thomasbrewster/2017/09/14/equifax-hack-the-result-of-patched-vulnerability/#7b45bbf55cda, 2017.

[38] Malay Ganai and Aarti Gupta. *SAT-based scalable formal verification solutions*. Springer, 2007.

[39] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 813–829. Springer, 2013.

[40] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 69–87, 2014.

[41] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512, 2016.

[42] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[43] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[44] The Guardian. '$300m in cryptocurrency' accidentally lost forever due to bug. https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether, 2017.

[45] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 281–292. ACM, 2008.

[46] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[47] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 634–640, 2009.

[48] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[49] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244. ACM, 2004.

[50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10,*

2003, Proceedings, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[51] Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[52] Business Insider. Hackers have stolen $32 million in ethereum in the second heist this week. http://www.businessinsider.com/report-hackers-stole-32-million-in-ethereum-after-a-parity-breach-2017-7, 2017.

[53] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 113–128, 2012.

[54] Rohit Jha. *Synthesizing Loop Invariants through a Multiplayer Game*. PhD thesis, 2017. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2018-03-24.

[55] David R Karger, Sewoong Oh, and Devavrat Shah. Efficient crowdsourcing for multi-class labeling. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):81–92, 2013.

[56] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[57] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap–the formally verified optimizing compiler compcert. In *SSS'17: Safety-critical Systems Symposium 2017*, pages 163–180. CreateSpace, 2017.

[58] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Type-based data structure verification. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315. ACM, 2009.

[59] Firas Khatib, Frank DiMaio, Seth Cooper, Maciej Kazmierczyk, Miroslaw Gilski, Szymon Krzywda, Helena Zabranska, Iva Pichova, James Thompson, Zoran Popović, et al. Crystal structure of a monomeric retroviral protease solved by protein folding game players. *Nature structural & molecular biology*, 18(10):1175, 2011.

[60] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM.

[61] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[62] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2010.

[63] Phil Koopman. A case study of toyota unintended acceleration and software safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf, 2014.

[64] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 447–462, 2010.

[65] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[66] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[67] Edith L. M. Law, Luis von Ahn, Roger B. Dannenberg, and Mike Crawford. Tagatune: A game for music and sound annotation. In *Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR 2007, Vienna, Austria, September 23-27, 2007*, pages 361–364, 2007.

[68] Rustan Leino. This is boogie 2. Microsoft Research, June 2008.

[69] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54, 2006.

[70] Heather Logas, Richard Vallejos, Joseph C. Osborn, Kate Compton, and Jim Whitehead. Visualizing loops and data structures in xylem: The code of plants. In *4th IEEE/ACM International Workshop on Games and Software Engineering, GAS 2015, Florence, Italy, May 18, 2015*, pages 50–56, 2015.

[71] Heather Logas, Richard Vallejos, Joseph C. Osborn, Kate Compton, and Jim Whitehead. Visualizing loops and data structures in xylem: The code of plants. In *4th IEEE/ACM International Workshop on Games and Software Engineering, GAS 2015, Florence, Italy, May 18, 2015*, pages 50–56, 2015.

[72] Heather Logas, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Daniel G. Shapiro, John Murray, Kate Compton, Joseph C. Osborn, Orlando Salvatore, Zhongpeng Lin, Huascar Sanchez, Michael Shavlovsky, Daniel Cetina, Shayne Clementi, and Chris

Lewis. Software verification games: Designing xylem, the code of plants. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014, Liberty of the Seas, Caribbean, April 3-7, 2014.*, 2014.

[73] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 136–146, 2009.

[74] J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 237–248, 2010.

[75] Eduard Marin, Dave Singelée, Flavio D. Garcia, Tom Chothia, Rik Willems, and Bart Preneel. On the (in)security of the latest generation implantable cardiac defibrillators and how to secure them. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 226–236, 2016.

[76] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[77] Kenneth L. McMillan. Lazy annotation for program testing and verification. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[78] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[79] MITRES.org. Cve-2017-9805. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9805, 2017.

[80] Don Norman. *The Design of Everyday Things*. 1988.

[81] Aleph One. Smashing the stack for fun and profit. http://www.businessinsider.com/report-hackers-stole-32-million-in-ethereum-after-a-parity-breach-2017-7, 1996.

[82] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[83] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 42–56, 2016.

[84] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[85] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[86] V. Abella S. Andersen. Data execution prevention: Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[87] Katie Seaborn and Deborah I. Fels. Gamification in theory and action: A survey. *International Journal of Human-Computer Studies*, 74(Supplement C):14 – 31, 2015.

[88] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446. Citeseer, 1992.

[89] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[90] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014.

[91] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 703–719, 2011.

[92] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 574–592. Springer, 2013.

[93] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June*

20-22, 2013. Proceedings, volume 7935 of *Lecture Notes in Computer Science*, pages 388–411. Springer, 2013.

[94] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2012.

[95] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 391–406. ACM, 2013.

[96] Aaron Stump, Clark W Barrett, David L Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 29–37. IEEE, 2001.

[97] Daniel Sui, Sarah Elwood, and Michael Goodchild. *Crowdsourcing geographic knowledge: volunteered geographic information (VGI) in theory and practice*. Springer Science & Business Media, 2012.

[98] Synopsis. Coverity static analysis. https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf, 2018.

[99] Rajat Tandon and Rajika Tandon. Article: Algorithm to compute cubes of 1st ”n” natural numbers using single multiplication per iteration. *International Journal of Computer Applications*, 101(15):6–9, September 2014.

[100] LLVM Team. http://clang.llvm.org/docs/safestack.html. http://clang.llvm.org/docs/SafeStack.html, 2014.

[101] PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[102] The Coq Development Team. The coq proof assistant reference manual. https://coq.inria.fr/refman/, 2017.

[103] New York Times. A hacking of more than $50 million dashes hopes in the world of virtual currency. https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html, 2017.

[104] Ramine Tinati, Markus Luczak-Roesch, Elena Simperl, and Wendy Hall. An investigation of player motivations in eyewire, a gamified citizen science project. *Computers in Human Behavior*, 73(Supplement C):527 – 540, 2017.

[105] Aditya Vempaty, Lav R Varshney, and Pramod K Varshney. Reliable crowdsourcing for multi-class labeling using coding theory. *IEEE Journal of Selected Topics in Signal Processing*, 8(4):667–679, 2014.

[106] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, pages 319–326, 2004.

[107] Luis von Ahn, Mihir Kedia, and Manuel Blum. Verbosity: a game for collecting common-sense facts. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, pages 75–78, 2006.

[108] Luis von Ahn, Ruoran Liu, and Manuel Blum. Peekaboom: a game for locating objects in images. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, pages 55–64, 2006.

[109] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 205–214, 2007.

[110] Johan Wagemans, James Elder, Michael Kubovy, Stephen Palmer, Mary Peterson, Manish Singh, and Rdiger von der Heydt. A century of gestalt psychology in visual perception: I. perceptual grouping and figure-ground organization. *Psychological Bulletin*, 138(06):1172–1217, 2012.

[111] Johan Wagemans, Jacob Feldman, Sergei Gepshtein, Ruth Kimchi, James R. Pomerantz, and Pater A. van der Helm. A century of gestalt psychology in visual perception: Ii. conceptual and theoretical foundations. *Psychological Bulletin*, 138(07):1218–1252, 2012.

[112] Ronald Watro, Kerry Moffitt, Talib Hussain, Daniel Wyschogrod, John Ostwald, Derrick Kong, Clint Bowers, Eric Church, Joshua Guttman, and Qinsi Wang. Ghost map: Proving software correctness using games. *SECURWARE 2014*, 223, 2014.

[113] Wikipedia. Pwn2own. 2018.

[114] Yichen Xie and Alexander Aiken. Saturn: A sat-based tool for bug detection. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 139–143, 2005.

[115] G. Zichermann. Intrinsic and extrinsic motivation in gamification. http://www.gamification.co/2011/10/27/intrinsic-and-extrinsic-motivation-in-gamification/, 2011.