Verification Games: Making Verification Fun

Werner Dietl Stephanie Dietzel Michael D. Ernst Nathaniel Mote Brian Walker
Programming Languages & Software Engineering Group, University of Washington
{wmdietl,sdietzel,mernst,nmote,bdwalker}@cs.washington.edu

Seth Cooper Timothy Pavlik Zoran Popović
Center for Game Science, University of Washington
{scooper,pavlik,zoran}@cs.washington.edu

ABSTRACT

Program verification is the only way to be certain that a given piece of software is free of (certain types of) errors — errors that could otherwise disrupt operations in the field. To date, formal verification has been done by specially-trained engineers. Labor costs have heretofore made formal verification too costly to apply beyond small, critical software components.

Our goal is to make verification more cost-effective by reducing the skill set required for program verification and increasing the pool of people capable of performing program verification. Our approach is to transform the verification task (a program and a goal property) into a visual puzzle task — a game — that gets solved by people. The solution of the puzzle is then translated back into a proof of correctness. The puzzle is engaging and intuitive enough that ordinary people can through game-play become experts.

This paper presents a status report on the Verification Games project and our Pipe Jam prototype game.

Categories and Subject Descriptors

D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification—*Correctness proofs*; F.3.1 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; K.8.0 [**PERSONAL COMPUTING**]: General—*Games*

General Terms

Verification, Economics

Keywords

crowdsourcing, human, verification, games, type system

1. INTRODUCTION

Our aim is to change the way that software is verified, to enable inexpensive formal verification. Current approaches that rely on testing are incomplete. Current approaches that rely on manual verification by skilled users are extremely expensive. None of these approaches is capable of keeping up with the rate of software production.

FTfJP'12, June 12, 2012, Beijing, China

Our approach is to remap the problem into a more accessible form, and use an engaging game to develop a significantly larger number of experts capable of solving verification problems in a remapped domain. Instead of relying on software engineers, we will develop a new skilled verification workforce, and use crowd-sourcing on a much more general audience of people who enjoy the challenge of playing a game.

The availability of inexpensive formal verification could change the economics of software verification and validation, making formally-correct software more cost-effective and thus more widespread, and leading to systems that are more reliable and robust. These benefits would extend throughout the software development lifecycle, because formal specifications make code maintenance easier and cheaper.

We have built a prototype end-to-end system that takes as input

- an arbitrary Java program and
- a security (or other) property, expressed as a type system

and produces as output

- a proof of correctness that the program satisfies the security property, or
- a specific source location where the program violates the property — that is, where the program may be insecure.

This system automatically converts the program and property into a game that can be played by people with no knowledge of or training in computing. When the player finishes a game challenge, the final configuration of board elements can be translated into a proof of correctness for the original program. More precisely, the board configuration corresponds to a set of type annotations that can be checked by a type-checker. The human player can be viewed as doing type inference.

Because the game is designed to encode the same constraint system as the type system, if the player solves a level of the game, then the corresponding annotated program will type-check. The type-checker gives a sound guarantee that the type annotations are correct and the program is secure with respect to the specific security property.

In many cases, the program will *not* type-check — equivalently, the game cannot be solved. There are two main reasons for this.

1. The program is not secure — it contains a vulnerability.
2. The program is secure, but the reason that it is secure is beyond the reasoning abilities of the underlying verification technology. Every verification system — type checking, theorem proving, model checking, abstract interpretation, etc. — issues warnings about some secure programs.

There is no *a priori* way to know which of the two reasons lies behind a verification failure — equivalently, an un-solvable game level.

A fielded version of our system would present each verification failure to a human verification expert. The human examines the source code and the type error, then determines which of two fixes to apply: change the code to correct the bug, or add an assertion/assumption indicating that the verification system should not issue a warning at this particular location. As with any other conservative type checker, a type error indicates an inconsistency between parts of the program, but does not prove that an error exists, nor does it provide a defect trace or test inputs that trigger a property violation at run time.

Even when players (who have become game experts over time) cannot fully verify a program, partial verification has great value: it permits the valuable time of highly-skilled engineers to be focused on the most important and difficult-to-verify parts of the program.

1.1 Types for program verification

We use pluggable type-checking as the underlying verification technology. Type systems are the shining success of program verification. Types are used on a regular basis by ordinary programmers who use them to verify the absence of certain types of errors. Programmers write a partial specification in the form of type annotations, then a type-checker detects potential errors. No other formal verification approach has seen such widespread adoption.

Pluggable type-checking provides an excellent foundation from which to improve upon the limitations of current verification approaches. Type-checking is *sound*, so it offers a guarantee. Type-checking is *expressive*: it is formally equivalent [33, 27, 13, 14, 17, 28] to any other verification technology, including model-checking and theorem-proving. Type-checking is already *familiar* to every developer, and it fits well into the development process. Type-checking is *modular*: it can be done on each component of a system individually, making it much more scalable. Related to the previous two points, type-checking is *transparent*, which makes it easy to use: its error messages tend to be clear, and a small change to the application does not make a large, non-local change to the type-checking results. This is relevant both to game players who will see the error messages in a different guise, and to verification experts who will take over when the game players get stuck. Type-checking is *extensible*: even a non-expert can create a new, custom type-checker that verifies a domain-specific property of interest; a type system that is extensible in this way is called "pluggable" [7].

The downside of type systems is that they tend to be less expressive than certain other verification techniques. A type system offers partial verification — it is designed to detect a certain class of errors, and the program may still be subject to other errors. This is the right way to start, because full formal verification remains impractical for realistic software. Furthermore, type-checkers are powerful and can check a wide range of important properties. Our system is built upon the Checker Framework [8], which has found hundreds of errors in millions of lines of code [30, 19]. It can be easily adapted to new verification problems and comes with type-checkers that prevent errors due to: null pointers; initialization; map keys; equality tests and interning; incorrect mutation (side effects); concurrency and locking; fake enumerations; information flow (trust and security); aliasing; regular expression syntax; property files; internationalization; the string representation of data; and many more. We have mapped the CWE/SANS Top 25 Most Dangerous Software Errors[1] to type systems.

Our system maps source code's typeflow properties into a network of pipes. Pipe widths, which are controlled by the player, directly map to type annotations in programs that can be mechanically checked and provide a proof of partial correctness. The constraints and relationships among game elements are representations of the constraints on program types and the relationships among program variables. By playing the game, the programmer is effectively choosing a type for each variable in the program. This is valuable because general type inference with precise error messages remains an unsolved problem that can benefit from crowdsourcing.

We believe that humans may have an edge over automated systems in certain situations, notably when the program is not verifiable. A program is unverifiable when it has a bug, or when it is correct for reasons that are beyond the power of the verification system. In either case, the failure is communicated back to an expert, who only considers cases that the crowd of players cannot resolve.

The remainder of this paper is organized as follows. Section 2 explains the Pipe Jam game mechanics. Section 3 discusses the translation of a program and property into a Pipe Jam game. Finally, Section 4 discusses related work and Section 5 concludes.

2. THE PIPE JAM GAME

Figures 1, 2, and 3 show screenshots of the Pipe Jam game. The screenshots show Pipe Jam being used to verify that a part of our system itself is free of null pointer errors. We now explain the figures.

Pipe Jam presents the game player a set of related ball-and-pipe puzzles. Each pipe is either narrow or wide, and the player is allowed to control the width of some pipes. Each ball is either small or large. A small ball can roll down any pipe without obstruction. A large ball can roll down a wide pipe, but gets stuck if it ever tries to roll down a narrow pipe. The player's goal is to ensure that the balls never get stuck.

A player might try to make all pipes wide, but this does not work because some pipes are narrow and cannot be adjusted. (This is sometimes represented as an obstruction, or pinch point, that only permits a small ball; see Figures 1, 2, and 3 for examples of pinch points.) Likewise, some balls are large and cannot be adjusted; see Figures 2 and 3 for examples of balls (atop gray pipes) that are fixed to be large. The puzzle is solved when the player has chosen widths that are consistent with all the constraints. The constraints include both fixed pipe sizes and which pipes flow into one another.

The basic idea behind Pipe Jam is simple and lends itself to quick learning and enjoyable play. We now explain some additional game mechanics that add interest and challenge. (These game mechanics are much easier experienced than textually described. See a video at the project homepage[2].)

Boards A Pipe Jam game is divided into boards, levels, and worlds. A board is a single network of pipes. Examples appear in Figures 2 and 3.

Levels A level consists of a set of boards. The left side of Figure 1 shows a level being played. A level is solved when all of the boards in it are solved. As explained below ("linked pipes"), actions on one board can affect another board: the player must solve all of them simultaneously, not one after the other independently.

Worlds A world consists of multiple levels. A player proceeds through the world, solving each level one by one. As shown in the world map of Figure 1, the levels depend on one another, which constrains the order in which the player may solve them. It may sometimes be necessary for a player to backtrack to a previous level to find a better solution for it, in order to solve a subsequent level. "Embedded networks" below explains how boards and levels depend on one another.

[1] http://cwe.mitre.org/top25/?2011

[2] http://www.cs.washington.edu/verigames

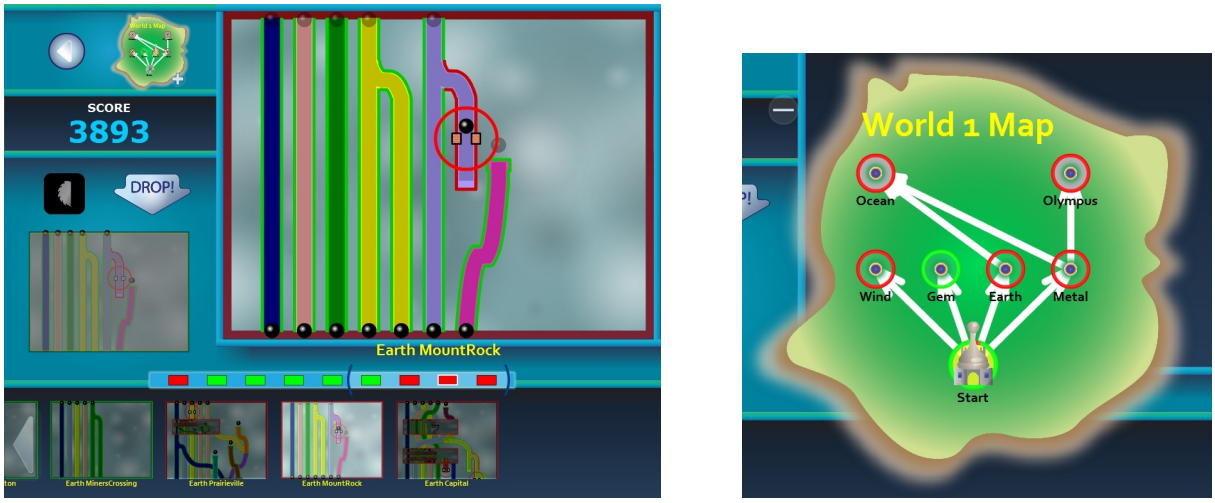

Figure 1: A view of the Pipe Jam verification game being played. The underlying verification problem is to prove that a part of our system itself is free of null pointer errors.
The left screenshot shows the player's view of the game. In the upper right is the board that the player is currently working on (named "Earth MountRock"). The bottom portion of the screen shows thumbnails of the other boards in the current level, together with a concise summary of which boards are solved and which are unsolved (the row of red and green rectangles). The thumbnail at center left is for panning, which is not needed in this example. Above it are controls for adding a buzzsaw and for viewing an animation. Above those are the current score; a back button, the collapsed world map.
The right screenshot shows the world map in its expanded state. The world map shows named levels and the dependencies between them, and permits the player to navigate to a new level.

Linked pipes All the pipes of a given color (in a board or an entire level) have the same width. (An example is the two orange pipes in Figure 2.) When the player changes the width of one of these pipes, then it also changes the width of all the other pipes of that color. When the player changes the width of a given pipe, that change may solve one board on a level but unsolve a different, previously-solved board on the same level. The player's challenge is to find a set of pipe widths that simultaneously solve the entire level.

Embedded networks One network can contain another network as a subcomponent. For example, in Figure 2, the left board "Metal SmeltCity" contains another board "Metal SteelTowne" as a subcomponent. When balls roll through SmeltCity to the subcomponent, they next traverse the subcomponent to its end before returning to SmeltCity. Changes to the width of the input or output pipes of the subcomponent, Metal SteelTowne, can cause collisions in SmeltCity. This is another example of relationships among boards that the player must satisfy. The embedded network may come from the same level, or any level that does not follow the current one on the world map. Mutually dependent levels can be processed in either order. In the worst case, a player may need to visit each one multiple times.

Buzzsaws: Exceptions to the laws of physics Not every Pipe Jam game is solvable. It is possible that every combination of pipe widths yields at least one collision. To let players proceed in this case, Pipe Jam contains a "cheat" — the buzzsaw — that any player is allowed to use at any time. A buzzsaw converts any ball that passes by it to a small ball, which can then fit through any pipe. See Figure 3 for an example. It would be possible to solve any level using sufficiently many buzzsaws, without changing the sizes of any pipes. However, placing a buzzsaw costs a large number of points, so players desire to use as few buzzsaws as possible.

Scoring A given board, level, or world may have multiple solutions. Each configuration of pipe widths earns the player a different number of points. Criteria in determining the score include the number of unsolved levels in a world, unsolved boards in a level, and collisions in a board; the number of buzzsaws used; and widths of pipes (more points are earned for large pipes entering a board and narrow pipes exiting a board). These criteria favor solving the underlying verification problem; avoiding assumptions or type loopholes in the proof; and creating type annotations that favor reuse of components.

The natural modularity of object-oriented programs, which are composed of classes that are composed of methods, enables a casual gamer to do a little bit of work and then later come back to the game. However, it is possible for interrelations to cross levels — because of global variables that are used in multiple classes, or because of calls from one class to another.

In Figure 1, the player has not yet solved the game. The green or red outline color of each board/thumbnail indicates whether it is solved. At any time, the player can animate the networks, making balls flow along them and seeing the result, but the colored borders give immediate visual feedback. We plan to support two different types of animations: a type-theoretic one that reflects what the underlying program verification tool can establish (this is already implemented), and an execution-based one that illustrates what the program actually does on some set of real executions. The latter is more precise but incomplete [21], so the two approaches are complementary.

3. MAPPING A PROGRAM AND A PROPERTY INTO THE PIPE JAM GAME

We now explain, at a high level, our approach for translating a verification problem — that is, a program and a property that may be true of the program — into an instance of the Pipe Jam game.

A Pipe Jam game is analogous to a dataflow network for a program. In this analogy, each ball represents a value, each pipe represents a variable, and assignment between variables is represented as one pipe flowing into another. In actuality, a Pipe Jam game

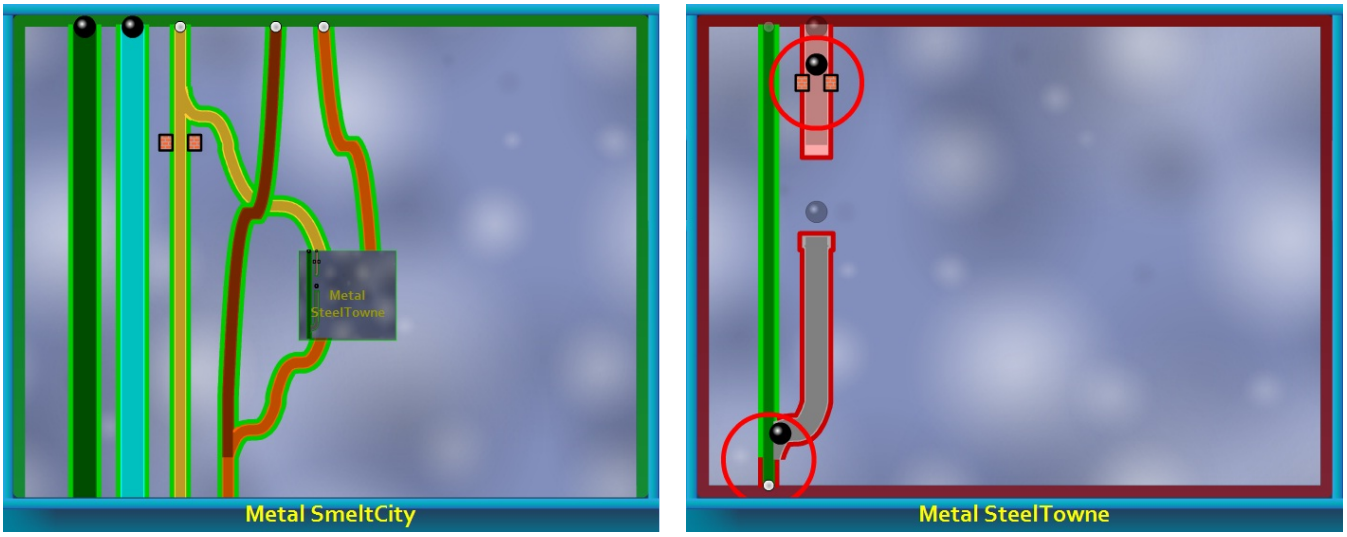

Figure 2: Detail of two boards from the Pipe Jam game shown in Figure 1. The left board demonstrates wide and narrow pipes, pinch points, a merge (near the bottom center), and a subnetwork. The right board demonstrates two collisions, each of which prevents the game from being solved. At the top, a large ball collides with a pinch point. At the bottom, a wide ball gets stuck trying to merge into a narrow pipe. The collisions are highlighted by red circles. A pipe segment is outlined in green if it contains no collision, and in red if it contains a collision. The gray pipe on the right board cannot be adjusted in width.

represents a relation we call "type flow". For example, a parameterized type `Map<String, Integer>` would be represented by three pipes, one (or more) each for the type qualifiers on `Map`, `String`, and `Integer`. The three pipes need not always flow in parallel, depending on how the code uses values of the given type.

The width of a pipe stands for the type of the variable that the pipe represents. These are not the underlying Java types, but the security properties that are represented in the pluggable type system. For concreteness consider a nullness type system: a wide pipe represents a variable that is permitted to contain the value `null`, and a narrow pipe represents a variable that is guaranteed to be non-null. More generally, for a two-element type system, the wide pipe represents the supertype and the narrow pipe the subtype.

Unmodifiable pipes stand for constraints arising from the source code. For example, a field dereference such as `x.f` requires that the reference `x` be non-null (narrow). The literal `null` yields a large ball, and a `new` expression yields a small ball. Similar constraints arise from secure sources and untrusted sinks in a security type system.

Each board represents a single procedure or method. Each level represents a class. An embedded board represents a procedure call. When there is a procedure call to a pre-annotated library routine, then its constraints are integrated directly into the network, to avoid a proliferation of subcomponents in the network.

The world map is a dependence graph among classes or, equivalently, a call graph. It ensures that callees are annotated before callers, though it accommodates mutual dependences. We take advantage of the decomposition of object-oriented programs into parts that can be specified, implemented, understood, and reused individually. This should help players just as it helps programmers. We may introduce new levels of abstraction (such as dividing the world into continents) to leverage modules within a program that are larger than classes.

Linked pipes that are of the same color represent different occurrences of the same variable or of two variables whose types must be identical. For example, a global variable or a class's fields flow to every method and thus appear on every board, but each global variable or field has only a single type that must be consistent across all methods in the program.

A buzzsaw represents an assumption/assertion about a given value — for example, a loophole, suppressed warning, or trusted cast in the type system. A programmer writes these to indicate externally-verified properties. Human insight is likely to be more effective than any inference tool, as evidenced by the sorry state of error messages for type inference systems. Once our system is fielded, then whenever many players, or a known successful player, uses a buzzsaw, a trained programmer can examine the code at that location. The programmer determines whether the code is correct and needs a trusted assumption, or the code has a bug and should be fixed. The game players have focused the scarce, expensive resource (expert programmer time) exactly where it is most needed.[3]

The names for levels and boards are chosen arbitrarily, but are intended to be memorable. Names from the original source code would not be meaningful to the players and might even confuse them. The players have no knowledge of computing, of the application domain of the program, or of its source code. The owner of the source code also might not wish for its identity or details about its design to be leaked to the players.

Distinction from dataflow.

The Pipe Jam game does not actually represent dataflow, but a different concept that we call type flow. The notion of type flow and our representation of it are novel, to the best of our knowledge. A particular variable may be represented by multiple pipes if it has a compound type. For instance, a variable of type `List<String>` would be represented by two pipes because two type qualifiers are possible, one on `List` and one on `String`. On the other hand, other variables would not be represented at all. For example, in a nullness type system, all primitive variables could be elided because a variable of primitive type (e.g., `int`) can never hold `null`.

Although programs very frequently contain loops in their dataflow, cycles in type flow on a single board do not occur, because the

[3] One can imagine other game mechanics for expert players, such as the ability to rewire the network. This corresponds to changing the logic of the program or rearranging assignments. It must be done in a way that respects the underlying Java types. Like the buzzsaw, this mechanism would reduce the player's score (though in most cases the final score would be more than for a complete failure to solve the level), and would be verified by a programmer.

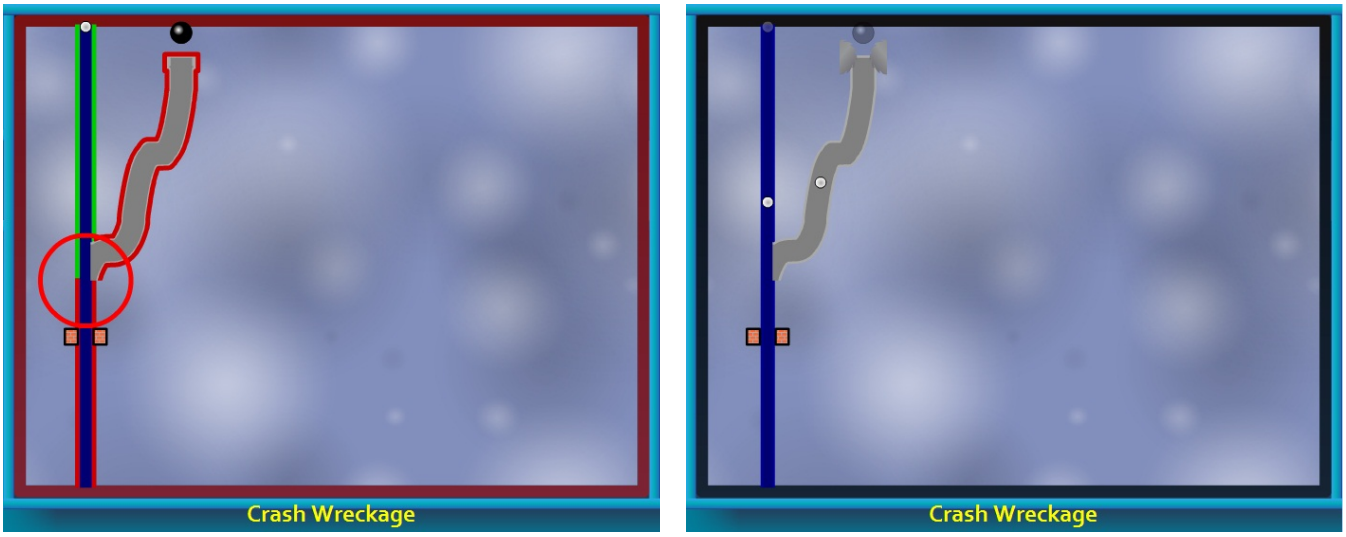

Figure 3: The left board is unsolvable. The gray pipe cannot be adjusted in width, and the black ball will get stuck trying to merge into the narrow blue pipe. Even if the blue pipe were made wide, the black ball would still get stuck at the pinch point.
The right screenshot shows how to solve the board by placing a buzzsaw. A buzzsaw cuts any ball that passes through it. The animation captured in the right screenshot shows that the large ball has been transformed into a small one after passing through the buzzsaw, and the level is solved. Placing a buzzsaw reduces the player's score, so players will try to solve their puzzles with the minimal number of buzzsaws.

network is really about relationships and flows between types and variables, not just between specific data values. Put more technically, it encodes the type constraints that arise when syntax-directed type constraint generation is performed on the program. While still complex, these are much simpler than the original program.

3.1 Mapping security vulnerabilities to game puzzles

We have set ourselves the initial goal of formally verifying that Java programs are free of the errors listed in the CWE/SANS Top 25 Most Dangerous Software Errors[4]. It is, in its own words, "a list of the most widespread and critical errors that can lead to serious vulnerabilities in software."

We now discuss how our pluggable type-checking approach can be instantiated for the errors on the list. Instantiation requires creating a *type system* that can detect the error or verify its absence, and a *mapping* from the type system to our Pipe Jam game. The mapping into Pipe Jam is generally straightforward. We give three rules that handle most situations.

- Many of the type systems have two qualifiers. The top qualifier can be mapped into a wide pipe and large ball, and the bottom qualifier can be mapped into a narrow pipe and small ball.
- When the type system consists of three type qualifiers in a chain (that is, $A :> B :> C$, where $:>$ is the supertype relation), then three sizes of pipes/balls suffice.
- When the type system contains two types that are incomparable, these can be thought of as orthogonal properties of an object. Examples from the nullness type system include whether a reference can be null; whether the referent is fully-initialized (versus being `this` within a constructor); and whether the referent is a key in a given map (this enables precise analysis of `Map.get()` calls). In this case, a different physical representation is required for the different orthogonal properties.
  We represent the primary or most important property as pipe/ball size, and the others as different colors or textures that are imposed on the pipe or ball. As a simple example, a value that is not necessarily fully initialized gets gray stippling. A more complex example is whether a value is present as a key in a map. Suppose there is a map whose pipe is blue. Then keys (pipes) that are guaranteed to be present in that map get a blue stripe. This shows the relationship between different values.

In some cases, a particularly sophisticated type system requires enhancements to the basic game mechanics. For example, consider the map key example above, and recall that `Map.get` returns non-null only if the key is in the map and all the map values are non-null. Representing a call to `Map.get` requires a new game widget that takes three input pipes and one output pipe. The widget has a narrow output pipe if *both* the pipe representing the key's type has a stripe of the color of the pipe representing the map's type *and* the pipe representing the value's type is narrow. In our experience with over a dozen pluggable type-checkers, the nullness type system has the most complexities such as these, because programmers use `null` to represent many special cases and because they can check for it at run time. Other type systems, such as those for security, are relatively simpler.

3.2 Simplifying the type constraints and the game puzzles

Even a multitude of game players would have trouble verifying a 5-million-line program with respect to 25 distinct type systems. Therefore, as future work we propose to simplify the problem before presenting it to the players. We will do so by pre-determining the types for a subset of variables in the program. Then, the players only need to find types for the remainder of the program. We will use the best program analysis tools available, and then the players will take over.

We expect to significantly reduce the size of the games that players must solve. One reason is that some properties are not applicable to many parts of the program. Nullness properties are not applicable to primitive types, which can never have the value `null`. Untrusted user inputs are strings, but most variables in a program are of other types. A second reason is that many values are restricted to a small

[4] `http://cwe.mitre.org/top25/?2011`

part of the program. In a correct program, user input strings are validated near where they are read, or at least do not flow into strings used for other purposes. A third reason is that highly-effective analyses exist that determine an answer for most of the program.

We plan to evaluate how much simplification is optimal for players. When a part of the code is difficult to verify, its context can provide critical clues, and it would be counterproductive to remove all of that and leave players just an abstract, unsolvable core. On the other hand, players will become bored and jaded if they are required to perform rote actions. At the same time, a game should have some easily-solved parts, to keep players engaged and to give them a feeling of forward progress.

The game player is accomplishing the task of type inference: creating an annotation for each variable declaration in the program, that gives additional information about that variable's type. Examples are whether the variable is permitted to hold the value `null`, and whether the variable is permitted to be mutated.

Whereas type *checking* is largely a solved problem, type *inference* emphatically is not. Even the most precise available inference tools leave much room for improvement. For example, the state of the art tool for inferring nullness properties is the Julia [24, 34, 35, 36, 37] tool for Java. It can prove about 98% of all dereferences in a program to be safe — that is, it proves that no `null` value ever flows to those dereferences, so those dereferences cannot result in a null pointer exception at run time. However, this leaves 2% of dereferences for a programmer to manually check — some may be erroneous code that might fail at run time, but most are probably safe even though Julia is unable to infer such properties. Human insight and higher powers of reasoning are required. As another example, a state-of-the-art tool for inferring mutability (whether side effects are permitted/performed) is Javarifier [31, 32] for the Javari [6, 39, 38] type system, but it, too, suffers imprecision due to its conservative analysis. An earlier inference system for Javari [22, 23] achieved even worse results, with precision of about 70% (compared to Julia's 100% precision and 98% recall).

Automated analysis and crowdsourcing are complementary. In particular, we can run the best available practical inference tool before the player starts, and use that as the initial configuration that the player attempts to improve. This leverages human intuition exactly where it is needed. In particular, every formal analysis has a limited vocabulary of properties that it can express, which limits the properties that it can prove. It is an unavoidable fact, related to the undecidability of the halting problem, that for every conservative (sound) program analysis, there are programs that never go wrong at run time but that the program analysis rejects as potentially erroneous. We hope that humans will be able to overcome these limitations via higher-level reasoning, pattern-matching, and heuristics to guide themselves through very large search spaces.

3.3 Human advantage and game-based expert development

When the game has a buzzsaw-free solution, then that solution can be found both automatically and very quickly: compute all pipes reachable by any large ball, and make those wide. We do not believe that game players will have any advantage over program analysis in such circumstances. However, our experience with pluggable type-checking so far leads us to expect that few if any programs will verify without buzzsaws — that is, without any trusted assumptions or suppressed warnings. It is our hope — yet to be validated experimentally — that people will have an advantage when a program is *not* verifiable. Perhaps people will observe the structure of the game, observe animated executions, notice patterns (such as that only small balls ever go down a particular chute, so the buzzsaw wouldn't actually have any effect and so can be placed for free), and use their intuition to find the smallest number of buzzsaws that can be placed. This will be more useful to the verification expert than an automated tool that chooses an arbitrary location at which to report a type conflict. The verification expert can also communicate back to the players, if they choose poorly, by preventing a buzzsaw from being placed at a particular location, or by adding a test case for the players to observe.

We expect that in general, problems that are harder for verification experts will be more challenging to players. We will evaluate which problems are more difficult in each domain.

An engaging game is capable of drawing in many new players. However, if the process of play does not produce increased skills, and improvement (both individual and collective) on ability to solve verification puzzles, then even a massive population of active players will not achieve our software verification goal. We need significantly more verification puzzle experts. This challenge is common to all games that attempt to solve hard problems. Foldit [11, 12, 10, 9], a proteomics game, effectively created the genre of games that solve hard problems. Prior to Foldit, it has not been known whether it is even possible to elevate human expertise on a particular domain to the point that it can disrupt the current scientific process and produce outcomes that present significant scientific advances. In the past few years, based on the success of Foldit, several researchers have developed games aimed at solving hard problems in science and engineering. Still to date, Foldit remains the only game with indisputable evidence of emergent expertise and outcomes that advanced the science.

The only way to accurately assess the emergence of expertise in the player population is to measure their performance on hard verification puzzles. Once assessed, the key game design question is how to modify the game towards improved performance. We plan an approach of iterative game sensitivity analysis, which will lead us to modifications that improve performance over time, as it did with Foldit.

4. RELATED WORK

Currently, there is no rapid, cost-effective approach for establishing that an application is safe. Testing and dynamic analysis are pragmatic and often effective, but they are unsound because tests are never guaranteed to exercise all program behaviors. Another common approach is to manually inspect the source code or machine code, relying primarily on expert human insight augmented by relatively low-level tools. This approach is slow, costly, and error-prone; automation, as we propose, is preferable. Another possible approach is formal verification via model-checking, theorem-proving, and similar technologies. These approaches are attractive because they have the ability to produce a proof of correctness that guarantees a particular property. However, they are not practical, for several reasons. They suffer a high false alarm rate, which is caused by analysis approximations that are a necessary compromise to make a sound analysis scale. For similar reasons, the properties they can prove are relatively weak, and in practice, loopholes/assumptions are required in strategic places. Even so, they require a high skill level — 6 months of training is considered a lower bound to use a theorem-prover, and model checkers also have a significant learning curve.

Automatic verification is the gold standard. One good example is Saturn [1], a precise and scalable static analysis tool that was used to verify cast safety of 6 million LOC from the Linux kernel. This is an impressive tool; however, custom analyses are written in a targeted logical programming language and require deep understanding of the analyses. Our approach is more flexible and general, and admits

both simpler and more sophisticated analyses, all written in Java.

Interactive verification of software has also seen recent impressive results. The formal verification of an L4 operating system microkernel [25] is a recent breakthrough of manual formal verification. The 8,700 lines of C code of an L4 microkernel were verified using the Isabelle/HOL interactive proof assistant [29]. The group co-designed the kernel and proof and started from scratch; they developed a Haskell prototype that was manually translated into a high-performance C implementation. The verification effort is huge: over 22 person-years of effort by highly-trained researchers. Another recent example is the manual verification of the Compcert C compiler [26] using the Coq proof assistant [5]. The compiler is written in 42,000 lines of Coq (note that this is both the implementation and proof) and took about 3 person-years. In contrast, by continuing to use Java as underlying development language and extending the type system with expressive, high-level types, we hope to allow even average software engineers to verify their software, with the help of the crowd.

A shining example of *software model checking* is the SLAM/Static Driver Verifier (SDV) tool [4, 2, 3] from Microsoft, which is used to show the correctness of kernel-mode drivers and is a standard part of the development process. The tool is applied to device drivers of between 1K and 30K LOC. This success is due to clever ideas, good engineering, and not least the choice of a very constrained domain with specific properties to check. Our goals are considerably more general.

Abstract interpretation [16, 15] serves as a foundation for several verification approaches. The Astrée tool is used to show the absence of runtime errors in safety-critical, embedded control code of up to a million lines of C code. Astrée runs not on programmer-written source code, but on generated code, which uses a limited subset of C (Astrée does not handle expressions with side effects, dynamic memory allocation, or recursion) and has properties that make the specific properties that Astrée checks easy to verify.

See our previous work on a type inference algorithm [20] for the Generic Universe Types ownership type system [18] and the Javarifier type inference algorithm [38, 32] for the Javari language for reference immutability [6, 39] for additional pointers to relevant literature.

5. CONCLUSIONS

This paper summarized the current status of the Verification Games project. Future work will focus on both the verification and game play aspects, for example, expressing more properties, scaling up, adjusting game play difficulty and interest, and play testing. More information and a demonstration video can be found at the project homepage[5].

Acknowledgments

Brian Britigan and Marianne Lee drew the Pipe Jam art. We thank Drew Dean for his encouragement and feedback.

This material is based on research sponsored by Defense Advanced Research Project Agency (DARPA) under agreement number FA8750-11-2-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Project Agency (DARPA) or the U.S. Government.

[5] `http://www.cs.washington.edu/verigames`

REFERENCES

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins, and B. Hackett. An overview of the Saturn project. In *PASTE*, pages 43–48, 2007.

[2] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: static driver verification with under 4% false alarms. In *FMCAD*, pages 35–42, 2010.

[3] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *CACM*, 54:68–76, July 2011.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, June 2001.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[6] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

[7] G. Bracha. Pluggable type systems. In *RDL*, Oct. 2004.

[8] Checker Framework website. `http://types.cs.washington.edu/checker-framework/`.

[9] S. Cooper. *A Framework for Scientific Discovery through Video Games*. PhD thesis, University of Washington, Seattle, WA, 2011.

[10] S. Cooper, F. Khatib, I. Makedon, H. Lu, J. Barbero, J. Fogarty, Z. Popović, and Foldit Players. Analysis of social gameplay macros in the Foldit cookbook. In *FDG*, 2011.

[11] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, and Foldit Players. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010.

[12] S. Cooper, A. Treuille, J. Barbero, A. Leaver-Fay, K. Tuite, F. Khatib, A. C. Snyder, M. Beenen, D. Salesin, D. Baker, Z. Popović, and Foldit Players. The challenge of designing scientific discovery games. In *FDG*, 2010.

[13] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, Feb. 1988.

[14] P. Cousot. Types as abstract interpretations. In *POPL*, pages 316–331, 1997.

[15] P. Cousot. The verification grand challenge and abstract interpretation. In *Verified Software: Theories, Tools, Experiments*, pages 227–240. Dec. 2007.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[17] P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL*, pages 12–25, 2000.

[18] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Department of Computer Science, ETH Zurich, Dec. 2009. Doctoral Thesis ETH No. 18522.

[19] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690, May 2011.

[20] W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for Generic Universe Types. In *ECOOP*, pages 333–357, July 2011.

[21] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA*, pages 24–27, May 2003.

[22] D. Greenfieldboyce and J. S. Foster. Type qualifiers for Java. `http://www.cs.umd.edu/Grad/scholarlypapers/papers/greenfiledboyce.pdf`, Aug. 8, 2005.

[23] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, Oct. 2007.

[24] Web interface to the Julia analyzer. `http://julia.scienze.univr.it`.

[25] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating system kernel. *CACM*, 53(6):107–115, June 2010.

[26] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7), July 2009.

[27] P. Martin-Löf. *Intuitionistic type theory: notes by Giovanni Sambin of a series of lectures given in Padua*. Studies in proof theory / Lecture notes, 1. Bibliopolis, 1984.

[28] M. Naik and J. Palsberg. A type system equivalent to a model checker. In *ESOP*, pages 374–388. 2005.

[29] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. 2002.

[30] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[31] J. Quinonez. Inference of reference immutability in Java. Master's thesis, MIT Dept. of EECS, May 2008.

[32] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of reference immutability. In *ECOOP*, pages 616–641, July 2008.

[33] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. 2006.

[34] F. Spoto. Nullness analysis in boolean form. In *SEFM*, Nov. 2008.

[35] F. Spoto. The nullness analyser of Julia. In *LPAR*, Apr. 2010.

[36] F. Spoto. Precise null-pointer analysis. *Software and Systems Modeling*, 2010.

[37] F. Spoto and M. D. Ernst. Inference of field initialization. In *ICSE*, pages 231–240, May 2011.

[38] M. S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006.

[39] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.