Verigames: Crowdsourcing Code Verification Through Video Games

Zan M. Balcom (zbalcom), Autumn Blackburn (kblack37),
Jake W. Chiang (jchiang2), Alex H. Davis (ahd2112)

June 8, 2018

Abstract

This report documents the steps and measures taken to return the verification game Flow Jam to an extensible and up to date state. An analysis of the original source code of Flow Jam revealed a large technical debt, notably sizable coupling between three particular modules. The analysis consisted of a graph of all classes in the program with each node representing a class and the edges representing dependencies. The source code also followed no clear software design scheme, so new systems were constructed to change the game to follow an entity-component-system architecture. The three classes were then refactored to fit into this new system by removing behavior from them and placing them into the relevant parts of the new architecture. After this work they were reduced in size in terms of code, had a reduced number of dependencies, and represented clear simple abstractions. Lastly, these classes were integrated into the new system and the game was returned to a functioning state.

Contents

1 Motivation

Computer programs have defects, which, as history and experience shows, can be very costly when they survive to be in the final product. For example, on June 4, 1996 the Ariane-5 rocket experienced a launch catastrophe due to an error caused by casting 64-bit floating point values to 16-bit signed integers[3]. Program verification is the process by which a program is proved to be free of some of these costly bugs; however, it is expensive to have trained programmers spend time to evaluate every line of code looking for these defects. This is the way nearly all verification is handled today. There are existing tools that can perform certain amounts of verification checking, but verifying a complete and meaningful program with these tools would take years even with a team of dedicated engineers. Due to these constraints, usually only the most important components of large codebases are verified.

Figure 1: Flow Jam Port Main Menu

Verification Games serve to transform the intricate task of code verification into an accessible game that can be completed with no special training, where finding a solution to a level provides valuable verification insight[1]. This game can then be deployed to a large audience, crowdsourcing the laborious task of code verification to a population that doesn't need any specialized education. Flow Jam (fig. 1), is one such game developed by a team at the University of Washington that uses a series of differently-sized pipes and colored liquid to represent the variables and values, respectively, of a program. Game levels are made by reading a Java program and a specification in the form of a type system and then converting these into corresponding system of pipes. The goal of the game is to manipulate the pipes to produce a configuration that allows the fluid to flow through the whole system. Such a solution equivalently is a proof that the program holds to the specification. One example of this type system in practice would be to look for null pointers in the program, with one width of pipe representing nullable assignments and another being non-null types. By solving the game, the player would be proving that no null value could be assigned to something that should not be null. If the player cannot find a solution, then they can use a "buzzsaw" to

make all the fluid flow through the pipes. This in turn signifies that there could be a type error in the code that the pipes represent; a trained software engineer can then look at that part of the code and resolve any errors they find manually. In other words, the buzzsaw helps identify areas where a configuration of the type system could not be found that solves the puzzle, which indicates that the specification is not yet met.

With a crowdsourcing approach to code verification, the task of identifying potential problem areas in the code for trained engineers to review becomes much more manageable. A current issue with code verification is that it is a laborious project that has to be undertaken by trained engineers and thus verifying 100% of a large program correct would take far too many people-hours of work to consider as an option. With verification games, large swathes of the program can be verified correct automatically by players and so the developers of the program can then utilize their engineers more efficiently. Currently, however, these verification games are only capable of representing small programs because the number of variables, values, and dependencies quickly becomes unwieldy in meaningful programs. As with Pipe Jam this limits the power of the tool considerably since the number of pipes in larger programs would quickly become unmanageable for a player to deal with.

2 Related Work

Other attempts have been made to create code verification games with different approaches and results. One is Paradox[18][15], which operates on the same principles as Flow Jam, although Paradox handles the complexity of large programs by leaving it up to the users to decide how much of the program they want to verify. Fig. 2 shows a game of Paradox being played. The red areas identify constraints that are not currently met by the program and it is the player's job to change the links between the small circles to meet that constraint. Although this solves the same problem as Flow Jam, it doesn't abstract the complexity of a program in a meaningful way. While in Paradox you can use the map in the bottom right hand corner to zoom in on an area to work on, in Flow Jam the user is only given a small area to work on at a time that is connected with other areas out of sight. We believe Flow Jam's way of obfuscating the complexity of the program is important to keeping players motivated to play the game and in drawing in new players for the crowdsourcing of verification.

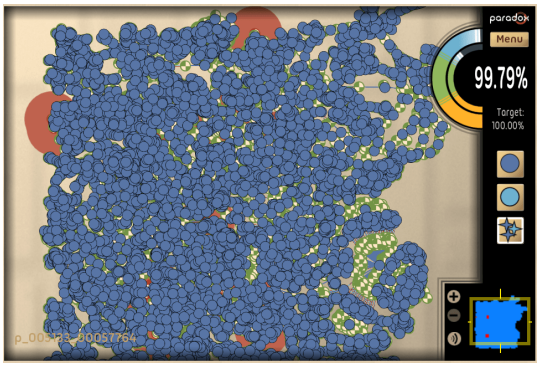

Figure 2: Paradox Full View

Pipe Jam is another project that is built on the same system as Flow Jam: it parses programs and constraints into JSON files that the game engine uses to construct its levels. This game represents variable assignments with pipes that lead from one screen to another to represent where variables are referenced and checked to see if they meet the constraint. The balls that run through the pipes represent those variables. If a ball is too large to fit into a pipe that represents the constraint, then that part of the program does not meet the specification. This is essentially the same mechanics as Flow Jam with a different skin, but we believe Flow Jam to be more aesthetically appealing than Pipe Jam.

Binary Fission[17] was created by researchers at UC Santa Cruz to move the task of identifying cyber vulnerabilities into the realm of crowdsourcing. From what little we have been able to find on it it seems to work by having players identify loop invariants that are hard for computers to recognize. Players of the game use different filters to sort out "quarks" in the game and from this they are filtering which invariants are true about loops in the code.

Besides code verification, games have been used to solve other kinds of difficult problems with success. A great example of this is Foldit[12], a game which has human players competing against each other to see who can fold a protein into the most stable shape. Prediction protein folding or designing new proteins that have a certain structure is an NP-hard problem and thus not practically solvable by computation alone. Another is Mozak[13], a game with the goal of deepening the understanding of neuron structure.

An effort to create verification games spearheaded by DARPA was released in 2013[18][10], but the links to the games mentioned on the website in the press release now point to broken versions or bad gateways. Currently Paradox is the only working verification game we could find. The research behind the games that were developed with the DARPA project points to a main flaw that was present in all the games. The problem was that all games of this type were not very fun to play. In any other game there is a development team that is designing the experience they hope the player will have, but with automatically generated verification games there can be little to no control over how the game is represented beyond that which is done automatically.

There also exist other tools for code verification besides games. For example, there is the Checker Framework[16], which Flow Jam uses as a backend to create the JSON level files. The Checker Framework's verifier is sound for most type annotations, like verification games, but also tends to create more false positives in its attempt to be entirely correct. Verification games, in contrast, utilizes people's innate problem solving abilities to reduce the amount of false positives it finds. Some research groups have been working with Markov Decision Processes to model non deterministic and stochastic properties of programs. In this process probabilistic model checking is used to analyze a system to obtain probabilities of events occurring in the system.[21][22] Automated computer aided solvers work by taking a program and translating it into a sequence of predicates for the program and then feed that into a satisfiability modulo theories (SMT) solver, then the SMT determines if those predicates are satisfiable[23]. This is useful but very different from how verification games work, they use the human brain which can still work in ways that are impossible for a computer to mimic to find solutions to novel problems

3 Approach

We decided that we would start updating the current codebase to a new platform and system architecture to make Flow Jam follow a modern software design with the hope of distributing the game to as large of an audience as possible, placing Flow Jam into the realm of large scale crowdsourcing among other crowdsourcing applications like TopCoder (boasting 1 million members)[19] to the protein folding solver FoldIt (75,000 players)[2].

The first step was to update the platform that Flow Jam was built on. The original game was written in ActionScript3, which has fallen largely out of favor in the game development world as Adobe has announced that it will stop supporting its Flash Player browser plug-in[4]. We began porting the codebase to the Haxe language[8] by utilizing the as3hx library[9] and its automatic conversion tool. We chose Haxe because of this existing tool to convert source code from ActionScript3 and also so the game can be easily configured to be distributed via multiple platforms such as iOS, Android, and the web, increasing the pool of potential players. After we had finished this porting (fixing any defects introduced by the conversion tool and any places where it failed to make a correct translation), we replaced the external libraries (such as Feathers)[20] the original codebase depended on that have either fallen out of use or do not have Haxe versions.

Once we did this and were able to compile the program, though, we ran into an issue: the position and scale of all the game objects was incorrect. We were unable to find the source of this issue after poring through the codebase and didn't think we could reliably fix this and any other issues that may arise from the poor code architecture. From this point we focused on refactoring the code to reducing the technical debt in the codebase and locate the flaws in the current code.

Figure 3: Flow Jam View Post-Port

We started rebuilding the system architecture of Flow Jam to follow an entity-component-system architecture, which would increase the modularity and reduce the coupling of the code. We determined a few classes to be causing a large amount of the issues and to break them down into modules that we could then insert into the new architecture. The dogma we chose to follow for our refactoring was to be able to represent clearly what the three classes that had the most coupling were meant to do. We did this by removing behavior that did not fit

a simple model and moving those behaviors into their own modules that could be concisely represented.

4 Architecture

As mentioned above, we moved the codebase away from its old, less flexible and maintainable architecture to an entity-component-system architecture (fig. 4). From its old state, the code required a lot of refactoring. Since it was designed as a prototype and not a codebase to be maintained long-term, it had no definitive architecture that would allow for interesting feature growth. Many classes (Level, World, GridViewPanel) were so-called "god classes", performing far too many tasks and having far too large of a stake. As well, many classes made extensive use of static variables to share information. There was also no clear recognizable architecture. For these reasons, we believed a refactoring is in order and should be our top priority.

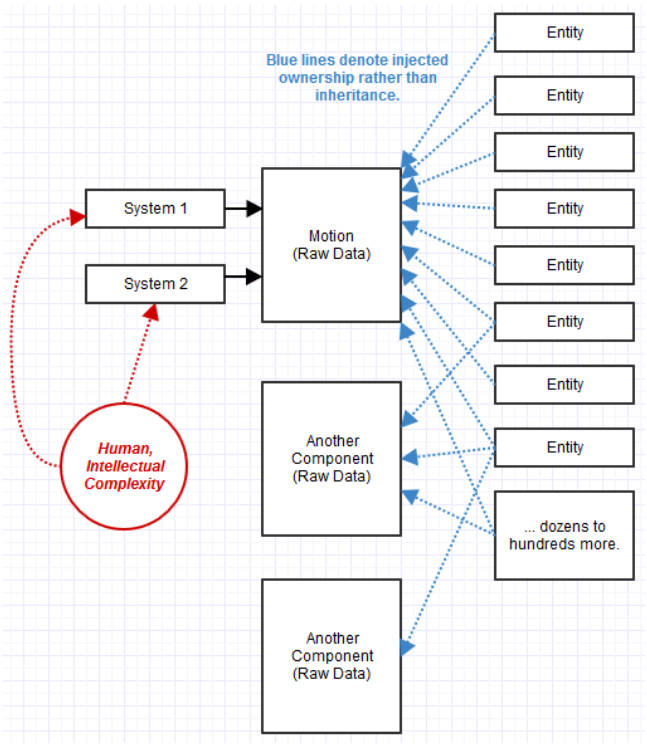

Figure 4: Standard Entity-Component-System Architecture

The goal of migrating the codebase to an entity-component-system architecture ensures new game systems can be added more easily as well as reducing the current complexity of the code (for example, the "god class" mentioned above, scenes.game.display.Level, is over 2000 lines of code and performs many different tasks that could be refactored into other classes).

ECS architectures allow for easier development of new game systems because adding one in is (almost) as simple as creating a new type of component and a system to govern it. They also reduce the complexity of the code through the principle of defining objects through composition instead of inheritance.

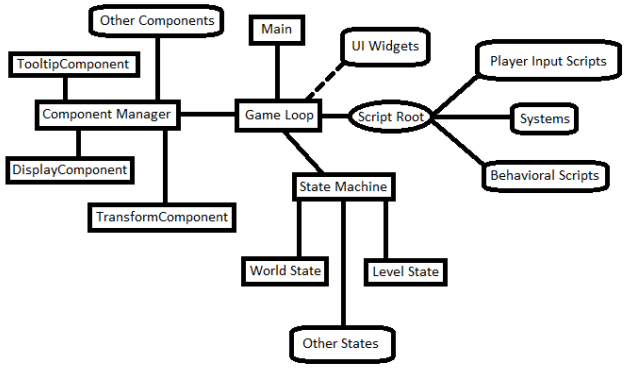

Figure 5: New High-Level Architecture

Above (fig. 5) is the new architecture at a high level. Main refers to the static entry point, where possibly different versions of the game (for different release platforms) can be instantiated.

The game loop is not quite a class itself, but part of a class instantiated by Main. It contains many different elements of the architecture and serves as the central locus of communication between them. It passes itself down to classes that may need to reference resources, such as scripts that operate on some particular entities.

The component manager is one element of the game loop class. It handles all addition and removal of components from entities as well as providing methods to retrieve components of a certain type or that belong to a specific entity. Components are generalized objects that systems can operate on; for example, a transform component contains an entities transformation information (translation, rotation, scale) that a free transform system would reference to move that entity's display component. For example, in Flow Jam, the fluid in the pipes would have a display component that references their sprite, a transform component that determines where that sprite is placed on screen, and a width component (narrow, wide) that would be checked against the constraint graph.

The state machine is another element of the game loop class. Its responsibility is to ensure clean transitions between different states of the game (ie, the main menu and the puzzle-solving gameplay would be two separate states). This means cleaning up the resources of the previous state and making sure the next state has the resources it needs. It also passes information between transitioning states, if necessary, such as the level data when transitioning from the level select state to the level state.

The script root refers to the field in the game loop class that is the root of the script tree. The game loop calls update on the root of the script tree, which then calls update on its children, and so on. These scripts define the mechanics of the game. The player input scripts

process player input such as keyboard, mouse, or touch events and pass this information along to affect the game state (usually in the form of events). The behavior of clicking on the flows to change their width would live here. The systems, as mentioned above, deal with the behaviors the components are meant to express, such as the free transform system that updates the screen position of any objects that have both a display and a transform component. The behavioral scripts handle the rest of the game's behavior; for example, the changing of flow states from small to wide would live here.

4.1 Challenges and Risks

Since the announcement that Adobe will be discontinuing Flash many modern browsers have stopped providing Flash support, adapting the Flow Jam codebase to a more accessible system was both a top priority and one that required an indeterminate amount of work that only became clear once the porting was underway. This process also required becoming familiar with both the ActionScript3 and Haxe languages, which were new to many of us.

This proved to be a challenge in the course of this project. Our original plan was to port two verification games (Pipe Jam and Flow Jam) from AS3 to Haxe, but due to the complexity of one of the versions and the technical dept present in both games, we chose to focus our efforts on only one of them, in the interest of making it a fun and enjoyable game. This was a hard decision to make because of the effort we had already put into porting both versions, but in the interest of producing a viable and well running game, we chose to discard Pipe Jam and move forward with the game we had made the most progress on Flow Jam.

Another challenge is that the codebase we were given for the game was clearly in a prototype state - even after we finished porting and were able to get the game to compile and run, there were significant issues with its comprehensibility, extensibility, and maintainability. Significant architectural issues were pervasive throughout the codebase that we refactored, leading to us taking steps to transition the codebase to a more traditional entity-component-system architectural pattern.

Adapting the existing code to fit into a new architecture proved to be a greater challenge than we anticipated. The original codebase had low cohesion, high coupling, and poor documentation, which combined with most of the team members being unfamiliar with an entity-component-system design, made the restructuring slower and more error-prone than anticipated.

Now that the game is adapted and ready to be played, future developers will need to remain cognizant of the problems involved in ensuring that the game is entertaining enough to draw a crowd of users. It will require consistent testing, feedback, and iteration to ensure the end product is enjoyable, or else risk not not attracting enough players to turn crowdsourced verification into a reality.

4.2 Results

Our original plan of porting two games, as mentioned in the Challenges and Risks sections, proved to be too much for our team to complete in the time we had to complete the project. Our plan then was to forego adding additional features and instead focus on a thorough evaluation and refactoring of Flow Jam's codebase. We believed this was a necessary task of greater importance than any feature addition could be. Owing to our awareness of how

few of the existing verification games are in a runnable state, let alone a developable one, it was imperative for the future of the Flow Jam project that we concentrated on reducing technical debt and bringing it to a future-proofed state. The old structure of the codebase featured a large amount of coupling, which in addition to making parsing and debugging extremely difficult, also reduced any ability to hook in additional features. We have created a visualization of this structure by representing the codebase as a directed graph where each class is a node and an edge represents a dependency on another class[12]. The size of a node is relatively proportional to the number of lines of code in the class it represents. We have also displayed the number of 3-cycles in the dependency graph as a simple measure of coupling, since a non-hierarchical codebase is more difficult to comprehend.

We identified three classes that we believe to represent the worst cases of high coupling: Level, World, and GridViewPanel. We picked these because they are large classes with no clear, single purpose and are tightly coupled with each other, so we cannot concisely describe their functionality. Figs. 6, 7, and 8 below demonstrate the coupling of these classes and also that they form a 3-cycle of references, which further confuses the specific purpose they serve.

We started refactoring these classes by first comprehending their role and relationship to other parts of the codebase and then separating their functionality into components that fit within our ECS architectural plan. For example, Level and World should be refactored into classes that contain just data and no display properties. Furthermore, all input handling and displaying they do should be refactored into scripts or widgets as described in the architecture section.

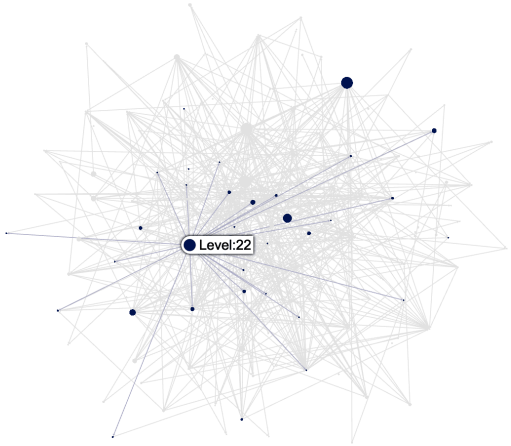

Figure 6: Dependency Graph Highlighting Level

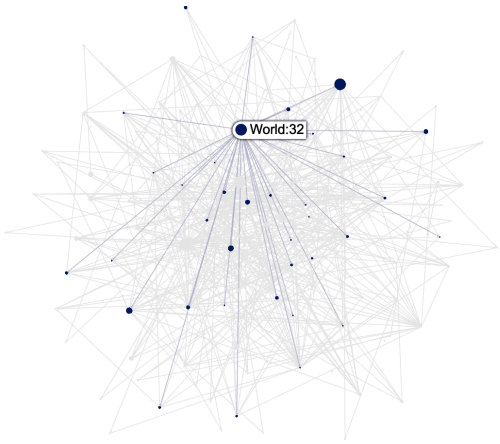

Figure 7: Dependency Graph Highlighting World

World, in its original form, was handling so many game components that it did not make sense in its current state. The class as a module could not be described in any quick meaningful way. It did many things that should have been handled by a different game system rather than by itself. Registering achievements, user input, displaying the world and level were all part of the class. These points and its high amount of dependencies pointed to it needing a code refactor. To refactor this class we started by moving behaviors that did not help describe World a place to store data about the running game, to scripts that are handled by the game engine. This involved checking that the behaviors were indeed unique to World and would not have side effects in other classes once the refactoring was done. After these initial changes were done World was already down to less than half its original amount of code. Next we looked deeper into some behaviors that were coupled with what we wanted World to be, a place to keep the data related to the current game, and its former purpose, a catch all class for things related to the game as a whole. These behaviors were harder to take out as many of them happened inside methods that primarily deal with the world in its data storing role.

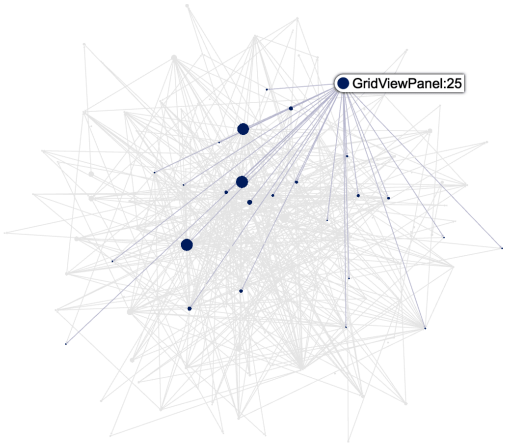

Figure 8: Dependency Graph Highlighting GridViewPanel

Tied to the World module was the GridViewPanel module, which World utilized for broad display manipulation, such as camera panning and zoom. GridViewPanel was also host to a collection of ancillary miscellaneous functions, such as fanfare particle effects, tutorial tooltips, and keyboard event handling. Separating this class involved several steps, the first of which was delegating its event handlers to the new centralized system used by all refactored classes. Following this organization, remaining functions were examined in the context of what GridViewPanel was intended to do, namely, provide a main display structure for the game world, and then pared down to only the essential pieces. Unrelated behaviors were separated into new single purposed modules.

5 Future Work

Though we have taken the first major steps toward cleaning up the codebase, there is still work to be done. Much of the top-level architecture has been restructured, but many improvements can still be made for some of the lower-level classes. For example, most of the game objects extend a ToolTippableSprite class but this functionality should be moved out into a system that is responsible for adding tooltip components to existing entities. Most other classes would also be well served if they were refactored the way Level, World, and GridViewPanel have been, separating kludged modules and transferring state and event handling to the new game engine systems.

The next step after reducing the technical debt of the codebase and ensuring that there is a playable level is to playtest the game in its current state. This will involve asking people to play the game while being observed and then to complete a short questionnaire afterwards in order to obtain feedback. This data will then be used to evaluate and iterate on new features. A fleshed-out plan for playtesting can be found in Appendix 7.1.

6 References

[1] Werner Dietl Stephanie, Dietzel, Michael D. Ernst, Nathaniel Mote Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popovi. Verification Games: Making Verification Fun. https://homes.cs.washington.edu/~zoran/verigames-ftfjp2012.pdf, 2012.

[2] Thomas D. LaToza and Andr van der Hoek. Crowdsourcing in Software Engineering: Models, Opportunities, and Challenges. https://pdfs.semanticscholar.org/787c/e8e0973fd21051260385bb27d74f6f53c7f9.pdf, 2016

[3] Gleick, James. "A Bug and a Crash." *A Bug and a Crash by James Gleick*, New York Times, 1996, www.around.com/ariane.html.

[4] Mackie, Kurt. "Adobe and Browser Makers Announce the End of Flash." *Redmond Magazine*, 25 Jul. 2017. Web. 4/5/2018. https://redmondmag.com/articles/2017/07/25/adobe-ending-flash-support.aspx

[5] Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, Seth Cooper, and Zoran Popovi. On the Harmfulness of Secondary Game Objectives. http://grail.cs.washington.edu/projects/game-abtesting/fdg2011/fdg2011.pdf, 2011.

[6] Hunicke, Robin, Marc LeBlanc, and Robert Zubek. "MDA: A formal approach to game design and game research." *Proceedings of the AAAI Workshop on Challenges in Game AI.* Vol. 4. No. 1. AAAI Press San Jose, CA, 2004.

[7] Owens, Thomas. "Is it reasonable to build applications (not games) using a component-entity-system architecture?" *StackExchange* 4 Feb. 2018. Web. 4/12/2018. https://softwareengineering.stackexchange.com/questions/186696/is-it-reasonable-to-build-applications-not-games-using-a-component-entity-syst.

[8] Haxe - The Cross-platform Toolkit. (n.d.). Retrieved from https://haxe.org/.

[9] HaxeFoundation. HaxeFoundation/as3hx. Retrieved from https://github.com/HaxeFoundation/as3hx.

[10] Crowd-sourced Formal Verification Program Generates Thousands of Software Annotations. (n.d.). Retrieved May 10, 2018, from https://www.darpa.mil/news-events/2015-05-28.

[11] Verigames: Free puzzle games featuring logic, math, and science. Retrieved May 24, 2018, from http://www.verigames.com/.

[12] Flow Jam Coupling Graph, https://verigames.github.io/flowjam/.

[13] Foldit: Solve Puzzles for Science. Retrieved May 17, 2018, from https://fold.it/portal/.

[14] Mozak. Retrieved May 17, 2018, from https://www.mozak.science/landing.

[15] Paradox — Center for Game Science. Retrieved May 24, 2018, from http://paradox.centerforgamescience.org/paradox/Paradox.html.

[16] The Checker Framework. 1 May 2018. Retrieved May 24, 2018, from https://checkerframework.org/.

[17] Stephens, Tim, and Dina Basin. "UCSC and SRI Release New Game for DARPA Crowd-Sourced Software Verification Program." *UC Santa Cruz News*, 23 June 2015,news.ucsc.edu/2015/06/binary-fission.html.

[18] Dean, Drew et al. "Lessons Learned in Game Development for Crowd-sourced Software Formal Verification." https://pdfs.semanticscholar.org/57f4/06f355daae96c32f5deae0f9381c53147dac.pdf.

[19] Our Community — Topcoder. Retrieved May 30, 2018, from https://www.topcoder.com/about-topcoder/community/.

[20] Feathers - Build user interfaces for games and applications with Feathers. Bowler Hat LLC. Retrieved May 30, 2018, from https://feathersui.com.

[21] Kwiatkowska, Marta, et al. "Automated Verification Techniques for Probabilistic Systems." *SpringerLink*, Springer, Dordrecht, 13 June 2011, link.springer.com/chapter/10.1007/978-3-642-21455-4_3.

[22] Marta Kwiatkowska, Gethin Norman and David Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of LNCS, pages 585-591, Springer, 2011.

[23] "Satisfiability modulo Theories." *Wikipedia*, Wikimedia Foundation, 9 May 2018, en.wikipedia.org/wiki/Satisfiability_modulo_theories.

7 Appendix

7.1 Playtesting Plan

Flow Jam will be evaluated by playtesting the games on students at a large university. We will test each player for 5 minutes while monitoring their gameplay. The players will be given a laptop with the game loaded and at the opening screen and will be provided with a brief description of what the game is and what they are doing in terms of verification while they play. They will then be given the task to play the game with no further instructions. During the test observers from our team will collect data on different metrics these will include:

1. Whether playtester ended playing early or not and by how many minutes
   (a) We believe this will point to the version lacking in fun mechanics
2. Whether or not playtesters show of signs of frustration playtesters display. Signs of frustration will be defined as exasperated noises or stressed body signals
   (a) This is a subjective measurement but it can be useful to gather this information in the instance were the tester does not offer information on their frustration freely. If we think someone was frustrated with the game we can ask them why and in

what point of their gameplay did they feel that way. This will help identify areas where things may need to be changed or explained in a more precise way.

After playing the game for 5 minutes (or whenever the tester ends early) the playtesters will be asked the following questions about their experience playing Flow Jam.

1. On a scale of very likely, likely, no opinion, not likely, never, where would you place yourself in wanting to play the game again?
   (a) This will be a general measurement on entertainment value of the game.
2. Please rate the level of frustration you experienced while playing the game on a scale of 1 to 10.
   (a) We believe frustration is a good measurement for assessing the game. Players who experience this are less likely to want to play the game in the future. Frustration could have many causes when playing a game and as we will not be changing any major gameplay mechanics of the game we will want to measure this in some way.
3. Are there any aspects of the game that were unclear or was there any point when playing that you didn't know what to do?
   (a) This will be used to further evaluate the games entertainment if the more general question yield no useful information.
4. Would the addition of any of the following features make you change your answers to any of the previous two questions? Leaderboards? Badges/Achievements?
   (a) This question will help us get more input for features that we want to add to the game to increase player engagement.

After an initial round of playtesting we will use playtesting sessions of this structure to incorporate additional features into the game. As game improvements are proposed and implemented, these playtests can be repeated using an A/B testing format wherein testers are randomly assigned to either play a control version of the game with no changes or a modified version that has a new feature, such as secondary objectives or a different scoring system. Using the measures of player engagement and enjoyment (based on observation and the self-reported feedback) we can determine the efficacy of the feature as a gameplay enhancer and whether it improves the game.

7.2 Code Repository

The repository of all code in this project can be found at:
`https://github.com/kblack37/verigames`