Oisin Doherty

oisind

Ryan Bruntz

rmbruntz

Visual Verification

Formally verifying that any given program is correct takes important development time away from skilled individuals, ultimately costing large amounts of money and time. Prior work has been done to create the framework for a process of gamifying the act of program verification to make it more applicable to a crowdsourced model. In our proposal, we present an implementation and additional features that lend to the accessibility and scalability of such a system.

The primary issue with current program verification is that it requires a skilled individual to have an extensive knowledge of the codebase and associated algorithms to either manually prove that the program is correct, or to use an existing automatic verification software such as Saturn or Coq that still require extensive knowledge of proprietary languages and proof techniques. These limitations imposed by current automatic software are necessary for such an exact task, but require large time investments and consequentially, increased costs. The current implementation of PipeJam by Verigames serves as a proof-of-concept that such a tool can work, showing that such a game can be scaled up depending on the number of players to potentially solve other important issues surrounding software development. The current PipeJam shows an example that solves issues involving the detection of null pointer exceptions. Our implementation attempts to solve issues of scalability by having a skilled individual partition the code base into separate ‘game boards’ that multiple users can solve. Although the translation from code to game boards is no longer fully automatic, it allows for much larger code bases to be gamified. For extremely large inputs, this will still take time away from this skilled developer to both partition the inputs and create fixes based upon the results of games but will ultimately still reduce the total time needed to verify the input. As such, our implementation primarily targets improving efficacy rather than quality.

Our first approach to solve this problem is by partitioning small subsections of programs into individual ‘game boards’ that could reasonably be solved by a single player. In prior approaches, large inputs would grow exponentially in state spaces as the number of parameters and states increased. Our version would solve this problem by having a programmer mark independent regions of code as miniature game boards that can be solved almost independently from others. The programmer would also determine which future “pipes” between these regions would be wide or narrow, depending on the code related to a given pipe. This task could be partially automated – perhaps a tool could split the program into likely sections, and the programmer could confirm splits and determine the type of pipes connecting the automatically generated sections.

Another approach to the scalability problem would be a combination of crowdsourcing and automation to handle breaking up the board into separate pieces. The part of the process where the board is broken up into pieces could be automated based on which regions of the code have few ‘pipes’ connecting them. Then, the value of these pipes (in our example, non-null or possibly-null) could be chosen by users based on their experience or reasoning about the puzzle. For example, if a given section of the board is terminal (no outputs to other sections) then a player could try to solve that section with

various combinations of inputs, aiming for the least restrictive possible combination, before moving onto other sections that could vary based on the values of the connecting pipes.

With enough players and few enough input pipes, we could modify this terminal board to represent each possible combination of inputs (as a static pipe) and distribute these modified boards to players. As players discover solutions to various input combinations, strictly more restrictive combinations could be pruned from the pool of variations, until a "consensus" (i.e. progress stops, experienced players are stumped, etc.) is reached on which variations have the least restrictive inputs while still being solvable.

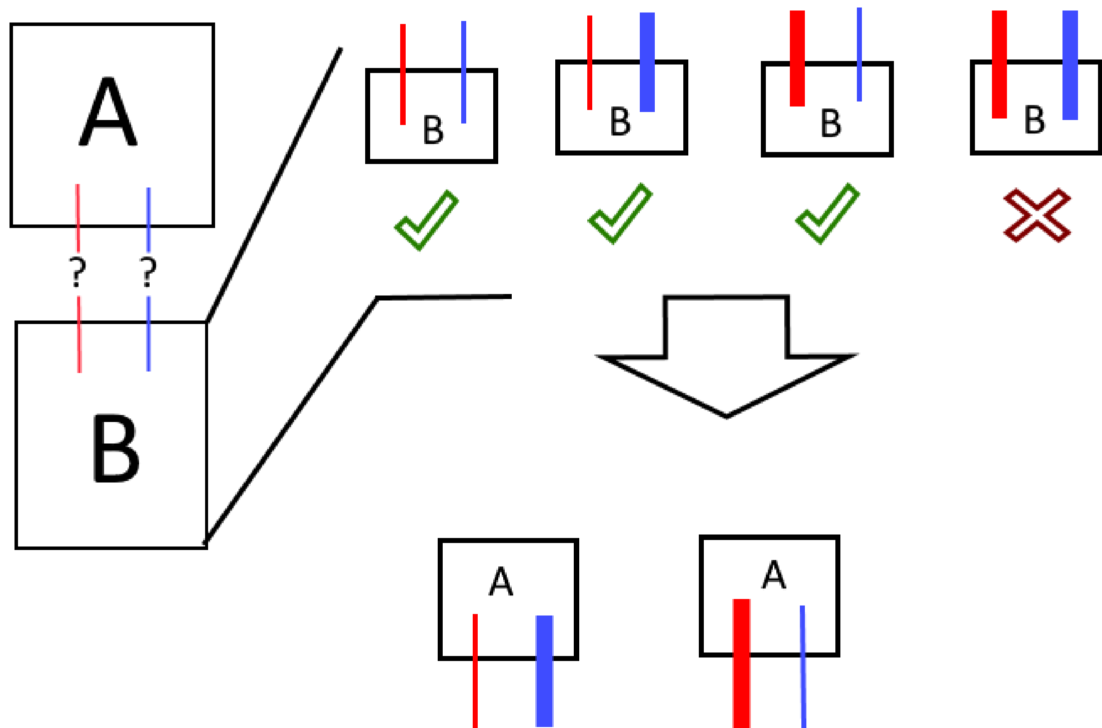

Figure 1: An example of the crowdsourcing/automation option for decoupling game boards. In this instance, boards A and B were split automatically, with 2 pipes connecting them. Players determined that the three less-restrictive combinations were possible for B, and a "concensus" was reached that the most restrictive combination (both narrow) was not possible. This narrowed the possibilities for A to only the middle two options, since the "both wide" option is strictly worse than either of these.

The single most challenging aspect of developing such a tool is that we rely on users to actually play the game and solve puzzles. If the puzzles supplied to the users are too difficult or the game isn't fun enough, then users have no incentive to continue playing. As such, choosing an appealing visual design with 'fun' gameplay mechanics is as important to the long-term success of our tool as the actual implementation of transforming code into game boards.