Chekofv: Crowd-sourced Formal Verification

Heather Logas[3], Florent Kirchner[1], John Murray[2], Martin Schäf[2], and E. James Whitehead Jr.[3]

[1] CEA, LIST
[2] SRI International
[3] University of California, Santa Cruz

Abstract. Over the past year, we have been developing Chekofv, a system for crowd-sourced formal verification. Chekofv starts with an attempt to verify a given C program using the source code analysis platform Frama-C. Every time the analysis loses precision while analyzing looping control-flow, Chekofv tries to obtain an invariant to regain precision using crowd-sourcing. To that end, Chekofv translates the problem of finding loop invariants into a puzzle game and presents it to players of a game, Xylem, that is being developed as part of the Chekofv system. In this paper, we report on the design and implementation of the Chekofv system, the challenges and merits of gamification of the invariants detection problem, and problems and obstacles that we have encountered so far.

1 Introduction

Software verification is a very labor-intensive task that requires highly trained experts. However, not all of the work requires special knowledge of verification or the underlying source code. Some parts, like identifying suitable predicates to help a verification tool to find good abstractions, only require an ability to identify abstract patterns that hold over a sequence of program states (e.g., loop invariants). While the degree of automation in finding and refining abstractions is increasing rapidly with the introduction of techniques such as counterexample-guided abstraction refinement [**?**], interpolation [**?**], or, more recently, abduction [**?**], these techniques never completely replace the need for user provided abstraction hints.

As part of DARPAs crowd-sourced formal verification program (CSFV) we have been developing a system called Chekofv (Crowd-sourced Help with Emergent Knowledge for Optimized Formal Verification) that tries to gamify the problem of predicate discovery and invariant generation and make it presentable to a broad audience. The goal is not to fully automate verification but to crowd-source particular tasks where possible.

As a case study, CSFV tries to prove the absence of bugs that are related to particular CWEs (Common Weakness Enumeration), e.g., buffer overruns, or dangerous use of format-strings, for real-world `C` code. While the goal itself is very ambitious, the focus is on which part of this verification could be crowd-sourced into a game. Here, CSFV has two main requirements: (1) the players of the game should not need to understand anything code-related to deal with the information that they are being presented, and (2) the game must be enjoyable by a broad audience. In the following we discuss some of our design choices, preliminary results, and the problems and merits of our system.

The first design choice we had to make is which part of the verification can be crowd-sourced under the given requirements. Here, we decided that finding loop invariants or at least predicates that can help to discover loop invariants is a suitable target. The intuition is that finding a loop invariant corresponds to identifying a pattern that describes a shared property of a sequence of program states. Similar tasks can be found in riddles where the player is presented a sequence of shapes and has to identify the one shape that does not follow a certain pattern (that is not stated in the riddle). So, instead of identifying an outlier in a sequence, we ask the player to state the pattern (and following the idea of abstract interpretation, program states can be represented as geometric shapes).

The sequence of states for which players have to identify patterns is then generated from (symbolic) executions of loops in a given program. Of course, such a pattern (or predicate) does not necessarily represent a loop invariant, however, having a set of such predicates may help an automatic analysis to find invariants more efficiently.

The next design choice is how to utilize these predicates and invariants for verification. In general, using invariants with deductive verification sounds more promising, we instead decided to use the Value Analysis plugin from Frama-C [?], which relies on the principles of abstract interpretation. While this might seem counter-intuitive in the first place, the motivation is that crowd-sourced predicates and invariants can only be of a very limited complexity, in particular when no knowledge about the code is available to the players. In that sense the crowd-sourced predicates and invariants can be seen as hints to the abstract interpretation how to sharpen the symbolic state after widening, but they are very unlikely to be sufficient for a deductive proof.

Probably the most important design choice is how to design a game that is attractive to non-expert players and yet allows the creation of predicates and invariants that are useful for formal verification. Certainly the average puzzle player will not be able to state an invariant that uses ghost variables, helper functions, and multiple levels of quantification. However, an overly simplistic game might be frustrating if players feel that the limited expressiveness of the game hinders them from expressing what they want. In the following, we discuss our game design choices and present the feedback that we collected from players so far.

2 Crowd-sourcing Invariant Generation

Our game, Xylem: The Code of Plants[4], is an iPad game where players make mathematical observations about synthetic plants, which are turned into predicates used for the construction of loop invariants. The game is a logical induction puzzle game where the player plays a botanist exploring and discovering new forms of plant life on a mysterious island. The Player observes patterns in the way a plant grows, and then construct mathematical equations to express the observations she makes.

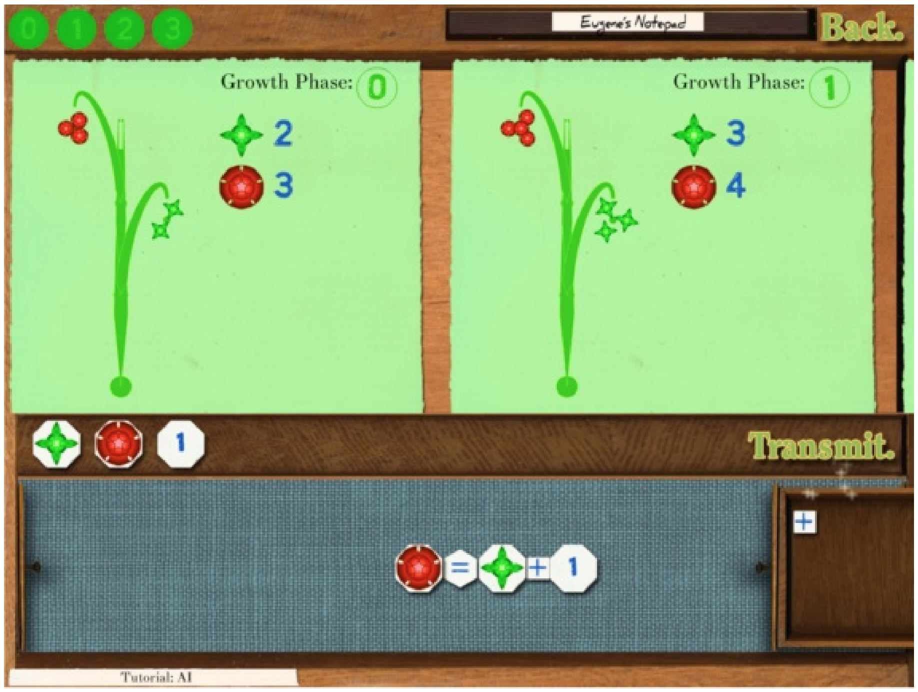

Fig. 1. Screenshot from our crowd-sourcing game, Xylem. The state of a loop iteration is represented by one growth-phase of a flower. Each blossom represent a variable, and the petals represent that value of that variable in this state. Players have to provide a generic description for all growth-phases of this flower (i.e., an invariant for the k-bounded sequence of states).

[4] An overview of the game is given on: http://xylem.verigames.com/, and the iPad version is available on https://itunes.apple.com/us/app/xylem-the-code-of-plants/id736179826?mt=8

Each puzzle presented to the player is generated from a loop in a program. To that end, the loop is (symbolically) executed k-times, and the program state is logged each time at the loop entry.

In the puzzle, each of the k recorded states is a *growth phase* of a flower. Each program variable is represented as a blossom of that flower and the petals of the blossom represent the current value of that variable. The narrative is that the player should find a description that describes the evolution of the plant as precise as possible. Figure 1 shows an example of a puzzle in Xylem. This narrative is motivated by the fact that there is an arbitrary number of possible invariants for a loop, and that it is not possible to determine the best one. This way players can provide weak descriptions of plants (i.e., weak predicates) which can be refined by other players later on.

The crowd-sourced invariants for the bounded sequences are not necessarily loop invariants since they are only generated from one possible path through the loop, however, they can serve as predicates for other loop invariant discovery techniques.

For the game to be enjoyable, we had to limit the number of variables (or blossoms) per puzzle. Real-world `C` code can have several dozens of variables that are being modified within the same loop. The feedback from our beta-testers suggests that anything above five variables is confusing and frustrating. Hence, we split the symbolic states into smaller portions that contain at most five changing variables.

3 First Results and Future Work

In a first feasibility study, we have generated puzzles for a set of programs from the sv-comp loop benchmarks [?]. So far, players of Xylem have solved 9589 puzzles. Out of these solutions, 5395 were duplicated answers (either exact duplicates or logically equivalent). For 1488 solutions, Frama-C could verify that they are valid loop invariants, for 6590 Frama-C could show that they are not invariants, and for 1511 invariants Frama-C failed to produce a result because they contained non-linear expressions that could not be handled by the employed theorem prover. This gives hope that even the relatively simple predicates that can be generated by Xylem are suitable to assist formal verification.

We carried out several player interviews to assess the usability of the game. One of the main complaints was that players wanted to be able to express transition predicates rather than invariant properties. Players complained that they want to state properties such as “x always increases by one”, or “y is equal to x from the previous state”. This seems to be a weakness of using the concept of evolution in our narrative and we are currently exploring ways to improve this.

Probably the biggest challenge that we are facing for our future work is the scoring system. For reasonably large programs, it is not feasible to check a users solution on the server within a reasonable time. However, since there are many formulas that hold for a bounded sequence of states (including all tautologies), it is vital for the long-term motivation of the game to provide immediate feedback about the quality of a solution to the player.

Acknowledgements

This work was supported by the US Defense Advanced Research Projects Agency under Contract No. FA8750-12-C-0225.