AFRL-RI-RS-TR-2016-060

CROWD-SOURCED HELP WITH EMERGENT KNOWLEDGE FOR OPTIMIZED FORMAL VERIFICATION (CHEKOFV)

SRI INTERNATIONAL

MARCH 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2016-060 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
CARL R. THOMAS
Work Unit Manager

/ S /
RICHARD MICHALAK
Acting Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2016	FINAL TECHNICAL REPORT	JUL 2012 – OCT 2015

4. TITLE AND SUBTITLE

CROWD-SOURCED HELP WITH EMERGENT KNOWLEDGE FOR OPTIMIZED FORMAL VERIFICATION (CHEKOFV)

5a. CONTRACT NUMBER
FA8750-12-C-0225

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
John Murray, Jim Whitehead, Florent Kirchner

5d. PROJECT NUMBER
CSFV

5e. TASK NUMBER
SR

5f. WORK UNIT NUMBER
II

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025-3453 (PRIME)
University of California Santa Cruz, 1156 High St., Santa Cruz 95064
French Commissariat a l'Energie Atomique, 91191 Gif Yvette Cedex, France

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St.
Arlington, VA 2203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-060

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # DISTAR 25646
Date Cleared: 17 Feb 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Formal Verification of Software is an expensive and time consuming task aimed at discovering and correcting software errors that lead to programming errors. The Crowd-Sourced Formal Verification (CSFV) program was developed to explore utilizing games to prove correctness proofs for software. Leveraging human pattern recognition skills, the CSFV games provide formal verification proofs a machine analyzing the code cannot. The SRI team developed two games; Xylem; The Code of Plants, and Binary Fission to prove Crowd Sourced game play can improve Formal Verification effectiveness and reduce the cost to verify code.

15. SUBJECT TERMS

Formal Software Verification, Crowd-Sourcing, Games, Cyber Security, Human-Machine Systems, Frama-C

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CARL R. THOMAS**
U	U	U	UU	142	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

In early 2012, a multi-national research team was formed to undertake advanced research as part of DARPA's Crowd Sourced Formal Verification (CSFV) program. In our teaming strategy, we sought to assemble a creative ensemble of free spirits that integrated comprehensive skills in videogame design and development with practical verification tool builders and insightful theorists in software formal methods. CHEKOFV, the chosen name for our group, stands for *Crowd-sourced Help with Emergent Knowledge for Optimized Formal Verification*. The name also recalls that of Anton Chekhov, the Russian writer and physician who maintained a successful blend of his creative skills and scientific knowledge throughout his career.

As we embarked upon the CHEKOFV adventure, we found ourselves paralleling the narrative experience of the players of *Xylem*, our first fully-functional game (Figure 1). We were beginning to explore distant and unfamiliar territory, searching together for exotic patterns and explanations, and seeking out innovative answers to constantly stimulating puzzles.

Figure 1. The adventure begins

Over the three years of CHEKOFV, we grew to more fully understand and deeply appreciate each other's disciplinary approaches and research methodologies. We profited significantly from the opportunity to learn from each other, as we jointly tackled the key challenges of this demanding program. We also benefited greatly from the experience of collaborating with our colleagues on

the other CSFV teams, as well as the sponsors and administrators of the CSFV program at DARPA and AFRL. And, in particular, our explorations were enhanced by the dedicated gaming enthusiasts who sought out our corner of cyberspace, and spent some valuable time accompanying us on our voyage of discovery. We are looking forward with anticipation to future research activities in this intriguing field, and we are confident that the core CHEKOFV tools and games will continue to be available for players and researchers alike, well beyond the end of the official CSFV program; for more information, see www.chekofv.net.

The culmination of our formal collaboration is this Final Report, which documents our approaches, methods, and results, as seen through the specialized lenses of our respective professional disciplines. Like the verification games and tools themselves, this report is an integrated creation that merges contributions from all CHEKOFV–ians over the life of the project. In that sense, it forms the cumulative ensemble of all our endeavors.

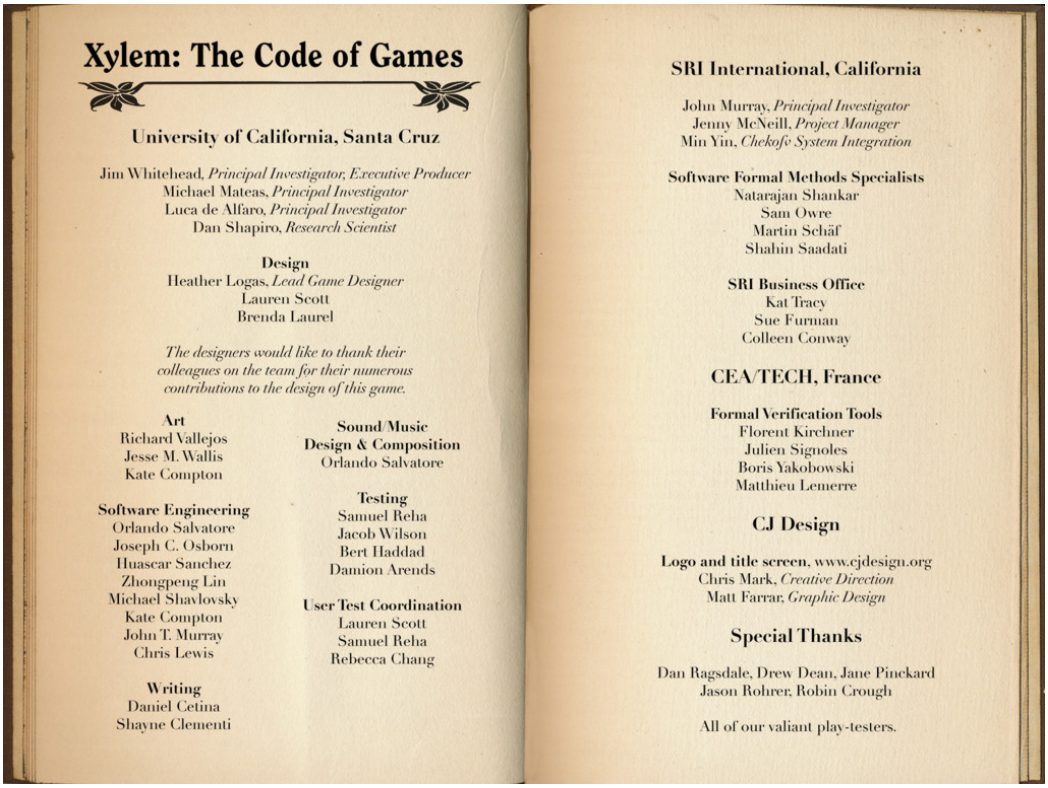
However, the responsibility for actually preparing the materials and editing the Report is a task unto itself, which fell to a focused band of devoted teammates at the end of the core CHEKOFV project activity. We acknowledge in particular the following colleagues for their enthusiastic dedication to this task: Heather Logas, Daniel Fava, and Daniel Shapiro at UC Santa Cruz; Matthieu Lemerre and Julien Signoles at CEA Tech; and David Wilkins, Martin Schäf, and Jenny McNeill at SRI International.

On a broader note, Figures 2 and 3 indicate the institutional affiliations of the primary CHEKOFV team members at the time of their principal participation in our project. As our work progressed, it helped to advance all our professional careers, and many of our colleagues have successfully transitioned to new adventures in new organizations. We wish them well in their new roles, and we look forward to further collaborative adventures and explorations in the future.

John Murray, CHEKOFV Principal Investigator
SRI International
Menlo Park
California.

October 2015

The views, opinions, and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.



Xylem: The Code of Games

University of California, Santa Cruz

Jim Whitehead, *Principal Investigator, Executive Producer*
Michael Mateas, *Principal Investigator*
Luca de Alfaro, *Principal Investigator*
Dan Shapiro, *Research Scientist*

Design
Heather Logas, *Lead Game Designer*
Lauren Scott
Brenda Laurel

The designers would like to thank their colleagues on the team for their numerous contributions to the design of this game.

Art
Richard Vallejos
Jesse M. Wallis
Kate Compton

Software Engineering
Orlando Salvatore
Joseph C. Osborn
Huascar Sanchez
Zhongpeng Lin
Michael Shavlovsky
Kate Compton
John T. Murray
Chris Lewis

Writing
Daniel Cetina
Shayne Clementi

Sound/Music Design & Composition
Orlando Salvatore

Testing
Samuel Reha
Jacob Wilson
Bert Haddad
Damion Arends

User Test Coordination
Lauren Scott
Samuel Reha
Rebecca Chang

SRI International, California

John Murray, *Principal Investigator*
Jenny McNeill, *Project Manager*
Min Yin, *Chekofv System Integration*

Software Formal Methods Specialists
Natarajan Shankar
Sam Owre
Martin Schäf
Shahin Saadati

SRI Business Office
Kat Tracy
Sue Furman
Colleen Conway

CEA/TECH, France

Formal Verification Tools
Florent Kirchner
Julien Signoles
Boris Yakobowski
Matthieu Lemerre

CJ Design

Logo and title screen, www.cjdesign.org
Chris Mark, *Creative Direction*
Matt Farrar, *Graphic Design*

Special Thanks

Dan Ragsdale, Drew Dean, Jane Pinckard
Jason Rohrer, Robin Crough

All of our valiant play-testers.

Figure 2. Xylem credits

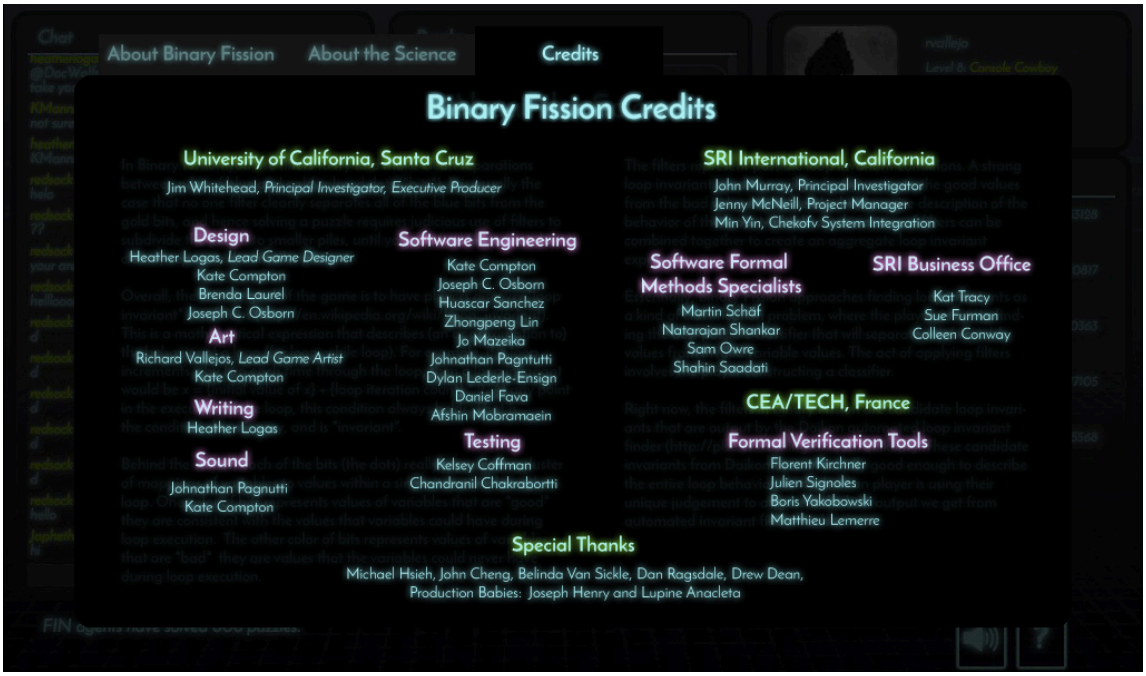


Figure 3. BinaryFission credits

1. SUMMARY

This report describes research undertaken in the CHEKOFV (Crowd-sourced Help with Emergent Knowledge for Optimized Formal Verification) project, under AFRL contract FA8750-12-C-0225. The research was sponsored by the DARPA CSFV (Crowd-Sourced Formal Verification) program, and was carried out from July 2013 to October 2015. The CHEKOFV team was led by SRI International (SRI) and included, as partners, research teams in the Center for Games and Playable Media at the University of California Santa Cruz (UCSC) and at CEA Tech (Commissariat à l'énergie atomique et aux énergies alternatives) in France.

1.1 The Problem

Unreliable software presents major problems throughout industry and government. The direct costs associated with identifying and correcting bugs are enormous on their own, not to mention the related effects that faulty software have on critical systems in cybersecurity, military equipment, medical instruments, and other devices where safety is paramount.

The discipline of formal program verification provides sophisticated techniques for assuring error-free software. The technology of formal verification advanced rapidly in recent years, but it is still dependent upon a limited pool of highly-trained professionals. The more automated techniques generate too many false alarms, which makes it infeasible to move beyond the verification of small modules of software. The underlying motivation for the CSFV program was to achieve both scale and precision in formal verification by leveraging human pattern recognition skills and insights.

From the outset, the CHEKOFV team identified several interrelated challenges and unknowns that needed to be overcome in order to create an effective crowd-sourceable game space for use in this endeavor. Of particular concern was the difficulty of expressing the targeted source code in an easy and intuitive manner for non-experts to grasp and enjoy. An accompanying challenge was the program requirement that the target source code be represented in a form where the original code could not be easily reverse engineered. We also needed to address the problem of providing players with a natural instinctive way to manipulate and refine the initial assertions that are generated by automatic means. Related to this problem were the challenges in representing software behaviors and patterns that elude automatic identification, and eliciting players' insights and observations based on their exploration of the game space.

Furthermore, we wanted to provide some mechanisms that could take advantage of inter-player collaboration/competition, and the sharing of insights among the crowd. For example, we felt that such cooperative activity would become easier when players share a common behavioral view of the problem space. At the same time, they should also have the means to hone their own individual skills, and to develop techniques and craft tools that express their perceptions of the game world activity.

Finally, we recognized the importance that players be properly rewarded for their in-game actions. The game world would need to incorporate an ability to trade, share, and/or exchange tools that will strengthen the mechanisms for structuring contests and cooperation among individuals and groups. An appropriate reward scheme would also recognize the time and effort expended by players in exploring challenges that may ultimately prove unsolvable.

1.2 CHEKOFV Workflow

A high-level overview of the notional CHEKOFV workflow is shown in Figure 4. The core system is conceptually divided into two interlinked subsystems – the Verification Framework (VF), and the Game Subsystem (GS) – which work together to accept verification tasks from a human coordinator and deliver useful, engaging gameplay experiences to a suite of game applications.

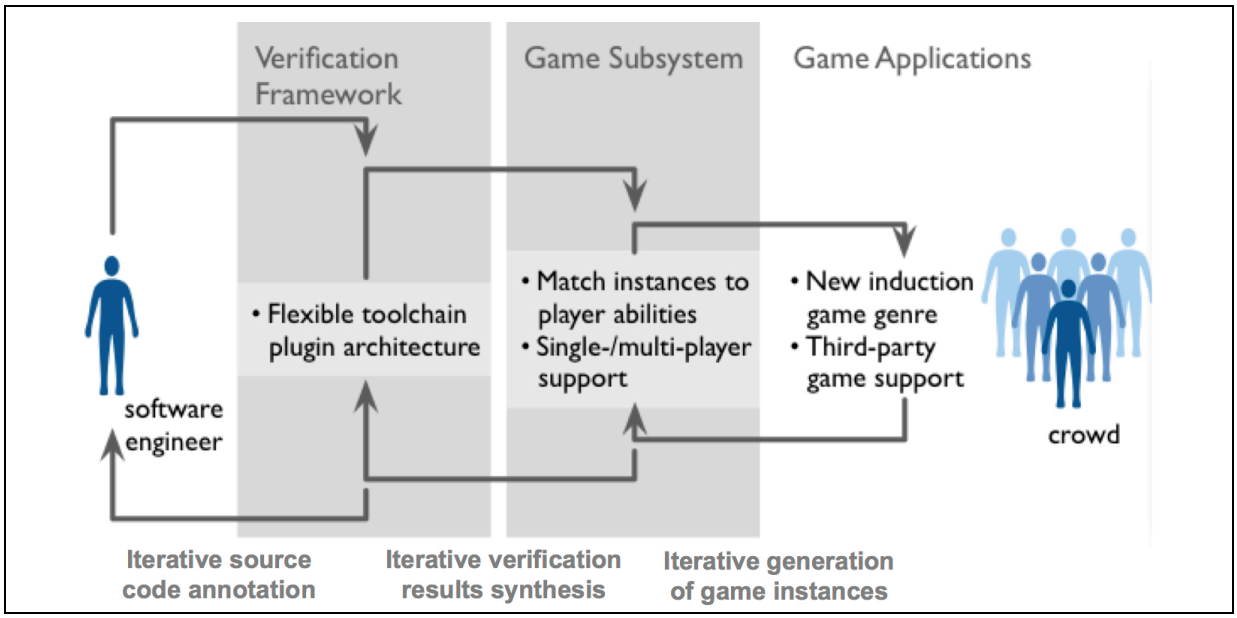


Figure 4. Nested iterative workflow processes for CHEKOFV

The workflow through the system forms a nested series of iterative processes. At the top level, the software-engineer coordinator submits source files to VF for verification and receives updated/annotated versions back. This process is repeated until an optimal annotation state is reached, when the code is ready for compilation. At the middle level of iteration, VF manages the automatic-verification processes, identifies specific verification tasks for crowd sourcing, and passes these to GS. VF also handles the crowd-sourced results analysis and the player rewards process, and issues updated tasks to GS. In the innermost loop, the Game Subsystem generates multiple game instances from the tasks received and distributes them to the suite of crowd-sourced games. The returned results for given code snippets may cause further instances to be generated in an iterative fashion. Upon completion, GS aggregates the task results and returns them to VF. A more detailed view of the architecture may be found in Figure 15 in Section 3.1.

1.3 The Games

In Phase One, the CHEKOFV team developed some initial prototype games to explore the general play space, and then homed in on a comprehensive design and development project for *Xylem: The Code of Plants,* our first fully functional game, which was initially released for the Apple iPad platform. The insights and lessons learned from the deployment of that game in Phase One led to the evolution of the follow-up game *Binary Fission*, which was deployed during Phase Two of CHEKOFV.

Xylem: The Code of Plants is a casual game for players using mobile computing platforms that was intended to appeal to a non-traditional computer-gaming demographic. Set in 1921, the

Xylem narrative centers on a mysterious, newly discovered island. The player, in the role of an explorer and botanist, makes observations about fabulous new plant species by spotting patterns in their behavior. There are numerous levels of difficulty in these puzzles, ranging from simple, though exotic flowers, to very complex plant structures, all of which are previously unknown. The flowers and plants are generated from data about the software source code under verification, and the player's observations of the plant growth patterns become candidate invariants that describe the software's dynamic behavior.

As it transpired, *Xylem* generally attracted a science-oriented audience, rather than the casual puzzle players, which was our original hope. Because the most prolific players of *Xylem* were people who were intrigued by the science aspect of the project, in Phase Two of CHEKOFV, we decided to take a very different tactic with our second offering.

With *BinaryFission*, we focused on addressing the citizen scientist audience with a fun game that allowed multiple players to collaborate on a single verification problem. From the CSFV point of view, the goal in *BinaryFission* is to compose and assemble conjunctive and disjunctive invariants, because this process is difficult for automated systems to produce. The game is structured to offer players pools of previously-generated invariants to use as classifiers that separate sets of data values into good and bad subsets.

1.4 Code Analysis and Verification Process

The project relied heavily on the use of abstract interpretation and the learning of loop invariants in the verification process. The management of the tools and tasks for these purposes was handled by *Frama-C*, an open-source multi-platform verification framework that was developed by our team partner CEA-Tech. *Frama-C* was enhanced to support a dynamic interface with the Game Subsystem, and provided plug-in capabilities for several individual verification tools, as well as adding analysis components that merge the formal verification toolchain outputs with the crowd-sourced results returned by the games. All of these resources were specialized for verifying C language source code, which was at the core of our project.

In the course of the project, we identified several important key techniques to generate progress metrics for software verification. In addition, we believe that *Binary Fission* clearly demonstrates the feasibility of crowd-sourced invariant discovery, and it illustrates the promise of crowd sourcing for other verification and classification tasks. This suggests a valid pathway for expanding the reach and practical application of verification technology in a variety of domains.

2. INTRODUCTION

2.1 Background

Unreliable software presents major problems throughout industry and government. The direct costs associated with identifying and correcting bugs are enormous on their own, and the indirect cost to the economy has been estimated to run to tens of billions of dollars. Faulty software adversely impacts the safety and security of critical systems.

Many instances of security properties can be verified by automated analysis methods based on a coarse abstraction of the program. The cases that are difficult to verify automatically are typically those that depend in some subtle way on the behavior of the software. For example, buffer sizes depend on the size and number of data values copied into the buffer. A miscalculation in the buffer size can lead to a buffer overflow that is easily exploitable through a bad input.

Examples of common software vulnerabilities include buffer overflows, SQL injection, and incorrect authorization and authentication methods. To address problems like these, we need to answer questions about programs and program points such as: Has this user input been sanitized? Is this data value encrypted? Is this pointer value active? Is this buffer access within bounds? Is this resource access authorized? Static analysis tools can answer such questions, but they can miss errors and raise false alarms. These gaps can be plugged by annotating the code and using sophisticated formal verification tools. However, there is a limited pool of highly trained professionals who can use these tools and create these annotations, at considerable cost and effort.

The goal of DARPA's Crowd Sourced Formal Verification (CSFV) program was to achieve economy, scale, and precision in formal verification by using games to leverage the intelligence of crowds. This is an extremely challenging problem. Crowd members are expected to contribute insights without access to the structure or behavior of the software. The crowd-sourced games must be simple and intuitive, while also being enjoyable and even addictive to play. There is also the key challenge of mapping complex verification problems to engaging and intellectually challenging puzzles. The resulting games must be easier for humans than for machines to solve. In particular, they must contribute breakthroughs that elude automated-verification techniques. Finally, there are the design and engineering challenges of building a game infrastructure that integrates verification technology with crowd participation.

With these considerations in mind, we designed the initial CHEKOFV system, to be built around the architecture outlined in Figure 5.

The CHEKOFV system includes two interlinked subsystems – the Verification Framework (VF) and the Game Subsystem (GS). The workflow through the system is summarized by the numbered steps in Figure 5. The VF accepts original source code and uses automated tools to annotate it where possible. A Game Abstraction Layer is used to transfer key code features to the GS, which generates numerous game instances. These are made available to players, who respond with results, which are then synthesized and used for further source code annotation. This updated code may then be recycled through the system or transferred to the CSFV optimized compiler upon completion.

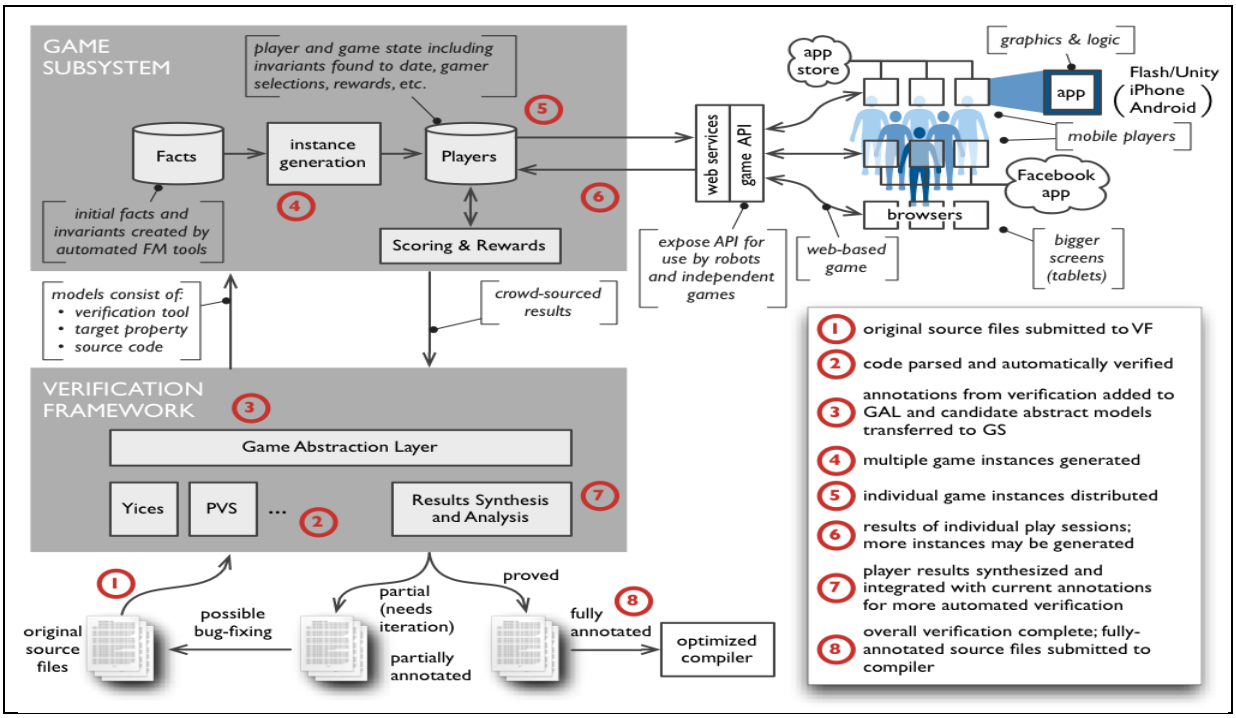


Figure 5. CHEKOFV System Architecture

In the remainder of this introductory section, we first discuss some characteristics of different techniques for software verification, and then describe *Frama-C*, our selected platform for the CHEKOFV Verification Framework [1]. Next, we provide an overview of abstract interpretation and its core role in identifying suitable program invariants. That is followed by our exploration of the casual game landscape, particularly gameplay concepts in the context of crowd-sourcing, which in turn sets the stage for our early design activities for the first CHEKOFV game, *Xylem:The Code of Plants*.

2.2 Analysis of Software Verification Techniques

A preliminary exploration of current software-verification techniques was undertaken in the early stages of CHEKOFV. They are summarized in the following list, and Figure 6 provides a visual comparison of their relative performance characteristics.

- Proof construction based on Hoare Logics in systems such as Spec#, VCC, Verifast, Why, and *Frama-C* using interactive proof checkers such as Coq, HOL, Isabelle, and PVS, and automated SMT (Satisfiability Modulo Theories) solvers such as Alt-Ergo, Yices and Z3.
- Software Model Checking tools such as SPIN, JPF, MOPS, and CBMC that explore a limited portion of the state space.
- Exploration tools like Alloy that examine software behavior on bounded configurations.
- Lightweight static analysis methods like Uno, FindBugs, Coverity, CodeSurfer, ITS4, JIF, Klocworks, and Splint that examine control and data flows.

- Heavyweight static analysis methods based on abstract interpretation such as Astree, Polyspace, and CodeContracts where program properties are constructed as fixpoints on an abstract lattice. Such techniques have been used on the A340 flight control software.
- Dynamic analysis techniques such as *Daikon* that filter out invalid putative invariants, and various approaches for test generation (SAL-ATG, DART, SAGE, and PEX) and runtime verification.
- Predicate abstraction methods underlying SLAM and Blast that construct invariants by considering the behavior of the program on some chosen predicates.

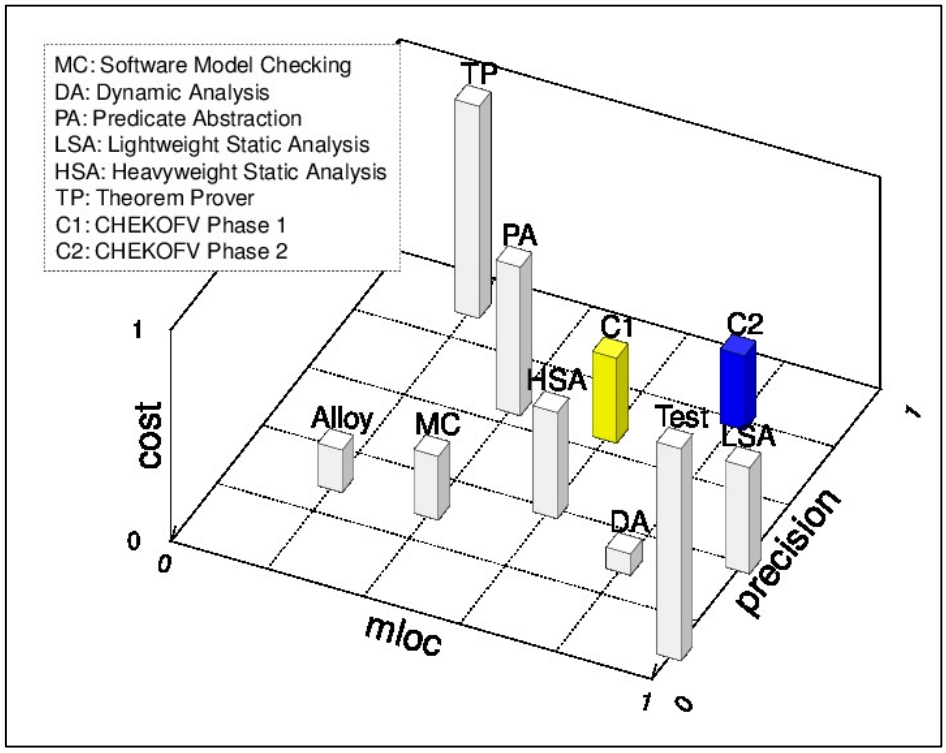


Figure 6. Predicted Performance for the CHEKOFV System (C1 and C2) compared to other verification techniques.

Figure 6 depicts the relative performance trade-offs for different verification techniques on the scales of cost, code space, and precision. For example, theorem proving (TP) offers high precision on modest code sizes, but a high cost. In contrast, dynamic analysis (DA) provides low-cost coverage of larger code bases, but sacrifices precision in the process.

For CHEKOFV Phase One, the optimal verification strategy centered on building upon heavyweight static analysis and using crowd sourcing and abstract interpretation to increase precision, as indicated by performance bar C1 in Figure 6. In Phase Two, the plan was then to drive up the code coverage capability, as shown by performance bar C2, which reflected the CSFV program goals of transitioning from BIND (~650K lines of code) to the Linux kernel (~18M lines of code).

2.3 *Frama-C* Verification Framework

The key design aspect behind the CHEKOFV Verification Framework was to provide an integrated workflow coordinator that managed the verification tasks of a suite of automated FV toolchains. For this purpose, we leveraged *Frama-C* [2] , an open-source multi-platform verification framework, which was developed by CHEKOFV team partner CEA. *Frama-C* was enhanced to support dynamic, two-way interactions with the CHEKOFV server platform, as well as creating analysis components that combined the toolchain outputs with the crowd-sourced results.

As seen in Figure 7, *Frama-C* already provided plug-in capabilities for several individual verification tools, including Yices, SAL, etc. We adapted the interfaces between *Frama-C* and these tools, to support a fully-automated, iterative verification process. All of these resources were specialized for verifying C language source code.

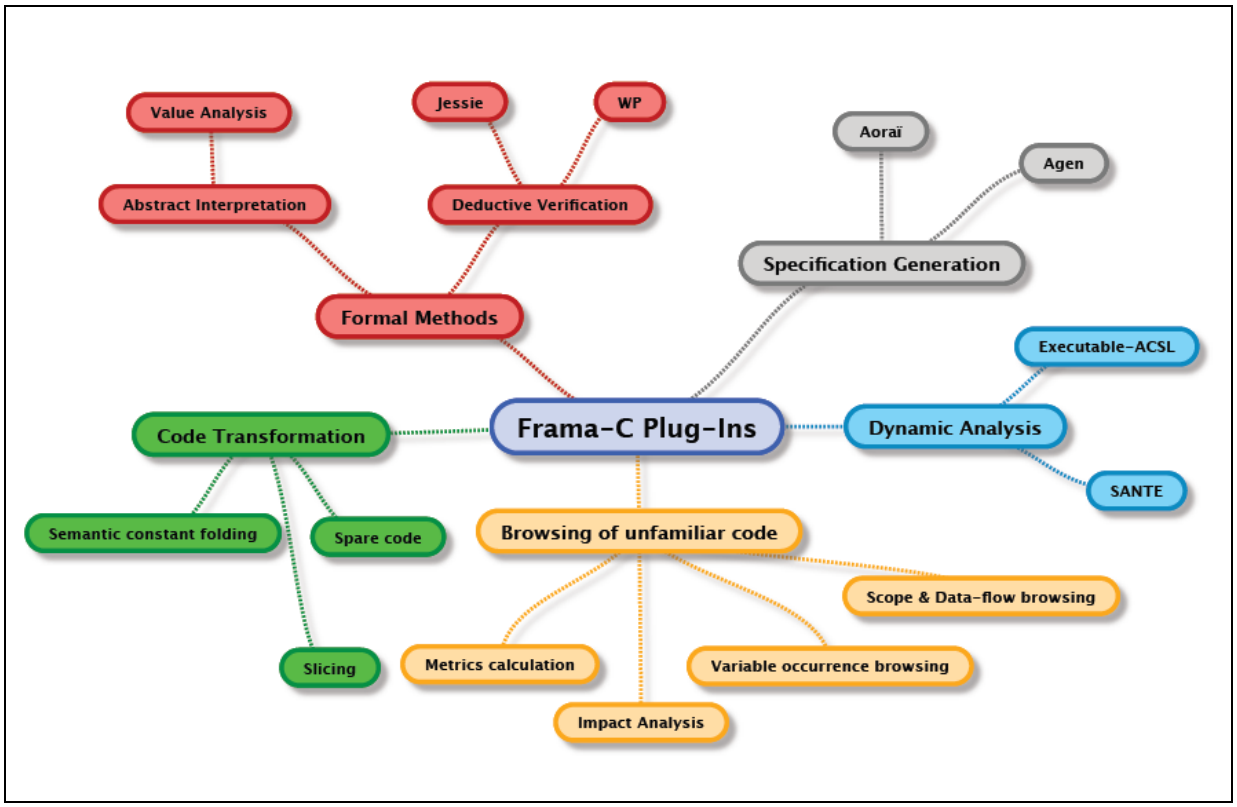


Figure 7. Overall structure of *Frama-C* capabilities and plug-ins

2.4 Abstract Interpretation and Machine Learning

Abstract interpretation [3] is a powerful technique for program verification. Tools like Astrée [4] and *Frama-C* [1] have demonstrated not only that abstract interpretation can prove the absence of run-time errors in real-world C programs, but also that it is commercially viable to do so.

To verify a given program, abstract interpretation approximates the semantics of this program based on monotonic functions. The analysis symbolically executes this program under analysis, keeping a set of possible states at each program point. If an error is not reachable in this abstraction, we have a proof that this error is also not reachable in the original program.

Unfortunately, even if these tools are fully automated, it does not mean that using them is simple. Sometimes, in particular when analyzing looping control-flow, abstract interpretation loses precision and the set representing the possible states of the analyzed program becomes too imprecise. This can result in a large number of false alarms, up to a point where the only option is to abort the analysis. In these cases, a verification engineer needs to step in and provide hints in the form of code annotations or custom parameterizations to help the analysis regain precision. For large programs, writing these annotations can be difficult and time consuming. The process tends to be incremental because an annotation that was used to drive the analysis forward may be insufficient a few statements later. In other words, previous annotations, which were considered sufficient, may have to be revised because they were either too weak or too strong to continue the analysis at a later point in the program. This leads to a labor-intensive process that is also costly because, to provide useful annotations, the analyst not only has to understand the analyzed code, but also the details of the abstraction used by the verification engine.

To lower the cost of applying abstract interpretation, we have seen a new trend of using machine learning to identify likely invariants. The idea is to collect two sets of concrete program states that are either part of a successful execution (good states) or failing executions (bad states), and use machine learning to find a classifier that separates those sets. Approaches such as *Daikon* [5], ICE [6], and work by Sharma et al. [7][8][9], have successfully demonstrated that machine learning can be used to learn likely invariants. This differentiates it from widening, which – although commonly used to generalize program behavior by expanding potential bounds – does not provide the generalization guarantees that machine learning offers.

However, there are limitations to using machine learning for finding likely invariants. For example, collecting good states and bad states is expensive (if it were easy to enumerate them, we would not need abstraction) and thus the machine learner has to operate on a small data set, which in turn increases the risk of over-fitting. Machine learners also have a tendency to produce large and clumsy invariants that are very difficult for humans to interpret. In addition, machine learners operate on a hypothesis space which allows them to express certain kinds of knowledge and empowers them with the ability to generalize. However, there can be mismatches in the type of representation strength of a classifier and the domain of the program under analysis.

The use of crowd-sourcing in CHEKOFV was intended to complement these machine-learning-based approaches to invariant discovery. A major benefit over machine learning is the fact that invariants are not limited by a particular kernel function or hypothesis space. Instead, we anticipated that a very diverse set of solutions could be obtained from different players. Also, we considered it likely that humans would tend to produce invariants that are readable and, given our natural limitations handling large amounts of data, we felt that humans would be less likely to produce a solution that over-fits.

The obvious problem of crowd sourcing is that it has to run for some time (depending on the number of active players, this may be a long time) before a reasonable set of solutions becomes available. However, compared to the several man-months of effort of verifying a real system, this may still be a cheap preprocessing step. Another potential problem is that human intuition breaks at high dimensions, and the dimensionality of the data to be classified depends on the number of variables in scope at a particular program point. This is why, when designing a verification game, the choices of visualization and data representation are important.

2.5 Related Work on Abstract Interpretation

The problem of finding suitable program invariants is a central part of formal verification research. Striking the balance between an abstraction that is sufficiently precise to prove a property and sufficiently abstract to reason about is what makes program analysis scalable. In static analysis, a variety of techniques exist to infer program invariants, such as CEGAR [10], Craig interpolation [11], or logical abduction [12]. However, these approaches have the inherent limitation that they rely on information generated from the source code of the analyzed program. If the needed invariant is a relation between variables that cannot be inferred from the source code, these techniques must fall back on heuristics or fail to compute an invariant.

The idea of learning likely invariants from program states goes back to *Daikon* [5]. *Daikon* learns likely invariants from a given set of (good) program states by working with a fixed set of grammar patterns. Numerous approaches have used *Daikon*; for example, iDiscovery [13] uses symbolic execution to improve on *Daikon*'s invariants. Similar to our approach, it inserts the learned invariants back in the code under analysis and then uses symbolic execution to confirm or break these candidate invariants. This process generates new states that can be fed to *Daikon* and can be iterated until either an inductive invariant is found, or symbolic execution fails to generate new states.

Sharma et al [8] formulate the problem of extrapolation in static analysis as a classification problem in machine learning. They also use *good* and *bad* states and a greedy set cover algorithm to obtain loop invariants. In a follow up work, a similar algorithm to detect likely invariants using randomized search is described [7]. While our approach is similar in the sense that we learn invariants from good and bad examples, our application is different. Rather than finding accurate loop invariants, we are interested in finding human-readable annotations using crowd sourcing that prevent abstract interpretation from losing precision.

The architecture of our approach strongly resembles the decision-tree learning-based approach of *DTInv* [14]. In fact, the authors of that paper provided their implementation, which we used to test our approach. The key difference between the two techniques is that CHEKOFV uses gamification instead of machine learning to find invariants.

Another popular approach for learning likely invariants is the ICE-learning framework [6]. Similar to *Daikon*, ICE-based algorithms search for invariants by iterating through a set of templates. Unlike *Daikon*, ICE does not discard likely invariants that are inductive. Instead, it checks a set of implications to decide if the counterexample is a new good or bad state.

Predicate abstraction [15] based on abstract interpretation has also been used to learn universally-quantified loop invariants [16] and was implemented in ESC-Java [17]. This approach may require manual annotations to infer smart invariants. It is a 100% correct technique but at the price of precision. Counterexample driven refinement has been used to automatically refine predicate abstractions and reduce false errors [18]. Fixpoint-based approaches have also been studied [2] however they do not explicitly generate bad states, unlike the work we describe here.

An approach to gamify type checking has been presented by Dietl [19], which is based on a different usage of crowd-sourcing in the context of software verification. Their approach uses crowd-sourcing to assist an extended type checker to propagate type constraints in a system that does not yet typecheck. While targeting a different domain than CHEKOFV, they also use crowd-

sourcing in a domain where an automatic solution could be computed but where this solution is unlikely to be adequately interpretable by human users.

2.6 Initial Exploration of Game Space

The key game-design challenge for CSFV is to craft a crowd-sourceable experience that helps elicit useful program annotations, especially loop invariants, from non-expert players in an intuitive and enjoyable manner. Most popular game genres cannot simply be adapted to work for this problem. Role-playing games (RPGs) or fast-reflex games, such as first-person shooters (FPSs), do not readily lend themselves to this core game-design challenge.

We found initial inspiration for invariant finding from puzzle games, an existing genre that can provoke thoughtful slow-paced exploration and play, in move-by-move casual games. Puzzle games also often present problems that, if they were to be solved by computer, would have high computational complexity. However, despite these similarities, most puzzle games are a poor match for the invariant-finding problem.

Some puzzle games, like *Sudoku*, are easier for computers than humans, however, the basic issue is that *in most puzzle games, constraints are hard wired, and given to the player*. Invariant finding is the opposite problem: *the constraints are initially unknown, and the player must determine them via inductive reasoning*.

Computers are not good at inductive reasoning, because rules must be generated out of thin air. Inductive reasoning is challenging for humans too, but we appear to be better at it than computers. Given a sequence of data values, our game players must determine rules or constraints that accurately describe the data values.

During our initial game exploration activity, we identified several existing games that bore some similarity to the activities in our project. The closest effort was *Pex4Fun*, developed by Microsoft Research (pex4fun.com). In *Pex4Fun*, the player is presented with an incomplete snippet of C# source code in an editor window, and is asked to make it consistent with a *hidden* implementation. The player can invoke an automated white-box test case generator for the code snippet, and obtains a behavior comparison with the hidden implementation. The player then iteratively writes C# source code and checks the output until the implementation is correct.

This game has multiple drawbacks: it requires players to be able to read and write source code, and have a basic understanding of black box testing. It also has a limited reward system, and limited connections to social media sites. Despite these drawbacks, the game had been played over 775,000 times at the time of our preliminary exploration, indicating that there is an audience even for explicit programming games. However, the CHEKOFV games could not require knowledge of programming or expose the underlying source code. We also needed to provide an interface that looked and felt like a game, not a programming environment.

The genre of logic induction games were similar in structure to the vision for CHEKOFV games. The best-known examples of this genre are *Eleusis*, *Mao*, *Zendo*, and *Jewels in the Sand*. In these games, a human game master picks a rule (concerning the arrangement of cards, plastic triangles, or plastic jewels), and then players must propose their own arrangements to try and determine the rule. The game master gives match/no match feedback to the players. These examples are primarily table-top games, rather than computer-driven ones. This is easier, because natural language can be used to give feedback to players, and to describe the relationship among items. CHEKOFV games are similar to logic induction games in that they require the player to find a

hidden rule, but differ because players use a novel visual language to specify the hidden condition. Furthermore, the hidden rule is determined by software code, not a human game master.

The successful science crowd-sourcing game *FoldIt* (fold.it) harnesses human game-playing ability to develop improved protein-folding schemes, and even the design of completely new proteins. *FoldIt* adds human intuition into the *physical* interactions of parts of a molecule, and hence involves physical models and reasoning. The data space is uniform, consisting of a wide range of combinations of a known set of atoms. In contrast, our proposed games involve players determining logical constraints among a much wider set of potential data values. Our games are similar to *FoldIt* in that we expect better results from a large number of human game players than from computational methods.

Some of the games used for early inspiration during the preliminary CHEKOFV design activities are illustrated in Figure 8 (images taken from Wikipedia); (left column: Words With Friends, Zendo (see: www.koryheath.com), Pex4Fun (see: http://pexforfun.com/), Jewels In The Sand (see: pweb.jps.net/~sangreal/jits.htm), and Marbledrop (see: https://en.wikipedia.org/wiki/Marble_Drop).

Other disciplines provided insight for our invariant finding games. The field of visual programming languages is one that can provide ideas for how to create visual languages for expressing logical constraints among variable values. The field of algorithm animation can provide insight into the thorny problems of how to represent complex data structures. Psychology has explored human performance on inductive reasoning tasks using multiple theories that explain how people approach the challenging task of finding rules from sequences of data. Insights from this literature, in particular the Wason 2-4-6 task, informed game design decisions and assessment of game players [20].

2.7 Preliminary Game Concepts

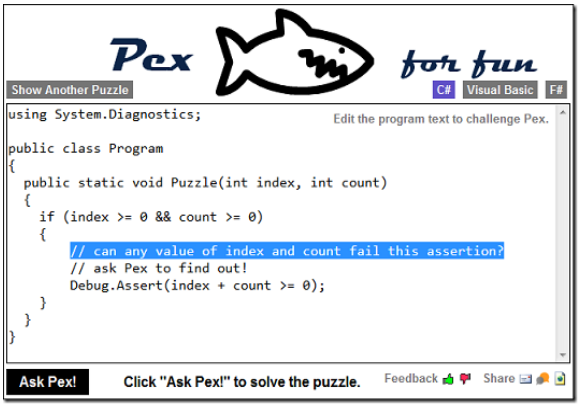
This initial exploration led us to a game concept – first called *CodeBreaker* and then renamed *CyphrSeekr* – that was an instance of an inductive puzzle game (like *Eleusis*, *Patterns II*, and *Zendo*) where players must guess a secret rule from instances.

Figure 9 illustrates some of the preliminary concept mockups for the design of *CodeBreaker*, and Figure 10 shows the corresponding interaction interface as implemented for demonstration purposed. This exploratory game was renamed *CyphrSeekr*, with an accompanying storyline as follows:

> *A newly constructed radio telescope has been receiving perplexing data sequences from various points in deep space. The data is intriguing: when interpreted numerically, it seems that clear, logical patterns emerge. Not language, as we normally think of it, but perhaps a language of numerical relationships. We do not know for sure, and that is why we need your help. The data is presented to you as a series of rows. Try to find the pattern that is common to all of these rows.*

Here, the player would observe a sequence of message states generated by successive executions of a program loop and constructs a formula from a given set of building blocks that best captured the relationships between the values in each state. The game could be played solo against a range of execution traces, or in multi-player tournaments that involve competition and/or cooperation.

See Appendix 1 for more information on this early design concept.



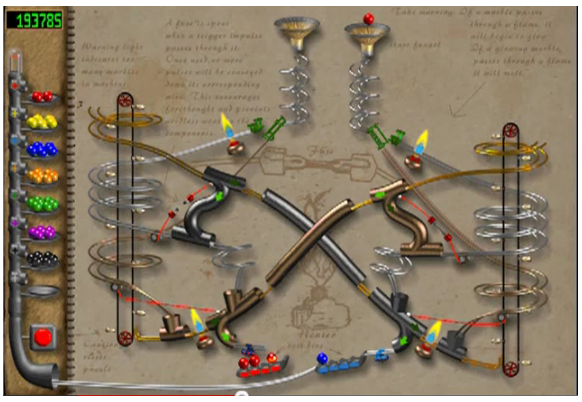


Figure 8. Early inspirations for CHEKOFV game design

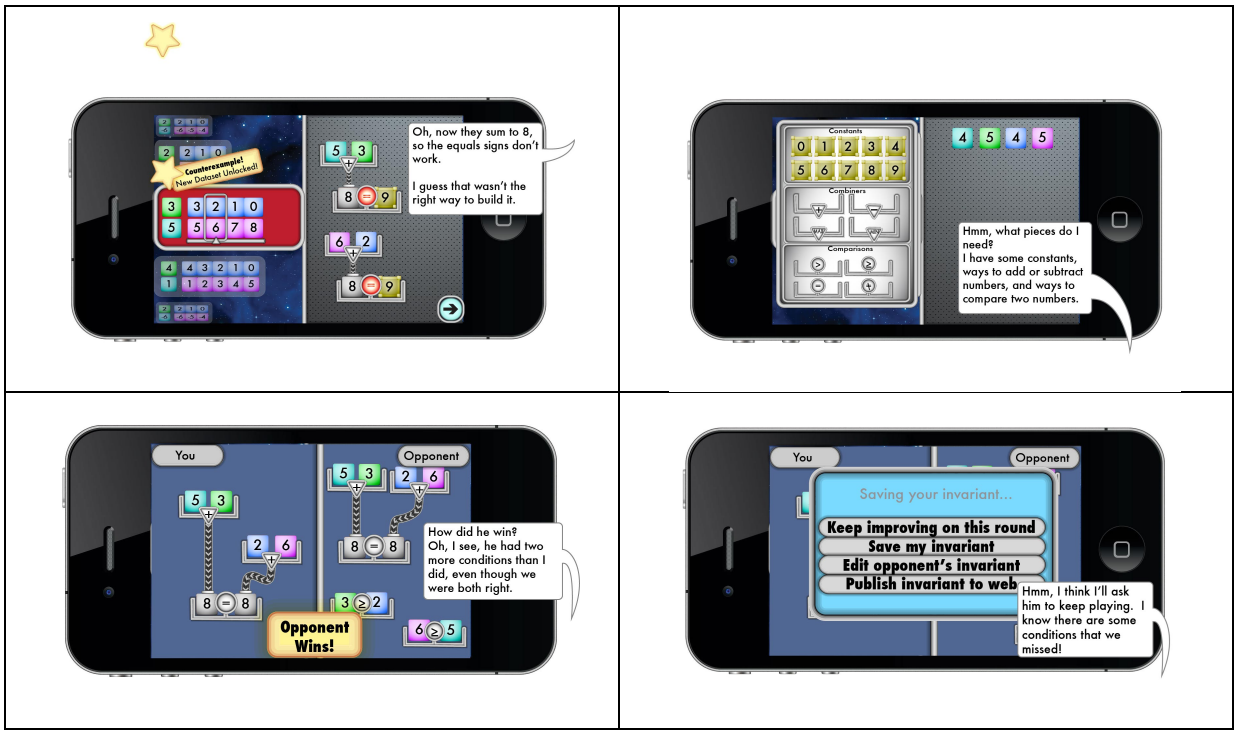


Figure 9. Early concept mockups for *Codebreaker/CyphrSeekr*

Figure 10. Initial *CyphrSeekr* demonstration

Figure 11 illustrates the proposed concept for collaborative/competitive play involving two participants. Starting with the same challenge puzzle, each player builds a candidate invariant which is shared with the other(s) at the end of Round One. For each successive round, the players use this information to build refinements and/or counterexamples, until no further improvements are feasible.

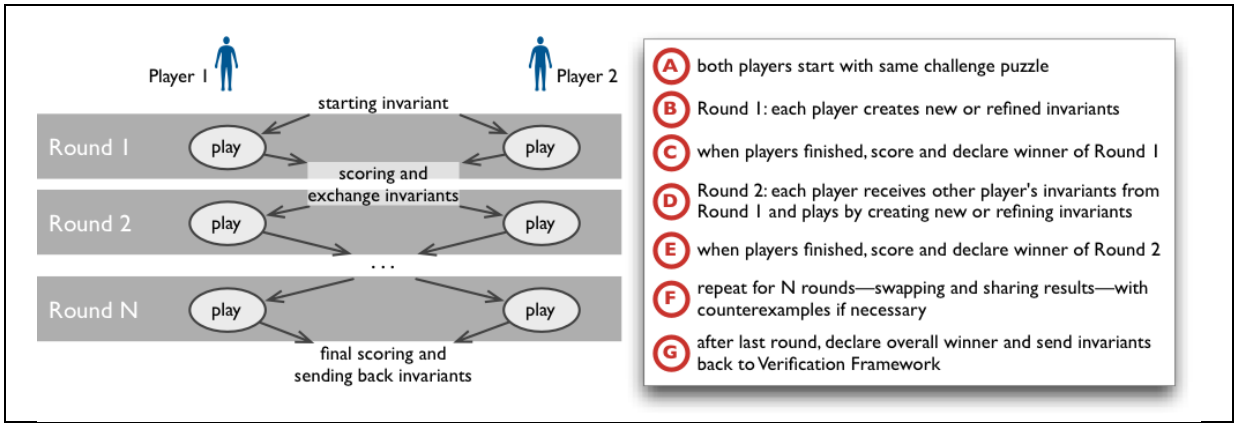


Figure 11. Concept for collaborative/competitive play with two participants

Two additional game-design concepts were also considered during this initial CHEKOFV project activity. One of these, tentatively titled *Invariants vs. Zombies*, was inspired by *Plants vs. Zombies* (www.popcap.com/all-games/plants-vs-zombies), a tower-defense style game where hordes of zombies approach the player's tower along parallel lanes that are defended by plants having varying offensive and defensive capabilities. For *Invariants vs. Zombies*, the player creates invariant conditions to defend against variable-eating zombies, as shown in Figure 12.

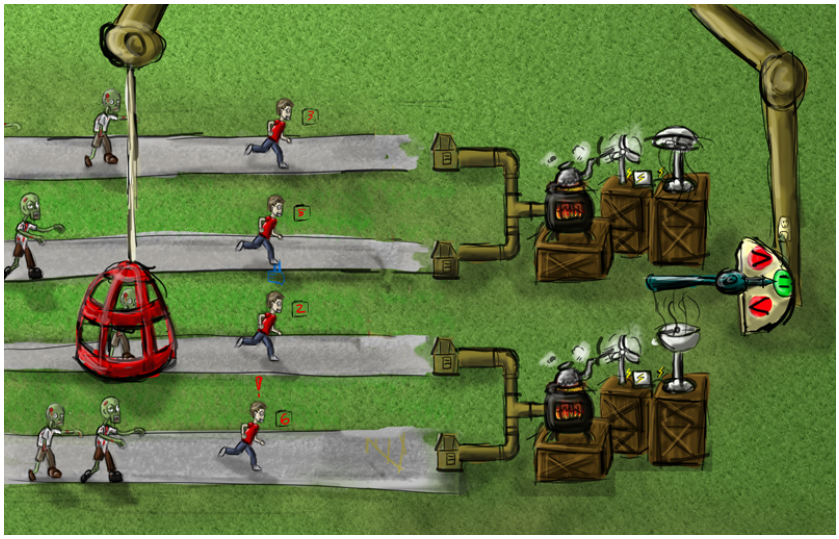


Figure 12. Concept for *Invariants vs. Zombies*

Each variable would run in its own lane, together with a representation of its current value, and the player's goal would be to specify an animated crazy machine (the invariant constraint) that is true across all current and past variable values. In contrast with a more casual puzzle-oriented

experience, this game would be single-player and faster-paced, with players potentially generating more invariant conditions during a play session. While the design of *CyphrSeekr* was oriented toward a slower, more thoughtful style of play, we anticipated that *Invariants vs Zombies* would support more rapid invariant generation on easy and medium complexity software loops.

The second alternate game concept under consideration in the early stages was another collaborative tower-defense game that involved several players cooperatively guarding gates on a road network. Their job was to challenge travelers who arrive and identify them as friends, foes, or neutrals, based on carefully crafted numerical questions about their identification (ID) tags. Tentatively called *IFF (Identification Friend or Foe; Invariants For Free)*, the idea was that collaborative question building in games of this form could be used to cooperatively define software preconditions, post-conditions, joins, loop invariants, and function summaries. Table 1 demonstrates how the key features of an IFF road network would represent the primary artifacts of the software under verification.

Table 1. Relationships between software artifacts and *IFF* road network features

Game feature in *Invariants For Free (IFF)*	Corresponding software artifact
Road network	*control flow graph*
Guards	*control points*
ID Tags	*variable values*
Questions	*assertions*
Friends	*legitimate inputs*
Foes	*generated values violating assertion;*
Neutrals	*generated value not violating assertion*
Change of ID Tags	*executions*

2.8 Concept Transition to *Xylem: The Code of Plants*

A key challenge underlying each of the early design concepts was the difficulty of visually representing variable values, data structures, and other artifacts of software. For example, small values of integer variables can be portrayed using repeated images of some in-game item; thus, a variable value of five can be depicted on screen by five animated frogs, flaming swords, etc. But what if a variable value is in the thousands? Further, lists are pervasive in software and raise even more representational challenges because they need to represent both individual element values, as well as the list as a whole. This implies that the visual theme for depicting the list needs to be consistent with that of portraying the individual list elements.

It became clear that the need to present many different types of data structures meant that the game would require a wide expressive range. With a broad set of supported data structures, the game could cover a wide range of actual program loops in the game, thus increasing the value of the crowd-sourcing approach. This created the challenge of designing a set of data structure visualizations and control flow depictions that held together with a consistent narrative – that of the game.

A key requirement for the overall CSFV program was that players could not have direct access to any of the source code of the target software, which included depicting even small snippets of actual source code. Therefore, the visualization must not only show the game data in a pleasing and consistent manner but also must effectively obfuscate the code such that members of the public could not reconstruct any part of the software under verification.

A further concern in our concept exploration was driven by the desire to design a game that appealed to a large audience, of the sort that might enjoy playing casual puzzle-type games. While our early design concepts were entirely numbers based, the profiles of our potential audience members exposed a discomfort with numbers and math. These various considerations led us to consider the plant kingdom as our core game theme – see Figure 13.

Flowers and plants are a familiar part of most peoples' experience in one way or another, and can be interpreted as friendly and decorative to many of us. Selecting this domain enabled us to develop a screen design that would support the necessary verification work while at the same time creating a narrative space that would be appealing and non-threatening to a wide variety of players. The plant kingdom offered us a coherent visual metaphor that would apply across multiple data structure types, control flow features, and other software artifacts, while at the same time abstracting out or obfuscating the numerical aspect of the player's experience.

Importantly, our intended audience was not anticipated to be overly computer science literate, and this approach enabled us to present complex software-specific information in a way that is

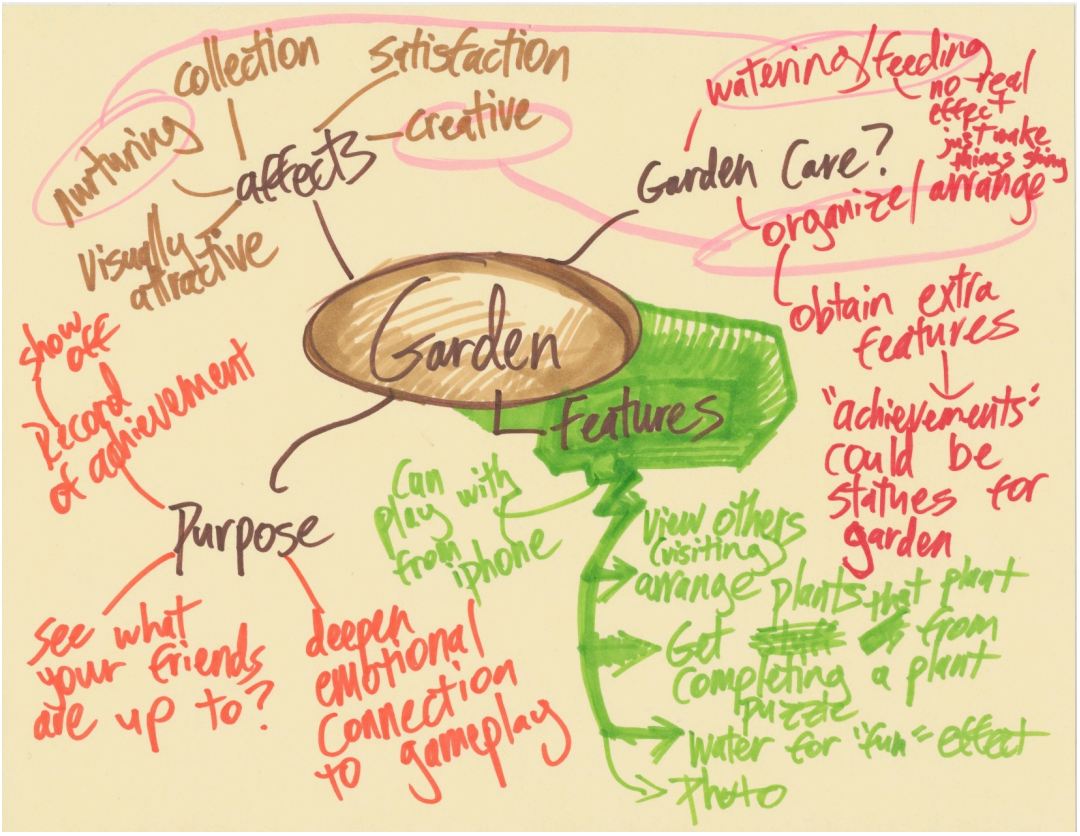


Figure 13. Concept sketch during early exploration of plant domain as a theme

understandable to users who may lack computer science domain knowledge. We also needed to make interaction with intricate software information motivating and clear, without having to introduce specialized notation and terminology. And, in keeping with the expectations of casual game players, this approach could support a consistent fiction within the game narrative.

Importantly, our intended audience was not anticipated to be overly computer science literate, In initial user tests, players could easily distinguish between different plant features as representing different variables. Using plant features that change at different stages of growth also made enough sense to players that it eliminated the need to explain too much backstory and in-game behavior before they could be productive.

Thus was born *Xylem: The Code of Plants*.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we first discuss the overall conceptual architecture for CHEKOFV. We then summarize an early analysis of the BIND source code library, and describe our planned player/problem ranking system. This is followed by the *Xylem* design process, the insights learned from the Phase One deployment of that game, and the evolution of the follow-up game *Binary Fission*, which was deployed in Phase Two of CHEKOFV. Next, we examine the use of abstract interpretation and invariant learning in the verification process. In particular, we present the method of integration of our tools into the crowd-sourcing environment, both for generating game level instances from initial source code and for checking the player-returned results. We end with a brief description of the techniques used to generate progress metrics for software verification. Figure 14 illustrates the introductory screen for both *Xylem* and *BinaryFission*.

Figure 14. Introductory screens for the primary CHEKOFV Phase One and Two games

3.1 Overview of the CHEKOFV system

The initial conceptual architecture and flow sequence for CHEKOFV is shown in Figure 15. The chart illustrates the proposed interconnections between the major internal components of CHEKOFV, as well as the planned system interfaces to other CSFV program elements, such as the TopCoder support platform, as well as to external users and facilities.

Here is a brief walk-through of the process flow:

1. The software developer submits code project into SVN. TopCoder notifies CHEKOFV (as well as other gaming systems) of the new code project.
2. CHEKOFV Facilitate Server (CFS) initiates a preliminary verification session by invoking Preprocessing Tool suite. Developer-provided annotations are checked for validity, and initial vulnerabilities are identified.
3. The *Frama-C* based Verification Framework analyzes source code with different plugin analyzers. The *FUSY* module splits the project into multiple sub-problems and generates loop instances for crowd-sourcing verification.
4. Sub-problems are compared against Pattern DB for possible equivalence matches. If no match, category classification and initial difficulty level are generated for each sub-problem.
5. CFS inserts new sub-problem and its instances into the Game/Problem DB.
6. Game Server generates specific games for each instance and deploys to *Xylem* clients.
7. Player registers with TC web front-end, downloads *Xylem* app, and requests game instance.
8. Game instances can also be made available to robot-players or third-party games as needed.
9. When Player finishes a game, Game Server first performs a simple check and if the check passes, it then notifies CFS of a new candidate invariant, and potentially a player-specified difficulty level.
10. Player is notified of initial score, and optionally updates FB/Twitter, blogs, forums, etc.
11. Incoming candidate invariants from gameplay are checked for equivalence and validated.
12. Verification Framework generates metrics determining the quality of the solution (candidate invariant) or a set of counter examples.
13. Metrics are notified to CHEKOFV Ranking System (CRS), which constantly computes the overall ranking of each player and game difficulty based on current game and player data.
14. Updated CRS rankings are used to revise Player DB, for possible player award notifications, leaderboard updates, etc.
15. All valid candidate invariants generated by the Game System are merged and processed with other candidates of the same sub-problem.
16. Updated ratings of all solutions of the same sub-problem passed to CFS for DB updating.
17. Applying behavioral analytics to in-game player activity logs provides additional insights into potential invariant-generation processes and extends the assessment of player expertise.
18. Current results of all sub-problems are combined as partial annotations to the original source code submitted by the developer, and passed back to SVN.
19. Flow analysis results and Preprocessing Tools outputs are merged to identify CWE coverage levels, which are reported back to the developer.

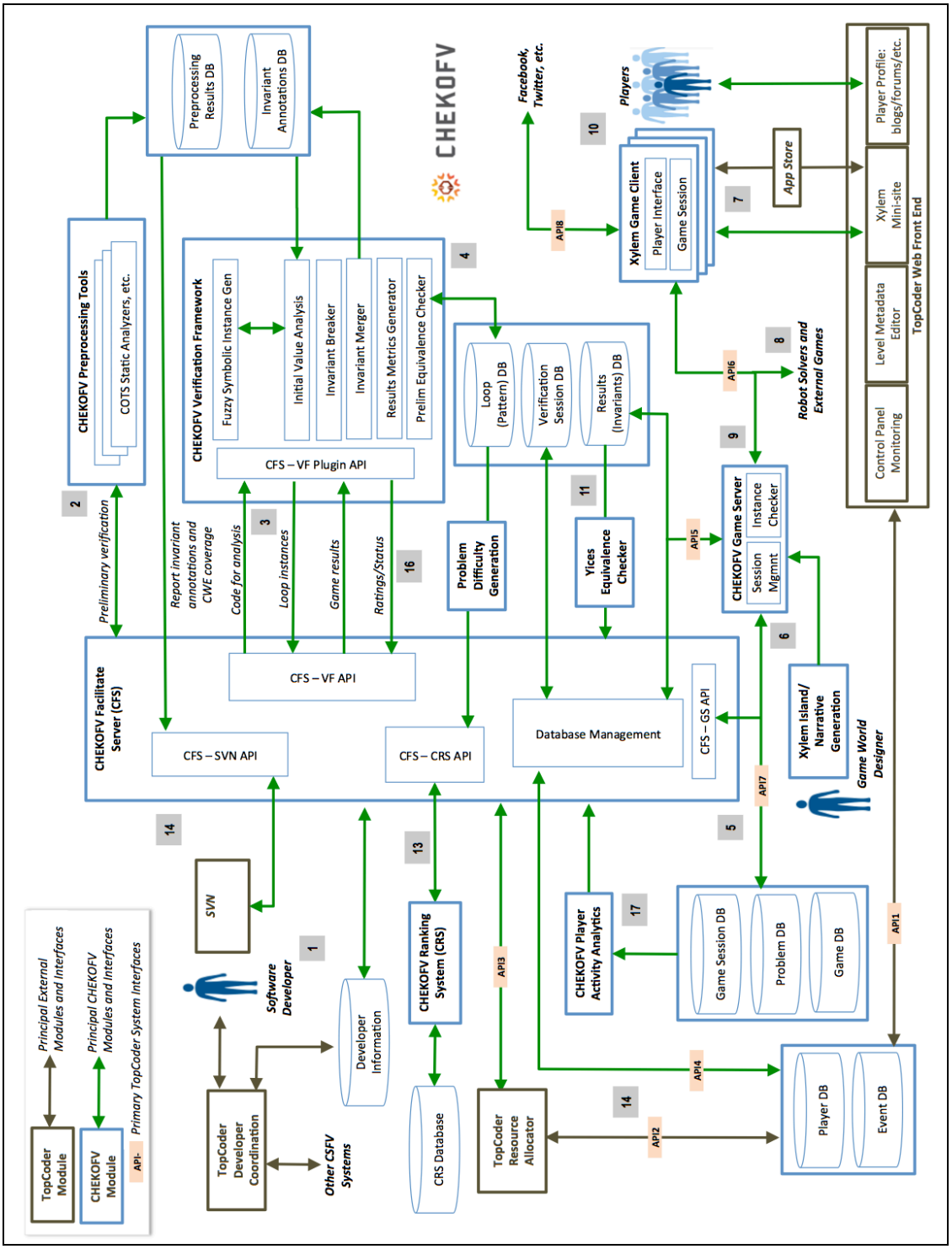


Figure 15. Overall CHEKOFV conceptual architecture

3.2 BIND Library Analysis

Early in the CHEKOFV project, we undertook an analysis of BIND source code. Specifically, we were interested in determining the number and range of loop types, because our project goal was to identify candidate loop invariants for this code base. Table 2 summarizes our overall findings.

Software loops are not all created equal; they are differentiated by features such as complex conditionals, numbers of internal function/method calls, exit conditions, etc. We hypothesized that by examining a large enough sample from BIND, we would discover logical clusterings containing loops with similar properties; i.e., *markers*. The sample needed to be small enough to be conveniently analyzed, yet large enough to help us detect meaningful patterns, e.g., control breaks, early exit, continuation with next iteration, etc.

Table 2. Summary of BIND source code analysis

Directory	While Loops	For Loops	Total Loops
lib/DNS	1659	651	2310
lib/bind9	3	43	46
lib/export	18	32	50
lib/irs	30	22	52
lib/isc	304	158	462
lib/isccc	8	14	22
lib/iscfg	185	31	216
lib/lwres	83	60	143
lib/tests	13	0	13
Totals	2303	1011	3314

Table 3 summarizes the total number of loop types within BIND/lib. Of these, we randomly selected 100 loops; this sample contained a mix of ***while*** and ***for*** loops. The following ***markers*** were identified, to enable reasoning about the loops' internals:

- Update (U). e.g., result = ISC_SUCCESS
- Control Break (CB). e.g., if(condition){ UPDATE }
- Early Exit (EE). e.g., break, return DATA, goto LABEL.
- Continue with next Iteration (CN). e.g., continue
- Inner Loops (IN).
- ANY (A).

This analysis showed seven classes of ***While*** loops and six classes of ***For*** loops in BIND/lib; see Tables 3 and 4.

Table 3. Summary of *While* loop patterns found in BIND.

Class	Pattern	Count
W1	(U, CB)	9
W2	(U, CB, U)	4
W3	(CB, U)	4
W4	(U)	9
W5	(U, IL, U)	5
W6	(U, CB, EE, IL, U|EE)	5
W7	(U|A,EE|CN, CB|U|A)	4

Table 4: Summary of *For* loop patterns found in BIND.

Class	Pattern	Count
F1	(U, CB)	18
F2	(U, EE)	7
F3	(U)	10
F4	(A, INNER_LOOP, A)	7
F5	(U, CN, CB, U, CB)	4
F6	(U, EE, CN, U)	3

Additional clustering can be found by using shared data types, as noted in Table 5. Further details of this analysis may be found in Appendix 2.

Table 5. Most commonly-occurring data types and structures in BIND *While* and *For* loops

While loop classes		***For*** loop classes	
Commonly-occurring data types	**Commonly-occurring data structures**	**Commonly-occurring data types**	**Commonly-occurring data structures**
• *int* • *isc_boolean_t* • *unsigned int* • *NULL* • *char *buffer*	• *isc_region* • *isc_mem*	• *unsigned int* • *NULL* • *char *buffer* • *int* • *isc_uint16_t*	• *cfg_obj* • *isc_log* • *dns_acache*

3.3 CHEKOFV Ranking Subsystem (CRS)

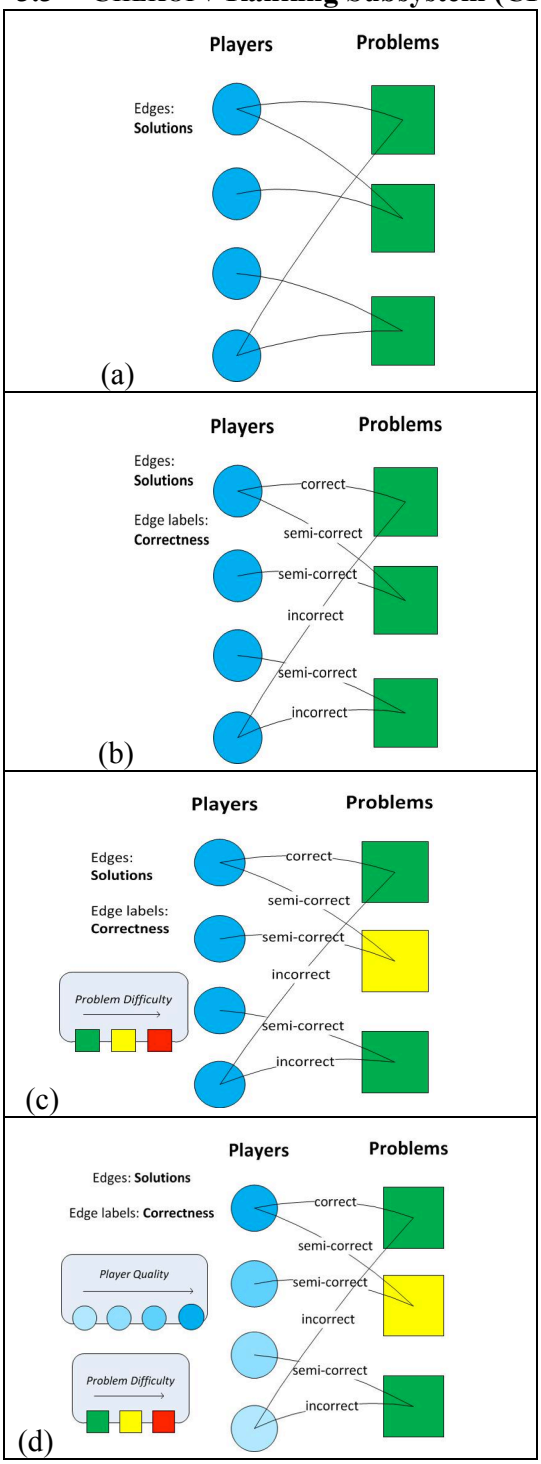


Figure 16.
CHEKOFV Ranking Subsystem

Early in the project, a preliminary reputation-management subsystem was implemented, to support the ranking of players based in part on the quality of their proposed solutions. It involved building a bipartite graph G=(U,V,E) that linked problems U, players V, and solutions E.

The scores are iteratively updated using link analysis and a weighted HITS-based algorithm on the graph [21]. HITS algorithms formulate the notion of authority or reputation, and are based on the relationships between sets of relevant authoritative nodes and the hub nodes that join them together in the link structure. The formulation has connections to the eigenvectors of certain matrices associated with the link graph, and these connections in turn motivate additional heuristics for link-based analysis.

The application of this approach to CRS is as follows: in Figure 16 (a), the graph edges are initialized as players are assigned problems, then (b) the edges are labeled with preliminary scores for the solutions generated. Then (c), the problems are separately indexed by difficulty, such as the number or range of software variables involved, and (d) an assessment of player quality is achieved.

A variety of APIs are provided for managing the CRS. These included adding and updating problem information, and player data including the player's level in the game, their assessed ability, how long they have existed, etc. Solution data to be transferred included: correctness of solution, duration of solution attempt, player_gave_up, start and end times when game was played, the game platform used, etc.

In the long term, and with sufficient data, the CRS was expected to help achieve a consensus on player quality, which would allow more selective challenges to be offered to those players. Although CRS was not fully deployed for CHEKOFV, we believe that a weighted HITS-based approach can provide a useful way to manage a variety of crowd-sourced tasks and projects.

The full CRS API may be found in Appendix 3.

3.4 Phase One: *Xylem*: the Code of Plants

3.4.1 Initial Game-Design Activity. As noted in Section 1, the design direction for the first CHEKOFV game was to abstract away as much numerical information as possible, with the goal of developing a successful "casual" puzzle experience. The game should be untimed, easily picked up and put down again, portable and versatile, and the overall aesthetic should support an intriguing but relaxing atmosphere. The kingdom of plants fit these criteria; plants are commonplace in everyday experience, and as living things, they have a certain universal appeal as evidenced by Figure 17.

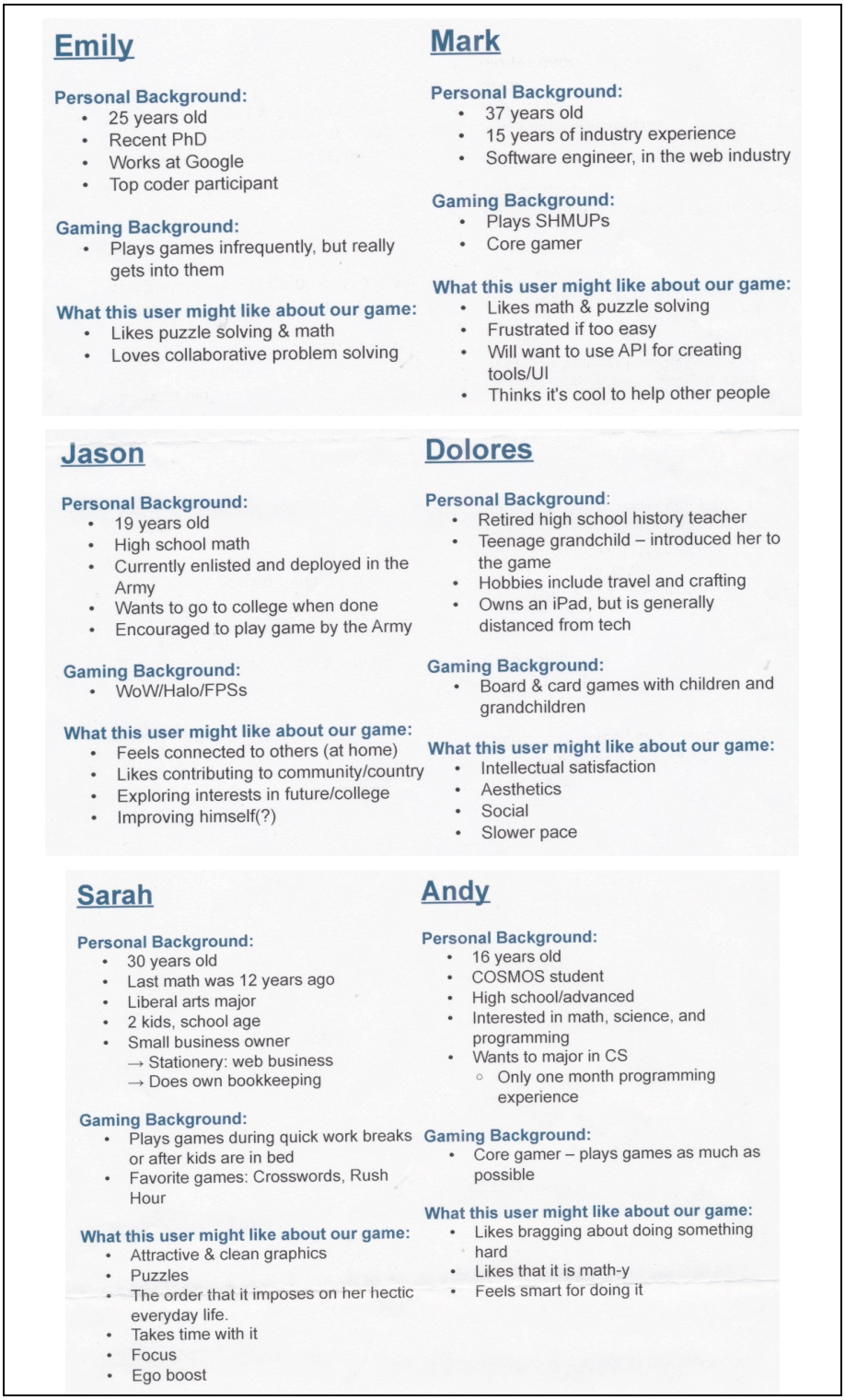
Figure 17. Expressive diversity in plants

From a source-code representation point of view, we noted that plants grow and change over time, as does the data produced by loops. Using features of a plant (number of flowers, number of petals on each flower, number of leaves, etc.) as variables would allow players to observe a pronounced visual change as they moved from iteration to iteration in the loop.

The plant kingdom is also startlingly diverse – a little research turned up plant structures that would map quite easily to different code data structures. Integers can be represented by the number of flowers on the plant, roots with nodules can be used to represent arrays, and trees map directly onto a plant's structure. Plants also provide the opportunity to build a story and gameworld around them in a richer way than boxes and arrows.

Our product was a niche game, but our goal was to create what we called a "low niche" experience. That is, a game that was not exactly casual, but did appeal to a casual-type audience that is comfortable with some level of basic math. We wanted to create a safe, comfortable space for math-puzzle play that would be accessible to as wide an audience as possible. As with crossword puzzles or *Sudoku*, we envisioned a non-stressful brain-exercising experience done while relaxing in a comfortable chair. An unwinding activity for someone who likes to stimulate their brain. This direction set the stage for nearly all of our design decisions.

A critical task in the early design was to figure out just who the audience would be for this game. We created several fictional user profiles or "personae" [22], each with their own demographics, game playing habits, and indications of what they would enjoy about our game. These fictional audience members (see Figure 18) became our guideposts in the early stages of designing *Xylem*.



Emily

Personal Background:
- 25 years old
- Recent PhD
- Works at Google
- Top coder participant

Gaming Background:
- Plays games infrequently, but really gets into them

What this user might like about our game:
- Likes puzzle solving & math
- Loves collaborative problem solving

Mark

Personal Background:
- 37 years old
- 15 years of industry experience
- Software engineer, in the web industry

Gaming Background:
- Plays SHMUPs
- Core gamer

What this user might like about our game:
- Likes math & puzzle solving
- Frustrated if too easy
- Will want to use API for creating tools/UI
- Thinks it's cool to help other people

Jason

Personal Background:
- 19 years old
- High school math
- Currently enlisted and deployed in the Army
- Wants to go to college when done
- Encouraged to play game by the Army

Gaming Background:
- WoW/Halo/FPSs

What this user might like about our game:
- Feels connected to others (at home)
- Likes contributing to community/country
- Exploring interests in future/college
- Improving himself(?)

Dolores

Personal Background:
- Retired high school history teacher
- Teenage grandchild – introduced her to the game
- Hobbies include travel and crafting
- Owns an iPad, but is generally distanced from tech

Gaming Background:
- Board & card games with children and grandchildren

What this user might like about our game:
- Intellectual satisfaction
- Aesthetics
- Social
- Slower pace

Sarah

Personal Background:
- 30 years old
- Last math was 12 years ago
- Liberal arts major
- 2 kids, school age
- Small business owner
  → Stationery: web business
  → Does own bookkeeping

Gaming Background:
- Plays games during quick work breaks or after kids are in bed
- Favorite games: Crosswords, Rush Hour

What this user might like about our game:
- Attractive & clean graphics
- Puzzles
- The order that it imposes on her hectic everyday life.
- Takes time with it
- Focus
- Ego boost

Andy

Personal Background:
- 16 years old
- COSMOS student
- High school/advanced
- Interested in math, science, and programming
- Wants to major in CS
  - Only one month programming experience

Gaming Background:
- Core gamer – plays games as much as possible

What this user might like about our game:
- Likes bragging about doing something hard
- Likes that it is math-y
- Feels smart for doing it

Figure 18. Fictional personae used to guide early *Xylem* design

To attract and retain our identified audience, we developed a rich narrative backstory in the game. This narrative framing was designed to attract players who might not otherwise find raw puzzle games appealing, as it evoked a connection to their experience of an engaging storyline. It thus needed broad appeal without too many complicated narrative trappings that would make the fiction harder to penetrate. An undercurrent of mystery and intrigue was woven into the game to create an additional sense of compulsion to the game while supporting the core gameplay.

Given the nature of the gameplay (examining and stating observations of plants) it made sense that the character would be a botanist of some kind. The game's procedurally generated flowers would not specifically map to existing plant species and discovering new things would be more interesting than examining existing things. Thus, the player's task would be discovering new plant species that had never been seen before. But still unanswered was the setting of the game. Should it take place in space on a newly-discovered planet? An alternative Victorian steampunk world? A Victorian steampunk world on Mars? Alien plant species invading Earth?

All of these options (and more) were examined but found lacking for various reasons. Finally we hit on the idea of setting the game in the early 1920s, on a mysterious, newly-discovered island that has been named Miraflora. This time period was an important one for exploration in the world, and at the same time the public's imagination was full of pulp adventure and lost lands. The game's story of botanists flocking to a newly discovered island fit with the themes of the time period well. At the same time, this fictional framing did not require too much backstory for players to potentially get mired in and because the theme was not overtly science fiction or fantasy would appeal to a much broader audience. Figures 19 and 20 illustrate some of the narrative art assets used in *Xylem*.

SAN FRANCISCO, TUESDAY, JANUARY 22, 1921

MIRAFLORA RIPE FOR THE PICKING.

Green Rush Begins on New Island.

California, Jan. 22 – Missing fishing vessel, the Indefatigable, has finally returned after several weeks of being lost at sea, and those aboard bring word of a mysterious new island found in the middle of the ocean booming with never-before-seen plant life.

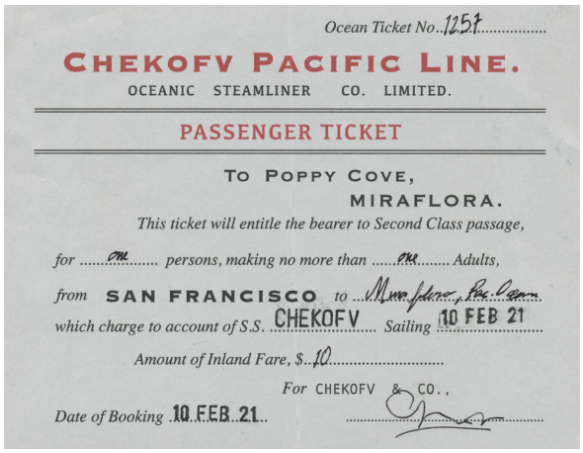
The Indefatigable, owned by local fishing company Logan & Larston, went missing early December when it failed to return from a routine voyage. But on January 21st, a ship matching the description of the missing trawler was seen floating off the coast of southern California. All crewmembers have since been accounted for, and along with their pristine health arrived with stories of an uncharted island dubbed "Miraflora" by the ship's captain, Jameson Salvatore.

"I've crossed the Pacific thousands of times on that route," said Captain Salvatore. "And I've never seen anything like it. The island seems to have just come out of nowhere."

Even the deckhand couldn't put a lid on his excitement over Miraflora's sudden appearance.

"I thought I was either dreaming or crazy when we found it," said Nathaniel Pennywash, a longtime worker on Captain Salvatore's ship. "I got the heebie-jeebies

Figure 19. Supporting introductory narrative for *Xylem*



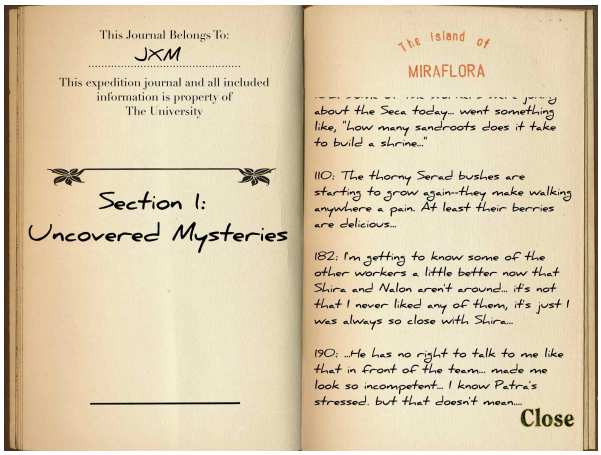


Figure 20. Narrative art assets in *Xylem*