Crowdsourcing Program Preconditions via a Classification Game

Daniel Fava
University of California
Santa Cruz
dfava@soe.ucsc.edu

Dan Shapiro
University of California
Santa Cruz
dgs@ucsc.edu

Joseph Osborn
University of California
Santa Cruz
jcosborn@soe.ucsc.edu

Martin Schäef
SRI International
martin.schaef@sri.com

E. James Whitehead Jr.
University of California
Santa Cruz
ejw@soe.ucsc.edu

ABSTRACT

Invariant discovery is one of the central problems in software verification. This paper reports on an approach that addresses this problem in a novel way; it crowdsources logical expressions for likely invariants by turning invariant discovery into a computer game. The game, called Binary Fission, employs a classification model. In it, players compose preconditions by separating program states that preserve or violate program assertions. The players have no special expertise in formal methods or programming, and are not specifically aware they are solving verification tasks. We show that Binary Fission players discover concise, general, novel, and human readable program preconditions. Our proof of concept suggests that crowdsourcing offers a feasible and promising path towards the practical application of verification technology.

1. INTRODUCTION

A key problem in software verification is to find abstractions that are sufficiently precise to enable the proof a desired program property, but sufficiently general to allow an automated tool to reason about the program. Various techniques, such as predicate abstraction [2], interpolation [21], logical abduction [8], and lately machine learning [25, 30, 13] have been proposed to automatically find such abstractions by identifying suitable program invariants. Each of these techniques provides its own approach for *inventing* suitable predicates, but unfortunately, the space of possibilities is essentially infinite and it is not currently feasible to reliably find such predicates via automated methods.

The human process for finding invariants relies on highly skilled people, schooled in formal methods, to reason from the purpose of programs towards possible predicates. However, this approach has an issue of scale: millions of programs could benefit from formal verification, while there are only a few thousand such experts world-wide. Automated methods rely on search, and expectations to constrain the predicate invention process. White box techniques leverage knowledge about program content to propose candidate invariants, while black box methods search a space of templates (often boolean functions of linear inequalities) using comparatively little knowledge of program structure.

Recent work on classification techniques employ data to constrain predicate invention. Here, the objective is to induce a boolean expression over a base set of predicates that admits "good" program states (inputs that satisfy desired properties encoded as assertions) while excluding all "bad" states (input that violates such assertions on execution). Machine learning methods are well-suited to this task [13, 16, 26, 25]. These techniques output likely invariants that can be tested by static or dynamic analysis methods to determine if they are invariant conditions of the underlying program. The key issue in this approach is generalization; useful invariants are broad statements while classification methods tend to overfit the data. Moreover, the data on good and bad program states necessary to achieve robust generalization is in short supply, as program sampling is itself a hard task.

This paper reports on a classification based system that addresses predicate invention in a novel way; it crowdsources logical expressions for likely invariants by turning invariant generation into a computer game. This approach has several potential benefits:

- It can take advantage of the human ability to extract general predicates from small amounts of data,

- It makes predicate invention accessible to a much larger pool of individuals,

- It allows the crowd to compose unexpected, likely invariants that fully automated methods might miss.

In more detail, the game, called Binary Fission, addresses the subtask of precondition mining; it assumes a set of annotations that encode desired properties, and seeks predicates that imply the annotations hold under program execution. Players function as classification engines. They collectively compose likely invariants by applying "filters" to separate "quarks" in a graphical display, without any specific awareness that they are performing program verification.

Binary Fission is an instance of a growing number of games with a purpose [4, 15, 28], which share the premise that many difficult and important tasks can be advanced by crowdsourcing [24]. As such, Binary Fission is an existence proof for crowdsourcing precondition mining. This paper also demonstrate that it is effective. We claim that:

- The crowd can employ Binary Fission to compose likely invariants for non-trivial programs.

- Binary Fission influences the crowd to produce likely invariants that are also program invariants.

- Binary Fission influences the crowd to produce program invariants that are non-trivial, reasonably general, and human readable.

In addition, we show that the invariants produced via Binary Fission are novel relative to the output of DTinv [16] (a related, fully automated classification system).

The following sections describe our approach and results. We begin by framing this effort against related work, and introducing Binary Fission. Section 4 discusses our methodology for assembling crowdsourced likely invariants from player contributions, extracting program invariants from that set, assessing the quality of crowdsourced results. Section 5 introduces the domain program we examine for preconditions, and Section 6 presents results obtained with Binary Fission. Section 7 discusses the source of power behind these results, while Section 8 examines threats to validity. We end with concluding remarks.

2. RELATED WORK

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 [2], Craig interpolation [21], or logical abduction [8]. 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.

As an alternative to static invariant discovery, we have seen an increasing activity in research on data driven approaches. A pioneer in this field is Daikon [10, 9, 11] which takes a set of good program states as input and applies machine learning to find an invariant that describes all states in this set. More recently, several approaches have extended this idea of inferring invariants from traces [22], some of these techniques also consider sets of bad states that should be excluded by a likely invariant [13, 16, 26, 25]. The benefit of machine learning or data driven approaches over static invariant discovery is that these approaches can search for invariants in a larger space and discover invariants even if they are based on relations that are not easily inferred from the program text. This paper explicitly compares results obtained by Binary Fission with results obtained through DTinv [16], which provides a classification model that is very close in spirit to our work.

Since Binary Fission is a crowdsourcing game, it can viewed as a game with a purpose (GWAP) [29]. Since Binary Fission involves people performing work that computers cannot, it can also be viewed as a form of human computation (see [1] for design issues concerning motivation and evaluation in this context, and [20] for a survey of crowdsourcing in software engineering). Since Binary Fission uses a game reward system to motivate players, it is a form of gamification [6]. We view Binary Fission as a deeper application of game design principles than typical in gamification efforts, as it simultaneously makes a hard science problem playable, and disguises the core activity more than typical human computation tasks.

Overall, the idea of building crowdsourced games for hard scientific tasks has shown enough promise to motivate a large investment in this area. Binary Fission was developed as part of the Crowd Sourced Formal Verification (CSFV) program, funded by DARPA in the United States. This program has resulted in the creation of ten games focused on the intersection with formal software verification [7, 18, 12]; a summary of the games developed in this program can be found in [5], and many of the games can be played at `verigames.com`.

3. BINARY FISSION

Binary Fission is a game for crowdsourcing program invariants. It is one of several recent efforts designed to exploit the "wisdom of the crowd" by transforming hard scientific problems into games [4, 15, 28]. Binary Fission is intended for players with no expertise in formal verification methods, and the players are at most peripherally aware that they are solving verification problems through game play.

The design for Binary Fission was inspired by the need for a broadly accessible mechanism for finding invariants. The game employs a classification metaphor. At the technical level, it inputs a program annotated with postconditions, a set of predicates relating program variables, and two sets of initial program states (each state is a vector of variable values), where "good" states satisfy the assertions, and "bad" states violate those assertions on program execution. Each Binary Fission player employs the available predicates to find a classification tree that separates good data from bad. This tree defines a logical formula representing a likely invariant.

At the game level, Binary Fission hides the nature of the program, data, and predicates from the player. Instead, the game's graphical interface presents problems to players in abstract form. As shown in Figure 1, it depicts program states as spheres, called quarks, colored blue or gold depending upon whether the state is *good* or *bad*. The quarks are initially mixed together inside the nucleus of an "atom." The player's goal is to separate the gold from the blue quarks using a set of filters (corresponding internally to predicates), which are capable of splitting the atom's nucleus.[1] The wheel around the quarks represents these predicates as pentagons. Each filter evaluates to true or false when it is bound to a given program state. Mousing over the wheel of pentagons displays the results of applying the associated filter to the program state; quarks bunch up on the left if the filter evaluates them to true, and they move to the right if the fil-

---

[1] We use the terms predicate and filter interchangeably throughout the paper.

ter evaluates them to false. Different filters create different splits, and the player's job is to decide which filters to apply, and in what order.
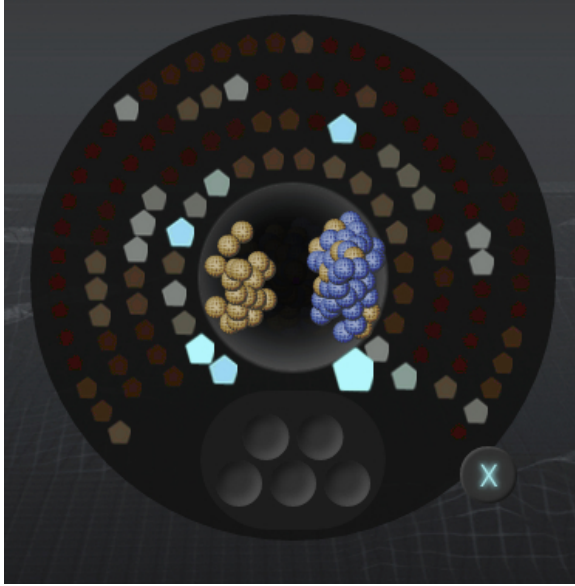

Figure 1: Binary Fission representation of a nucleus with blue and gold quarks surrounded by filters

By mousing over pentagons, the player quickly sees the effect of many filters on the different quarks. The player clicks on a filter once she finds one that she would like to apply. That action splits the "atom" into two child nodes. The left child contains all states from the root that satisfy the predicate, and the right child contains states that falsify the predicate. The recursive application of this process on the left and the right child creates a decision tree as shown in Figure 2.
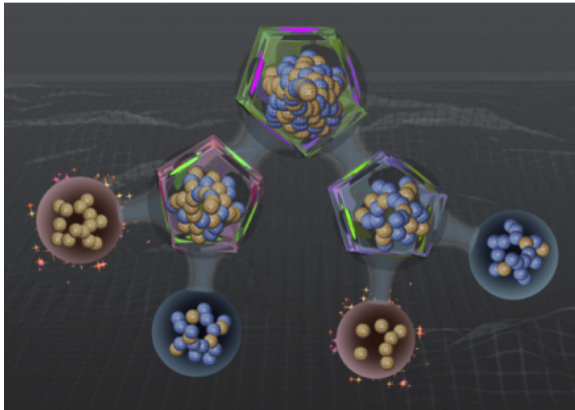

Figure 2: Sample decision tree built by a Binary Fission player

Binary Fission imposes a five level depth limit on player generated classification trees. This bounds the complexity of the resulting classifiers, and limits the screen real-estate required for display (a necessary concern in game design). Binary Fission also provides a scoring function (shown in Equation 1) that influences players to create leaf nodes com-

posed purely of good, or bad program states (where the pure good nodes have special utility for defining likely invariants).

$$N \times \sum_{i \in \text{leaf nodes}} \left(purity_i^A \times size_i^B \right) \qquad (1)$$

Here, $purity$ is the maximum over the percentage of good states and the percentage of bad states in the node, and $size$ is a count of the quarks (states) in the node. A and B are arbitrary constants. N increases with the count of pure nodes in the solution, and decreases with the maximum depth of the classification tree ($N >= 1$). It influences players to produce as many pure nodes as possible, as early as possible, which is a force towards producing useful, and general descriptors.

Each classification tree produced through Binary Fission is typically partial: some leaf nodes only contain good states, some only contain bad states, while others contain a mixture. In addition, the solutions are idiosyncratic, as the players generally employ different subsets of filters during game play. As a result, the game software combines descriptions of pure good nodes and pure bad nodes across solutions to obtain a consensus view of the likely invariant. We discuss this process below.

4. METHODOLOGY

Our methodology for crowdsourcing precondition discovery repeats the following steps:

1. Express an invariant generation task as a data classification problem.

2. Present the problem to Binary Fission players.

3. Assemble a likely invariant across player solutions.

4. Extract clauses from the likely invariant that satisfy program assertions.

5. Assess utility of the program preconditions found.

6. Assess novelty of the program preconditions found.

Following these steps, we assess the value added by crowdsourcing invariants by comparing the results with the solutions produced via an automated classification technique, called DTInv [16]. The following sections clarify these tasks.

4.1 Expressing Invariant Generation Tasks

The goal of Binary Fission is to aid the discovery of function preconditions in a program under analysis. This is done by searching for combinations of predicates that, when placed at function entry points, will prevent states that lead to abnormal termination. These predicates should not, on the other hand, prevent states that lead to normal program termination from executing. We express these problems as classification tasks by specifying {*good states, bad states, predicates*} tuples. We obtain the state data by running a large set of test cases on the underlying program and monitoring its execution with a debugger. We collect the program state at the entry point of each function, and monitor the program's exit status. If the input state satisfies end assertions and exits normally, we add that vector of program variables to the *good* states. If it violates assertions or causes the program to crash, we add it to the set of *bad*

states. We augment these states by randomly sampling the variable ranges observed in the program test cases, after validating with gcov [27] that the new values exercise the same code paths. We retain these states in a hold-out set for testing the generality of any preconditions found, and do not present them to players.

The objective is to find a combination of predicates that segregate good and bad states. Binary Fission can utilize logical predicates of any kind, obtained from any source, with the caveat that they need to be relevant to the classification task at hand in order to be useful. We generate a base set of predicates by employing the Daikon system [11], which is able to explain regularities in program states by searching a library of structural forms. In particular, we supply Daikon with a small subset of good program states (and separately, a small set of bad states), and collect the candidate invariants it produces. Individually, the predicates produced with Daikon on these subsets of program states are not good discriminators. The job of the Binary Fission player is to find combinations of predicates that, together, are able to distinguish between good and bad program states.

We present each of the {*good states, bad states, predicates*} tuples generated in this way to multiple Binary Fission players who create compound logical statements. Binary Fission has many game levels, each associated with one function of the program under analysis. A Binary Fission level is composed of a subset of the good and bad states derived for a given function, and of predicates whose free variables bind to this program state. As a concrete example, take the algorithm in Figure 3 that computes the quotient and remainder of dividing the numerator `N` by the denominator `D`. To produce a Binary Fission level for `Divide`, we collect all (`N`, `D`) pairs observed at function entry during multiple program executions. The tuples that lead to normal program termination are labeled as *good*. However, the program states that cause a runtime exception (the ones in which the denominator `D` equals zero) will be labeled as *bad*. Predicates over the program states can be any logical expression involving `N` and `D`.

```
1   def divide(N, D):
2     if D < 0:
3       (Q,R) = divide(N, -D)
4       return (-Q, R)
5     if N < 0:
6       (Q,R) = divide(-N, D)
7       if R == 0:
8           return (-Q, 0)
9       else:
10          return (-Q - 1, D - R)
11    return divHelper(D, 0, N)
12
13  def divHelper(D, Q, R):
14      if R < D:
15          return (Q, R)
16      return divHelper(D, Q + 1, R - D)
```

Figure 3: Integer division algorithm

We are interested in finding predicates that will, when placed at function entry, prevent runtime exceptions and post-condition violations. For example, the predicate `N=D` evaluates to true on some good program states (like `N=D=42`) as well as on some bad program states (like `N=D=0`). For this reason, `N=D` is not very helpful at segregating *good* program states from *bad*. On the other hand, the disjunction $D >$

$0 \lor D < 0$ is a useful discriminator because it evaluates to true on all valid inputs to the function and to false when $D = 0$.

4.2 Presenting the Problem to Players

The game starts with *good* and *bad* quarks (program states) mixed together in an atom's nucleus. The application of filters (predicates) on the nucleus splits it into a left and a child node. Through the recursive application of filters, players build a decision tree. Figure 4 depicts this process. In this tree, a player applied predicate P at the root-note, then predicate Q on the left child from the root, and predicate R on the right, thus forming a four leafed tree. Two of the leaves contain only *good* program states (represented by the plus signs), one leaf node contains only *bad* program states (represented by the minus signs), and one leaf remained impure, that is, it contains both *good* and *bad* program states.

4.3 Assembling a Likely Invariant

Each classification tree generated by a Binary Fission player separates program states into a collection of Pure Good, Pure Bad, and Impure nodes (where a Pure node only contains program states of one kind). As shown in Figure 4, a conjunction of predicates that links the root to a Pure Good node describes a set of states that satisfy program assertions, and expresses a likely invariant. A single player solution can contain several such paths. By extension, we define the disjunction of paths to Pure Good nodes across all player solutions as the consensus, likely invariant. This results in an expression in Disjunctive Normal Form:

$$PureGoodConjunct_1 \lor ... \lor PureGoodConjunct_n$$

Note that the individual conjuncts might be drawn from the same or different classification trees. As a result, the conjuncts might not employ the same variables, or be mutually exclusive either as logical statements or in terms of the data they explain.

It is tempting to employ the negation of predicates describing Pure Bad nodes across players instead, since an invariant that excludes Pure Bad states is potentially weaker, and more desirable than an invariant that explicitly admits only good states. However, given a partial classifier, the logical expression $\neg(PureBadConj_1 \lor ... \lor PureBadConj_m)$ includes Impure nodes, and accepts bad states that cannot be admitted by any invariant.

4.4 Extracting Program Invariants

Given a likely invariant expressed in DNF, we use the CBMC bounded model checker [17] to identify any component conjuncts that qualify as program preconditions. That is, if $c_1 \lor c_2 \lor ... \lor c_n$ is a predicate derived from data points from function `myFunc`, we consider each clause c_i for $i \in \{1, 2, ..., n\}$ in turn. We place a check of its negation at the entry of the function as shown on line 2 of Figure 5. We then run CBMC on this modified program. When CBMC encounters the if-statement, it splits the analysis between the two paths. The path in which c_i is falsified *dies* when it encounters `exit(0)`. On the other hand, when c_i is satisfied, the analysis continues and the model checker attempts to find function arguments `args` that will later cause post-condition violations (line 6 of Figure 5). If CBMC cannot find inputs that satisfy c_i and violate the postconditions, then c_i is a precondition of the function. The full Binary
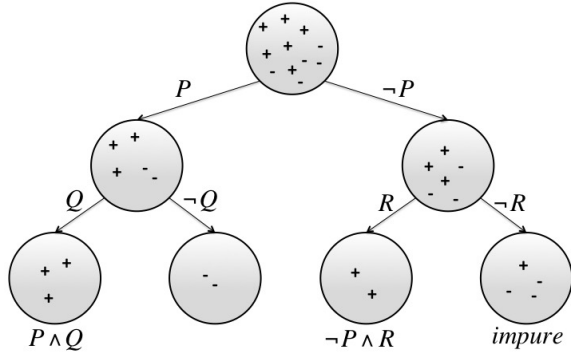
Figure 4: Example of a decision tree produced by Binary Fission. Tracing from the root node to the two pure positive nodes we have $P \wedge Q$ and $\neg P \wedge R$ which form the candidate invariant $(P \wedge Q) \vee (\neg P \wedge R)$.

```
1  def myFunc(args):
2    if (c_i == False): exit(0)
3    # Remainder of the function...
4
5  myFunc(args)
6  assert(postcondition)
```

Figure 5: Pseudocode showing program transformation for discovering function preconditions.

Fission invariant is the disjunct of all clauses that satisfy this test.

4.5 Assessing Invariant Utility

Assuming Binary Fission players discover likely invariants and program preconditions, we would like to understand the usefulness of those expressions. We address this question by measuring the coverage of these invariants against data. The more data explained, the weaker the likely invariant or program precondition, and the more utility it offers for further formal analysis.

Binary Fission relies on a classification technique to separate good states from bad. However, classification methods are prone to overfitting; they must guard against the tendency to explain exactly and only the training data, without providing insight into the general case represented by the data *not* seen. Common defenses include penalizing overly complex expressions considered during classification, and testing against held back data to ensure the generality of the induced function. We utilize both techniques here. In particular, we rely on the Binary Fission scoring function and depth limit to prevent overfitting, and we distinguish training data from test sets.

In more detail, we measure expression generality against a set composed of Good program states. To increase the amount of data available, we interpolate between good states supplied with the program under analysis, and ensure that new states exercise the same code paths as the original states. We measure coverage of likely invariants against the training set, and coverage of preconditions against this new data, which comprises the test set.

4.6 Assessing Invariant Novelty

In addition to assessing the utility of any invariants found,

we examine the conjecture that crowdsourced invariants are novel relative to the results obtained through other methods. If they are novel, it is an indication that crowdsourcing brings some special leverage to the task, and we can analyze the source of that power.

We attempt to place Binary Fission in context by comparing it to other machine learning methods for invariant discovery. Many invariant learners now exist but DTinv is possibly the closest in spirit to our work. DTinv is a fully automated classifier that has been shown to outperform at least six other machine learning methods for invariant discovery [16]. Like Binary Fission, DTinv builds a decision tree from good and bad program states (that preserve or violate end assertions), plus a set of primitive predicates that relate program variables. The key differences are that DTinv builds its own predicates from a basis set (vs importing an arbitrary predicate set), and it constructs decision trees of arbitrary depth that perfectly classify the data into Pure Good and Pure Bad sets (vs the partial classifiers of bounded depth produced by Binary Fission).

We apply DTinv to the same data used in creating game levels presented to to Binary Fission players, and we compare the resulting likely invariants for legibility, generality in terms of data coverage, and veracity as program preconditions. To make the comparisons fair, we pre-process the code of the program under analysis to represent arrays (which DTinv cannot currently consume) as separate variables. In addition, rather than test the DTinv solution as a whole for its status as a program precondition, we transform it into Disjunctive Normal Form and test individual disjuncts as candidate preconditions via the CBMC model checker. This approach is symmetric with our examination of disjuncts describing Pure Good nodes in the partial classifiers output by Binary Fission.

We compare the generality of the likely invariants and preconditions found by measuring their coverage of program states, as before.

5. EXPERIMENTAL SETUP

In order to assess our methodology for finding preconditions, we need to employ some program as the subject of analysis. While Binary Fission can be applied to any program, and accept its inputs from any source, the application to invariant generation imposes constraints. The underlying program must be compatible with automated analysis tools that can generate quantities of good and bad data, and candidate predicates for input to Binary Fission.

We selected TCAS; an aircraft collision avoidance application originally created at Siemens Corporate Research in 1993. TCAS has been a common subject of verification methods [14, 19] and test case generation systems since it was incorporated into the Software-artifact Infrastructure Repository [23].

At the code level, TCAS performs algebraic manipulations of 12 integer variables and a constant four element array. It contains nested conditionals and logical operators; there are no loops, dynamic memory allocations or pointer manipulation. As a result, TCAS admits analysis via model checking – we use the CBMC model checker to test whether potential preconditions found through gameplay indeed exclude states that cause postcondition violations. In addition, TCAS comes with a large set of test data we use to generate good and bad program states. Finally, the program's alge-

braic structure is amenable to analysis by Daikon, which we employ to generate candidate predicates.

TCAS consists of 173 lines of C code split into nine functions. As shown by the call graph in Figure 6, the main function calls an initialization routine before transferring control to `alt_sep_test`, which tests the altitude separation between an aircraft and intruder that has entered its protected zone. TCAS then generates warnings, called "Traffic Advisories' (TAs), and recommendations, called "Resolution Advisories" (RAs), to the pilot. The TAs alert the pilot of potential threats, while the RAs are proposed a maneuver meant to safely increase the separation between planes.
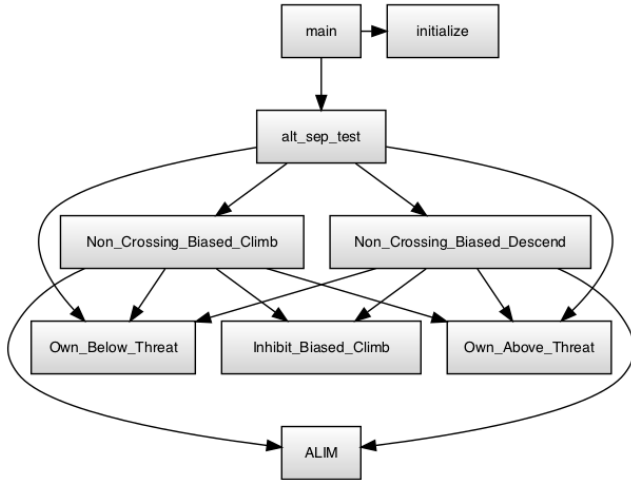

Figure 6: TCAS call graph.

A theory for avoiding aircraft collisions determines when certain maneuvers are safe; these conditions identify safety properties that the TCAS implementation should ideally guarantee. Table 1 illustrates some of these safety properties (reproduced from [3]). For example, the last two entries specify that a maneuver that reduces the separation between two planes must never be issued when the planes have intruded into each others' protected space. These safety properties can be encoded as postconditions of the TCAS program, via assertion statements at its end. The problem of proving the TCAS program safe translates into the task of verifying that the implementation cannot violate these assertions.

5.1 Game Levels from TCAS

We tackle a subtask of the verification process, which is, to find suitable preconditions for TCAS functions. Function preconditions are conditional statements about program variables; if they hold on input to the function, program execution is guaranteed to produce the postconditions that encode desired properties. We define seven precondition finding tasks from the TCAS code. They are to discover preconditions for each of the functions *ALIM*, *alt sep test*, *Non Crossing Biased Climb*, *Non Crossing Biased Descend*, *Own Below Threat*, *Inhibit Biased Climb* and *Own Above Threat* as shown in Figure 6, where those preconditions ensure the conjunction of program postconditions illustrated in Table 1.

We monitor TCAS' execution and collect program state with a Python script driving GDB. We then feed subsets of the program state to Daikon in order to create simple predicates. For TCAS, the set of predicates consists of several hundred boolean combinations of equalities and inequalities among linear functions of 1-4 variables, including max and min operators, numeric thresholds, and explicit set membership tests. Three sample predicates are shown below.

```
Alt_Layer_Value >= 0
size(Positive_RA_Alt_Thresh[]) == 4
Climb_Inhibit <
  Positive_RA_Alt_Thresh[Alt_Layer_Value]
```

Figure 7: Sample predicates inferred by running Daikon on a subset of program states.

From subsets of the {*good states, bad states, predicates*} tuples, we create levels in Binary Fission. The game is available on-line at `http://binaryfission.verigames.com`, and we invite readers to try it. To date, close to one thousand players have generated about three thousand solutions for TCAS problems.

6. BINARY FISSION RESULTS

Following the methodology described in the previous section, we collected crowdsourced solutions for the seven TCAS problems identified in Section 5.1. For purposes of illustration, we discuss the solution for the TCAS function *Non Crossing Biased Descend* in detail, and then summarize across the remaining six examples. We discuss the structure and coverage of the likely invariants found, we identify the valid program preconditions, and we evaluate the generality of these results. We assess novelty through comparison of the Binary Fission and DTinv solutions for the same problem.

6.1 Likely Invariants for TCAS Problems

The consensus solution for *Non Crossing Biased Descend* has 398 disjunctive clauses that represent the Pure Good nodes found across Binary Fission players. Each clause is a likely crowdsourced invariant. Figure 8 illustrates the top three, measured by their coverage over program states. Their content is syntactically similar; each clause is a conjunct of 2-3 primitive predicates (shown as top-level ANDs), where the primitives express numeric equalities and inequalities over multiple TCAS variables. These are non-trivial statements about domain variables, and they appear reasonably general; they clearly do not pick out specific data values. Following the methodology described in Section 4.5, we measure the generality of these expressions by their coverage of the training data; they each explain circa 30% of the good program states. The three likely invariants also appear to be describing a similar truth, as they utilize many of the same variables and terms. As a result, they can describe many of the same states.

The solutions for all seven TCAS problems have a similar structure. Table 2 shows that they contain between 262 and 704 clauses. These solutions are simple collections, and have not been simplified; they can overlap both logically and in terms of the data covered, and their number strictly grows with the quantity of game play.

6.2 Crowdsourced Solution Progress

Figure 9 illustrates the crowd's progress towards finding a consensus likely invariant. It plots cumulative data explained by the crowdsourced solution, as accumulated in de-

	Postcondition	Explanation
If	`Up_Separation ≥ Positive_RA_Alt_Thresh[2]` ∧ `Down_Separation < Positive_RA_Alt_Thresh[2]`	A downward RA is never issued if a downward maneuver does not produce adequate separation
Assert	`result ≠ need_Downward_RA`	
If	`Up_Separation < Positive_RA_Alt_Tresh[2]` ∧ `Down_Separation ≥ Positive_RA_Alt_Tresh[2]`	An upward RA is never issued if an upward maneuver does not produce adequate separation
Assert	`result ≠ need_Upward_RA`	
If	`Own_Tracked_Alt > Other_Tracked_Alt`	A crossing RA is never issued
Assert	`result ≠ need_Downward_RA`	
If	`Own_Tracked_Alt < Other_Tracked_Alt`	A crossing RA is never issued
Assert	`result ≠ need_Upward_RA`	
If	`Down_Separation < Up_Separation`	The RA that produces less separation is never issued
Assert	`result ≠ need_Downward_RA`	
If	`Down_Separation > Up_Separation`	The RA that produces less separation is never issued
Assert	`result ≠ need_Upward_RA`	

Table 1: TCAS postconditions.

```
(not(Other_Capability > Two_of_Three_Reports_Valid))
    and (not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))

(not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and ((Alt_Layer_Value <= size(Positive_RA_Alt_Thresh)-1))

(not(Alt_Layer_Value >= Up_Separation))
    and (not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and ((Cur_Vertical_Sep != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
```

Figure 8: The best three likely invariants measured by Good state coverage.

creasing order of predicate quality (i.e., the number of good program states recognized by the conjunctive predicate associated with each Pure Good node). This figure supports several interesting observations. First, the top 20% of the solutions explain 80% of the data, and this pattern repeats across all TCAS problems. This suggests a statistical regularity in crowd performance, and an uneven distribution of expertise across players. Second, the consensus solution is partial, meaning it fails to explain all the data even after incorporating every player's contribution. This is an expected result, as Binary Fission limits the depth of player classification trees – some truths are simply hard to express in bounded space.

In order to investigate this point further, we employed a greedy search algorithm to construct a classifier for the same problem, over the same primitive predicates. The method used average impurity for scoring splits. When invoked with a depth limit of 5, the resulting partial classifier explained 21 good program states. This splitting metric clearly provided insufficient motivation to distinguish Pure Good nodes early in the classification process that have utility for invariant generation. In contrast, the reward metric employed by Binary Fission clearly influenced players to isolate Pure Good nodes at shallower depths, with the associated benefit for explaining good program states. This pattern repeated across TCAS problems.

We also tested the expressive power of the primitive Binary Fission predicates by invoking the greedy classification algorithm without a depth limit. The result here, and in all 7 TCAS problems, was that the predicates had the power to correctly separate all good program and bad program states. As a result, our statistics on Binary Fission solutions concern the performance of the crowd, not the expressivity of
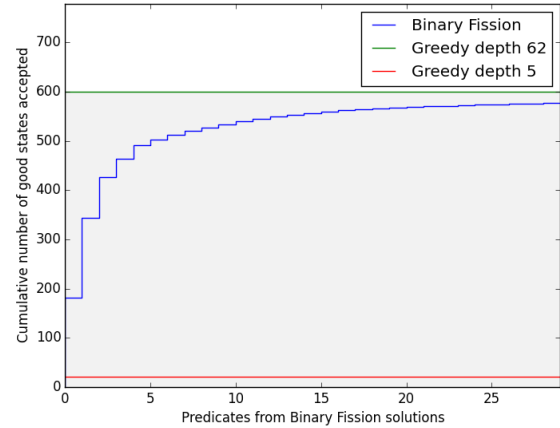

Figure 9: Crowd progress in classifying data points from `Non_Crossing_Biased_Descend`

the predicates at their disposal.

6.3 Program Preconditions Found

We tested the likely invariants generated for *Non Crossing Biased Descend* using the CBMC model checker as discussed in Section 4.4. Of the 398 clauses supplied by players, 16 qualified as program preconditions. That is, if any of these preconditions hold on function entry, the postconditions described in Table 1 hold at program exit. Figure 10 lists the three most general preconditions found, ordered by their coverage over the test set of good program states. These are the first instances of program invariants found by

```
(not(Other_Tracked_Alt > Own_Tracked_Alt))
    and (Up_Separation < Positive_RA_Alt_Thresh[Alt_Layer_Value])

(Other_Tracked_Alt > Positive_RA_Alt_Thresh[Other_Capability])
    and (Down_Separation >= Up_Separation)
    and (not(Up_Separation <= Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and (Other_Tracked_Alt > Own_Tracked_Alt)

(not(Other_Capability == 2))
    and (not((Down_Separation == 800) or (Down_Separation == 600)
                            or (Down_Separation == 500)))
    and (Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value])
    and (not(Other_Tracked_Alt > Own_Tracked_Alt))
    and (Up_Separation < Positive_RA_Alt_Thresh[Alt_Layer_Value])
```

Figure 10: The three best crowdsourced preconditions found.

Function	Clauses from BF	Precon- ditions
ALIM	422	45
alt_sep_test	462	103
Inhibit_Biased_Climb	262	7
Non_Crossing_Biased_Climb	360	14
Non_Crossing_Biased_Descend	398	16
Own_Above_Threat	500	0
Own_Below_Threat	704	6

Table 2: Quantity of Crowdsourced Preconditions and Likely Invariants: A fraction of the likely invariants qualify as program preconditions.

Function	Good states	Total states	%
ALIM	51	95	53.7%
alt_sep_test	424	2000	21.2%
Inhibit_Biased_Climb	59	295	20.0%
Non_Crossing_Biased_Climb	60	295	20.3%
Non_Crossing_Biased_Descend	108	295	36.6%
Own_Above_Threat	0	161	0%
Own_Below_Threat	0	185	0%

Table 3: Testing preconditions' generality by comparing the number of good states accepted versus the total number of good states in the held-out test set.

crowdsourced methods. As with the likely invariants, these preconditions are non-trivial statements about domain variables, here relating the positions and capabilities of aircraft in the sky. For example, the first/best precondition in Figure 10 states that advising a pilot to descend (the function of *Non Crossing Biased Descend*) will satisfy safety assertions when (a) the other plane's altitude is higher, but (b) advising the pilot to climb will result in a vertical separation (*up separation*) that is less than the required tolerance.

Binary Fission players collectively found program preconditions for 6 of the 7 TCAS tasks. None were trivial. Table 2 identifies the quantity of preconditions found for each task, and the numbers are substantial.

6.4 Invariant Generality

Following the methodology described in Section 4.5, we assess the generality of the crowdsourced preconditions found by measuring their coverage over good program states in the test set. Table 3 counts the number of program states explained by for the seven TCAS problems. The best-case scenario is for the precondition to accept all good states. In the case of *Non Crossing Biased Descend*, the aggregate precondition (composed of the 16 clauses reported in Table 2) explains 36.6% of the good program states withheld during the classification task. This corresponds to 2.3% of the good states per precondition clause on average, although the distribution was uneven. Figure 10 shows the best three preconditions for this problem. The first explained 20% of the data, while the second and third best preconditions captured 14% and 9% of the program states in the test set respectively. The net result is that the crowd discovers multiple program preconditions with noteworthy coverage/generality.

6.5 Novelty Relative to the DTinv Solution

As discussed in Section 4.6, we compare the Binary Fission and DTinv solutions for each TCAS problem in order to examine the conjecture that the crowd provides novel insight in the search for program invariants. We compare the legibility and coverage of the likely invariants they produce, as well as their ability to discover program preconditions.

In its raw form, the DTinv solution for *Non Crossing Biased Descend* is a depth 15 decision tree containing 65 primitive predicates that completely segments the good and bad program states. The corresponding logical expression is not human readable (nor was it intended to be). We converted this form to DNF to extract less monolithic likely invariants, and show the top three clauses (as measured by the number of Good states covered) in Figure 11.

It is immediately obvious that these expressions rely heavily on numeric thresholds. This is by design, as DTinv's primitive predicates represent planar cuts in the octagon domain. Although it is an aesthetic judgment, this design appears to make the DTinv statements harder to interpret than the Binary Fission output in Figure 8.

Of the three DTinv expressions in Figure 11, the second overlaps the first, and the third is a specialization of the second. They cover 29%, 16%, and 11% of the Good program states, respectively. It is worth noticing that the single best *likely* invariant found by crowdsourcing (Figure 8) and the DTinv classifier have essentially identical capture, and that the top three employ the same variable set, though in notably different formulas. This is an indication that both systems are after similar insights.

We tested the DTinv solution for *Non Crossing Biased Descend* using the CBMC model checker to determine if

```
(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (2*Up_Separation -2*Alt_Layer_Value <= 801)
and (2*Alt_Layer_Value -2*Down_Separation <= -799)
and (2*Alt_Layer_Value + 2*Two_of_Three_Reports_Valid <= 9)

(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (not(2*Up_Separation -2*Alt_Layer_Value <= 801))
and (not(2*Up_Separation -2*Down_Separation <= -1))
and (not(2*Own_Tracked_Alt -2*Other_Tracked_Alt <= -1203))
and (not(2*Own_Tracked_Alt_Rate + 2*Up_Separation <= 1594))
and (2*Alt_Layer_Value + 2*Other_Capability <= 5)

(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (not(2*Up_Separation -2*Alt_Layer_Value <= 801))
and (not(2*Up_Separation -2*Down_Separation <= -1))
and (not(2*Own_Tracked_Alt -2*Other_Tracked_Alt <= -1203))
and (not(2*Own_Tracked_Alt_Rate + 2*Up_Separation <= 1594))
and (not(2*Alt_Layer_Value + 2*Other_Capability <= 5))
and (2*Other_Tracked_Alt -2*Down_Separation <= 95)
and (not(2*Two_of_Three_Reports_Valid + -2*Positive_RA_Alt_Thresh[3] <= -1481))
and (not(2*Cur_Vertical_Sep + 2*Other_Tracked_Alt <= 1906))
```

Figure 11: The top three DTinv likely invariants.

it contained valid program preconditions. The surprising conclusion is that it did not, either as a whole, or when we tested individual DNF clauses. This pattern repeated across all seven TCAS problems; none of the DTinv solutions contained valid preconditions. In contrast, the crowd, acting through Binary Fission, produced preconditions for 6 of the 7 TCAS problems. As a result, the crowdsourced solutions are clearly novel relative to the DTinv output.

The cause for the lack of DTinv-based preconditions appears to be overfitting; numeric thresholds induced from data are highly likely to break in the presence of a hold-back set, and the lengthy expressions DTinv discovers to explain all the training data have limited opportunity to generalize. In contrast, the more abstract predicate base and 5 conjunct limit imposed by Binary Fission essentially forces players to paint with a broader brush. Players can only produce shorter, more powerful statements, some of which generalize, as shown above.

7. DISCUSSION

This paper has addressed the problem of crowdsourcing program preconditions, under the model that crowdsourcing offers an alternate, and viable method for addressing a difficult task. We have provided an existence proof in the form of the Binary Fission game, and we have shown that crowdsourcing is effective by employing the game to discover program preconditions for 6 TCAS problems. The preconditions are non-trivial, reasonably general (as measured by data coverage on a test set), and human readable. They are also novel with respect to the output of DTinv, which finds likely invariants, but none that qualify as program preconditions on TCAS problems.

There are three sources of power behind Binary Fission: it employs an expressive representation, it relies on the crowd to conduct a thorough search, and the game imposes restrictions on that search that select for general solutions. In more detail, the representational power comes from Daikon, as Binary Fission inputs the highly structured predicates it produces. The game exploits crowd search by collecting and testing the large number of piecewise solutions that players contribute. The game influences the shape of the solution by

limiting classifier depth, and by rewarding discovery of partial classifiers that isolate positive data, which has special utility for invariant construction.

While Binary Fission employs a classification model, improving classification technology is not our goal. Our main point is to introduce crowdsourcing as a promising approach to invariant discovery. From this perspective, the key conjecture behind crowdsourcing is that many non-expert individuals have the desire and ability to provide insight into highly technical problems when they are presented in a suitable form. This conjecture holds for Binary Fission. If it generalizes, related games will provide leverage on additional verification tasks, and crowdsourcing will offer an avenue for expanding the reach of verification technology.

More broadly, Binary Fission suggests that other highly technical tasks will be amenable to crowd-sourced science games. From our experience, the enabling factor is the use of a very clean mechanic for game play (here, classification tree learning).

8. THREATS TO VALIDITY

This paper reports first results from a crowdsourced approach to precondition discovery. As mentioned above, the key points are that crowdsourcing is feasible, effective, and promising as a practical avenue for expanding the reach of verification methods. That said, there are several threats to the validity of these claims, as well as our more detailed results.

First, while crowdsourcing finds preconditions on TCAS, the approach may not generalize to more complex programs. In particular, TCAS is a short, straight line, arithmetic program that lacks pointers, loops, complex data structures, and a range of other language features that complicate the verification task. The counterpoint is that Binary Fission is agnostic to the structure of the underlying program, because it formulates precondition discovery as classification. The limits on its use come from the need for inputs common to classifiers; a base of relevant primitive predicates, and labeled data distinguishing bad program states from good. It is true that these inputs are hard to provide for more complex programs (especially the predicate base and assertion

violating program states) as they are the product of deep analyses of program structure. However, Binary Fission is also agnostic as to the source of these data, which greatly increases its avenues for application.

Second, our results on the novelty of the Binary Fission solution could be the product of our choice of DTinv as the comparator. This is quite plausible; the likely invariants produced by other machine learning methods might qualify as preconditions. However, our experience with Binary Fission has illuminated constraints that should be applied to the use of classifiers for this task; they should penalize solution size (which is common wisdom), employ a powerful predicate base to support human legibility of the end result, and reward identification of pure good nodes rather than focus on an entropic measure as the splitting criterion.

A third concern is that our use of crowdsourced classification could be replaced by a suitable automated method. This issue reduces to the underlying question, "What does the crowd bring to classification that is difficult to automate?". Here, the core property is novelty; we have shown that the crowd discovers program invariants that DTinv is unable to find, and that it does so by employing non-greedy search and by imposing constraints on the form of the solution. With sufficient implementation effort, that strategy might be automated - it might require a mixture of random forest techniques to approximate crowd search, and a highly non-linear scoring function, like Binary Fission's, which is difficult to optimize. The required mechanism is non-trivial.

More broadly, the purpose of a crowdsourced science game like Binary Fission is to unleash human insight to solve a hard technical problem. That value can be present even in tasks that are well-characterized as search. For example, FoldIt [4] lets players employ their spatial intuition to determine the shape of complex proteins. The game has obtained results never achieved in 30+ years of research based on search over molecular conformations in combination with energy minimization methods. Precondition discovery is equally hard, and the task has a natural framing as classification search. In this context, the crowd may intuit which predicates to employ en route to a more general solution (where predicate invention is a major component of precondition discovery as conducted by human experts). Binary Fission currently hides a bit too much information to support this type of player intuition (a design choice made in service of broadening the game's appeal), but advanced versions will provide more context about the underlying task, and more leverage for predicate selection.

A final, and related argument is that Binary Fission addresses the wrong crowdsourcing problem. Rather than ask the crowd to combine primitive predicates, we should unleash them on the task of inventing the predicates themselves. This step seems natural as predicate invention (including predicate abstraction from data) is a critical, but elusive process currently performed by people. We have, in fact, developed a game for this task, called Xylem [18], and it is available on-line at `xylem.verigames.com`.

9. CONCLUSION

We have employed Binary Fission, a crowdsourced game for invariant discovery, to analyze the implementation of an on-board aircraft collision detection and avoidance system. We have shown that the crowd can employ Binary Fission to prove program properties. They find function precondi-

tions (statements about program variables associated with function inputs) that guarantee important safety properties hold on program exit, where those properties are encoded as postconditions. Binary Fission players discover concise, general, and human readable preconditions, which are also novel relative to the complicated logical expressions often produced by other classifications systems. The players have no special expertise in formal methods or programming, and are not specifically aware they are solving verification tasks.

Binary Fission demonstrates the feasibility of crowdsourced invariant discovery, and it illustrates the promise of crowdsourcing for other verification tasks. This suggests a pathway for expanding the reach, and practical application of verification technology.

10. ACKNOWLEDGMENTS

11. REFERENCES

[1] J. Chamberlain, U. Kruschwitz, M. Poesio, and P. Michelucci. Methods for engaging and evaluating users of human computation systems. In *Handbook of Human Computation*. Springer Science+Business Media, New York, 2013.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, Sept. 2003.

[3] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE 2001*, pages 142–151, 2001.

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

[5] D. Dean et al. Lessons learned in game development for crowdsourced software formal verification. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE'15)*.

[6] S. Deterding, M. Sicart, L. Nacke, K. O'Hara, and D. Dixon. Gamification. Using game-design elements in non-gaming contexts. In *CHI '11 Extended Abs. on Human Factors in Computing Systems (CHI EA '11).*, 2011.

[7] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. Verification games: Making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 42–49. ACM, 2012.

[8] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 443–456, 2013.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2):99–123, 2001.

[10] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, Computer Science, Univ. of Washington, Nov. 1999.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[12] D. Fava, J. Signoles, M. Lemerre, M. Schäf, and A. Tiwari. Gamifying program analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 591–605. Springer, 2015.

[13] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *Computer Aided Verification*, 2014.

[14] A. Gotlieb. Tcas software verification using constraint programming. *The Knowledge Engineering Review*, 27(03):343–360, 2012.

[15] A. Kawrykow, G. Roumanis, A. Kam, D. Kwak, C. Leung, C. Wu, E. Zarour, L. Sarmenta, M. Blanchette, J. Waldispühl, et al. Phylo: A citizen science approach for improving multiple sequence alignment. *PloS one*, 7(3):e31362, 2012.

[16] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725*, 2015.

[17] D. Kroening and M. Tautschnig. CBMC–C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[18] H. Logas, J. Whitehead, M. Mateas, R. Vallejos, L. Scott, D. Shapiro, J. Murray, K. Compton, J. Osborn, O. Salvatore, et al. Software verification games: Designing Xylem, The Code of Plants. In *Foundations of Digital Games (FDG 2014)*, 2014.

[19] J. Lygeros and N. Lynch. On the formal verification of the tcas conflict resolution algorithms. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 2, pages 1829–1834. IEEE, 1997.

[20] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *RN*, 15:01, 2015.

[21] K. L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 1–12, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Dig: a dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):30, 2014.

[23] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact infrastructure repository, 2006.

[24] N. Savage. Gaining wisdom from crowds. *Communications of the ACM*, 55(3):13–15, 2012.

[25] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105, 2014.

[26] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, 2012.

[27] R. M. Stallman et al. Using the GNU compiler collection. 2003.

[28] K. Tuite, N. Snavely, D.-y. Hsiao, N. Tabing, and Z. Popovic. Photocity: training experts at large-scale image acquisition through a competitive game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1383–1392. ACM, 2011.

[29] L. vonAhn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, Aug. 2008.

[30] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 Int'l Symp. on Software Testing and Analysis*, pages 362–372, 2014.