Puzzle-Based Automatic Testing: Bringing Humans into the Loop by Solving Puzzles

Ning Chen and Sunghun Kim
The Hong Kong University of Science and Technology
Hong Kong, China
{ning,hunkim}@cse.ust.hk

ABSTRACT

Recently, many automatic test generation techniques have been proposed, such as Randoop, Pex and jCUTE. However, usually test coverage of these techniques has been around 50-60% only, due to several challenges, such as 1) the object mutation problem, where test generators cannot create and/or modify test inputs to desired object states; and 2) the constraint solving problem, where test generators fail to solve path conditions to cover certain branches. By analyzing branches not covered by state-of-the-art techniques, we noticed that these challenges might not be so difficult for humans.

To verify this hypothesis, we propose a Puzzle-based Automatic Testing environment (PAT) which decomposes object mutation and complex constraint solving problems into small puzzles for humans to solve. We generated PAT puzzles for two open source projects and asked different groups of people to solve these puzzles. It was shown that they could be effectively solved by humans: 231 out of 400 puzzles were solved by humans at an average speed of one minute per puzzle. The 231 puzzle solutions helped cover 534 and 308 additional branches (7.0% and 5.8% coverage improvement) in the two open source projects, on top of the saturated branch coverages achieved by the two state-of-the-art test generation techniques.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*

General Terms: Human Factors, Reliability

Keywords: Testing, Human Computation, Code Coverage

1. INTRODUCTION

Software testing is a difficult but important part of the software development process. However, manual test case writing is labor intensive. Though many automatic test generation techniques [4, 10, 15, 16, 19, 20, 22, 23, 26–29, 34] have been proposed, their coverage needs to be improved. For example, our evaluation study shows that only 61.6% and 53.0% of the total branches of the two subjects were covered by test cases generated by a state-of-the-art approach [22]. Other approaches based on dynamic symbolic execution (DSE) like Pex [29] and jCUTE [26] yield similar coverage results when applied to complex real world programs [33].

The low test coverages are largely due to challenges of generating test inputs, especially when the approaches are applied to object-oriented programs. The major challenges include 1) insufficient objects, 2) *complex constraint solving* and 3) *object mutation*.

First, to construct test cases for object-oriented programs, objects need to be instantiated. However, the process of instantiating a valid object is often not straightforward. This issue is partially addressed by [19] and [28], which capture objects from program executions or code repositories, and reuse them as test inputs.

Second, it is necessary to solve path conditions [29] to create and/or modify test inputs to cover paths. To solve path conditions automatically, various Satisfiability Modulo Theories (SMT) solvers such as Yices [13] and Z3 [9] have been proposed. However these solvers often yield limited results when used to solve certain forms of path conditions such as floating point, non-constant bit-vector and non-linear modulo arithmetic [13]. As shown in Figures 1(a) and (b), a state-of-the-art SMT solver, Yices, cannot solve these path conditions.

```
Path condition:
(foo.x << n) < foo.y                                       (1)
n > 2                                                      (2)
```

(a) A simplified non-constant bit-vector example taken from apache-commons-math 3.2.1 at MultiKeyMap.java:555

```
Path condition:
(foo.x % bar.y) >= n                                       (1)
```

(b) A simplified non-linear modulo example taken from apache-commons-math 3.2.1 at ExtendedProperties.java:1447

Figure 1: Path conditions unsolvable by a state-of-the-art SMT solver, Yices.

```
A path condition solution (model):
specialContainer.size() == 10                            (1)
```

Figure 2: An example of object mutation challenge

Third, even when valid objects are instantiated and solutions to path conditions (i.e. *models*) are obtained, objects still need to be mutated according to the models before using them as test inputs. For example, if we have instantiated a valid `specialContainer` object and have obtained a model as shown in Figure 2 for a path condition, satisfying the model is still hard since automatically figuring out ways to increase the *size()* of this `specialContainer` can be non-trivial. Satisfying a model even as simple as this, by mutating objects, may require extensive and sophisticated static and/or dynamic analysis. For example, in [19], the authors have applied various static analysis techniques to automatically mutate objects based on solution models, but the result was limited. In [33], the authors identified the object mutation challenge as the most significant cause of the low test coverage of the state-of-the-art test generation technique, Pex.

However, the *constraint solving* and *object mutation* challenges presented in Figures 1 and 2 might not be so hard for humans, since we are very good at finding logic and rules quickly. For example, humans can quickly figure out solution: (`foo.x == 1`, `foo.y == 9`, `n == 3`) for Figure 1(a) and solution (`foo.x == 3`, `bar.y == 2`, `n == 1`) for Figure 1(b). Based on this observation, we propose a Puzzle-based Automatic Testing environment, PAT, which decomposes constraint solving and object mutation problems into small puzzles that humans can solve.

We evaluated PAT by generating puzzles for two open source projects and asking humans to solve these puzzles. Our evaluation results showed that humans voluntarily played the presented puzzles and could solve them quickly and effectively: 231 of the 400 presented puzzles were solved by humans, and each puzzle was solved in less then one minute on average. These puzzle solutions improved coverage by 7.0% and 5.8% in two open source projects on top of the very saturated test coverages achieved by state-of-the-art automatic test generators, which is non-trivial.

We compared the effort of writing test cases without PAT and found that solving PAT puzzles is more efficient than writing test cases manually. In addition, solving PAT puzzles does not require any programming skills or domain knowledge of the subjects. In our evaluation, none of the participants had domain knowledge but they improved test coverage significantly by solving PAT puzzles.

This paper makes the following contributions:

A novel puzzle-based testing environment that generates useful test cases through puzzle solving by humans.

An implementation of the proposed approach: PAT.

An experimental study to evaluate PAT.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents a motivating example. Section 4 presents the detailed design of PAT. Section 5 presents evaluation results. Section 6 discusses threats to validity. Finally, Section 7 concludes.

2. RELATED WORK

2.1 Automatic Test Generation

Many automatic test generation techniques have been proposed, such as Randoop [22], which is based on random approach to generate test inputs. Michael et al. [20] proposed another random test generation technique based on genetic algorithm. [4, 10, 16] generates test inputs by symbolic executions. PAT is designed to work as complementary to all these automatic test generation techniques to achieve higher test adequacy.

Dynamic symbolic execution (DSE) such as [15, 23, 29, 34] generates test inputs by solving path conditions captured along dynamic execution paths. This technique combines concrete execution with symbolic execution [26, 27] to achieve high test coverage. However, complex constraint solving and object mutation for object-oriented programs have always been challenging. Xiao et al. [33] reported that complex constraints which cannot be solved by SMT solvers and the object mutation challenge account for nearly 68% of branches not covered by Pex, a DSE framework for C#. Erete et al. [14] also demonstrated limitations of SMT solvers in constraint solving.

Pasareanu et al. [25] tried to enhance classical symbolic execution techniques by partially addressing the complex constraint solving challenge. Their approach splits complete path conditions into simple and complex subsets. Information from the simple subset of the path conditions and concrete executions are leveraged to simplify the solution of the complete path conditions. However, their approach has limitations since information from the simple path conditions is not always adequate to decide the satisfiability of the complete path conditions [25]. PAT can complement this approach by leveraging humans to solve the complex constraints (for SMT solvers, but may be easy for humans) even when the simple path condition information is inadequate.

Thummalapenta et al. [28] proposed MSeqGen which mines existing code bases to extract call sequences that can create objects and mutate them. This approach relies on existing code bases and may not find all necessary call sequences. PAT complements it by transforming object mutation into puzzle solving and leverages humans to help mutate the objects.

PAT is a novel framework for decomposing complex constraint solving and object mutation challenges and presenting them as simple puzzles to humans. Through solving puzzles generated by PAT, humans can help address the challenges in automatic testing.

2.2 Human Computation

Human computation has been an emerging area in computer science [6]. Several projects have been put on the Internet and are open to the general public, such as Foldit [6], reCAPTCHA[1], the ESP game [31] and Pex4Fun [30]. The main idea of human computation is that: though humans compute at a much slower pace than ordinary computers, they are capable of solving many things that computers cannot. For instance, the reCAPTCHA project digitizes books with help from millions of online users typing captchas everyday. The underlying idea of this project is that humans can easily recognize words more precisely than computers. So with the help of human resources available on the Internet, millions of words not recognized by computers can be easily digitized. Most recently, human computation has received attention in the software engineering area also. Dietl et. al [11] designed verification games which can be used to verify program properties with the help of human players. PAT is inspired by these human computation projects. We also believe that humans, even without much domain knowledge of testing subjects, can help software testing by solving simple object mutation and constraint solving puzzles.

3. MOTIVATING EXAMPLE

Figure 3 presents a motivating example taken from Apache Commons Math 2.1 which demonstrates the two major challenges in automatic test generation techniques.

In this code snippet, the `VectorialCovariance.getResult()` method creates an object of type `RealMatrix` by calling a static method `MatrixUtils.createRealMatrix()` (Line 86 (a)). The `MatrixUtils.createRealMatrix()` method then calls the constructor of either `Array2DRowRealMatrix` or `BlockRealMatrix` (Line 62 (b)). If the `Array2DRowRealMatrix` constructor is called, it further invokes its parent constructor `AbstractRealMatrix`, illustrated in Figure 3(c). After creation of the `RealMatrix` object, the `getResult()` method proceeds to a conditional statement, where the execution flows to different branches according to the value of member field `n` (Line 88 (a)). Suppose the branch at Figure 3(a) line 89 has never been covered by existing test cases, and we want to generate new test cases to cover this branch.

[1]http://www.google.com/recaptcha/

```
public RealMatrix getResult() {
84
85  int dimension = sums.length;
86  RealMatrix result =
      MatrixUtils.createRealMatrix(dimension, dimension);
87
88  if (n > 1) {
89    double c = ...     /* target branch to cover */
      ...
98  }
99
100 return result;
}
```

(a) VectorialCovariance.getResult()

```
public static RealMatrix createRealMatrix(final int rows,
                                          final int columns) {
61  return (rows * columns <= 4096) ?
62    new Array2DRowRealMatrix(rows, columns) :
      new BlockRealMatrix(rows, columns);
}
```

(b) MatrixUtils.createRealMatrix(int, int)

```
protected AbstractRealMatrix(final int rowDimension,
                             final int columnDimension)
55                        throws IllegalArgumentException {
56  if (rowDimension <= 0 ) {
57    throw MathRuntimeException...
60  }
61  if (columnDimension <= 0) {
62    throw MathRuntimeException...
65  }
66  lu = null;
}
```

(c) AbstractRealMatrix.AbstractRealMatrix(int, int)

Figure 3: A motivating example from Apache Commons Math

3.1 Object Mutation Challenge

The first challenge of covering this branch is object mutation [33]. Given a model satisfying some path conditions, generating/mutating an object to the desired object state without breaking class invariants [2] is challenging [33]. Without the necessary test inputs in the correct object states, the target branch may not be covered.

The object mutation challenge surfaces when the model for covering the target branch requires non-publicly accessible fields to have certain values or external library calls to return certain values.

```
A model:
this == instanceof VectorialCovariance                (1)
this.sums == instanceof double[]                      (2)
this.sums.length == 1                                 (3)
this.n == 2                                           (4)
```

Figure 4: A model satisfying the path condition in Figure 5

For example, Figure 4 is a model satisfying the path condition in Figure 5 for the target branch (Figure 3(a) line 89). In this model, line (3) requires that a `VectorialCovariance` object's private field `sums` have a length of 1. Similarly, line (4) requires that the private field `n` should have a value of 2. Since both fields are private, and no direct setters are defined in the class for changing their values, mutating `VectorialCovariance` to satisfy the model is a non-trivial task. For this challenge, human computation might be helpful in that by observing the effects of executing member methods or even just observing the method names, humans may recognize and generalize rules about the effects of these methods. The generalized rules can help mutate an object to satisfy a given model.

3.2 Constraint Solving Challenge

In typical dynamic symbolic execution approaches, path conditions need to be retrieved along different program execution paths. For instance, along the path (`getResult():85, 86, createRealMatrix(): 61, 62, AbstractRealMatrix():56, 60, 61, 65, 66, getResult(): 87, 88, 89`) in Figure 3, a path condition is retrieved (Figure 5).

```
Path condition:
this == notnull                                       (1)
this.sums == notnull                                  (2)
this.sums.length * this.sums.length <= 4096           (3)
this.sums.length > 0                                  (4)
this.n > 1                                            (5)
```

Figure 5: Path condition along an execution path

However, when the approaches try to compute models for certain types of path conditions, such as those containing non-linear or floating point arithmetics, SMT solvers may return an error. For example, a state-of-the-art SMT solver, Yices [13], fails to compute a model for the path condition in Figure 5, with an error message: *"Error: feature not supported: non linear problem."* due to constraint (3). Even though there might be other SMT solvers which provide stronger support, many theories such as non-linear arithmetics are undecidable. In addition, some weaknesses in SMT solvers are demonstrated by [14]. Without a necessary model satisfying the path conditions, these automatic test generation approaches would not be able to create the necessary test inputs to cover the corresponding target branch. For this challenge, human computation can be helpful since some of these arithmetics might not be too difficult for humans.

Based on these observations, we propose PAT, a framework which tries to leverage human intelligence to help address the two challenges to improve test adequacy.

4. DESIGN AND IMPLEMENTATION

This section presents the overall design and implementation of PAT. In general, two types of puzzles are generated for the not covered branches. Puzzle solutions by humans are then used to generate test cases automatically.

Figure 6 presents the architectural design of PAT which consists of five main phases: up-front testing runs, path computation, mutation puzzle generation, constraint solving puzzle generation and finally test case generation from puzzle solutions. 1) Initially, PAT runs test cases generated by several state-of-the-art automatic test generation techniques to obtain a code coverage report together with various dynamic information. 2) Based on the coverage report, PAT collects all branches not covered by the up-front testing runs. Then, PAT tries to find program paths along with models which satisfy the path conditions by a symbolic execution algorithm for each of the not covered branches. 3) For branches which have feasible paths along with models that satisfy the path conditions, but contain difficult to mutate fields such as the one in Figure 4, PAT transforms the mutation problems into object mutation puzzles. 4) For branches for which PAT cannot find any feasible program paths to cover in phase 2), due to the SMT solver's limitations, PAT decomposes the path conditions of the branches and transforms them into constraint solving puzzles. 5) Finally, puzzle solutions obtained from humans are automatically analyzed and converted into executable test cases. The following subsections describe each phase in detail.

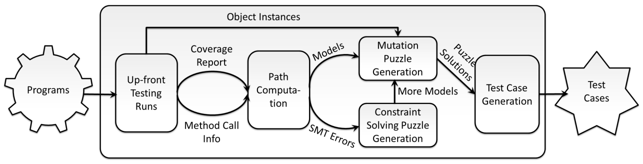


Figure 6: The architecture of PAT.

4.1 Up-front Testing Runs

In this phase, PAT runs test cases automatically generated from various state-of-the-art automatic test generation techniques [19, 22]. There are two main purposes of the up-front testing runs. First, all automatically coverable elements are covered. Up-front testing runs phase makes sure that human efforts are not spent on elements that can be effectively covered by automatic test generation techniques. The second purpose is to collect dynamic information from the executions, including object instances generated during executions and the method call sequences. A method call sequence is the sequence in which methods call each other. Object instances are captured through the object-capturing code inserted using the ASM [5] library. The captured objects are saved and later restored using the XStream [32] framework. The dynamic information is essential to facilitate path computation and puzzle generation phases of PAT.

After the up-front testing runs, PAT obtains a detailed coverage report and identifies elements (e.g., branches) not covered by automatic test generation techniques. These not covered elements become targets of the later puzzle generation phases.

4.2 Path Computation

From the coverage report of the up-front testing runs, PAT identifies the set of not covered elements (i.e., branches). Our goal is to construct test cases which can effectively cover these elements. To achieve this goal, for each of the not covered branches, PAT looks for feasible execution paths covering the branch together with the corresponding path conditions.

Our approach can be viewed as an instance of the weakest precondition [12] computation similar to [3, 21]. Our approach is an inter-procedural, path-sensitive and context-sensitive backward symbolic execution algorithm. PAT first transforms the subject program from Java byte-code into static single assignment (SSA) [7] representation using WALA [18]. It then performs the symbolic execution from a target branch and runs backward to the entry of the current method which the branch is in. If the current method is public, the symbolic execution terminates. Otherwise, PAT finds a caller method of the current method from the method call sequence information collected during the up-front testing runs. The backward symbolic execution continues from the call site of the caller method to its entry. This process continues until it reaches a public method's entry or PAT cannot find any caller method anymore.

During the backward symbolic execution, PAT collects and propagates a set of path conditions represented in a symbolic manner, along the backward execution path. Thus, when the symbolic execution is completed, a set of path conditions representing the necessary conditions to reach the target branch from a public method's entry is retrieved. PAT then tries to solve the set of path conditions using an SMT solver, Yices [13]. If the set of path conditions is satisfiable, a model is returned from the SMT solver. Otherwise, the backward symbolic execution is restarted by backtracking to an alternate backward path at the nearest program branching point. The process of collecting feasible execution paths and the corresponding path conditions for a branch is completed when PAT has obtained a certain number (currently 10) of models corresponding to different execution paths.

Termination handling However, when computing feasible paths of a branch, the above approach might run forever because of the undecidability problem in software verification [1]. This problem is generally caused by loops and recursive calls in the code. To address this issue, two parameters are introduced that limit 1) the maximum number of times a loop can be unrolled, and 2) the maximum invocation depth allowed. Hence, under the restrictions imposed by these two parameters, the path computation process for each branch can enumerate every possible execution path within a finite number of steps and guarantee termination. However, under such approximations, the computation approach becomes neither sound nor complete. In our experiment, we set the two parameters to two and five.

We also impose two additional bounds to the path computation phase to make sure it can terminate within a reasonable time: 1) the maximum computation time for one path is limited to 600 seconds; and 2) PAT is allowed to invoke the SMT solver for at most 1,000 times in the path computation phase for each branch.

4.3 Mutation Puzzles

For branches which have valid models satisfying the path conditions, but do not have test inputs satisfying these models, PAT generates object mutation puzzles from these models. As indicated in Section 3, generating test inputs satisfying models automatically might not be a trivial task in object-oriented programs since not all fields are directly assignable. Therefore, we leverage human intelligence to help address this challenge.

To satisfy a given model, we need to mutate one or several objects into certain object states. For instance, for the model in Figure 4, we need to mutate an `VectorialCovariance` object into an object state where field `n` has a value of 2, and field `sums` has a length of 1. If such fields are public, PAT can simply assign the corresponding values to them. However, if they are non-public, PAT needs to figure out a sequence of method calls that can mutate an object to this goal state. The objective of object mutation puzzles is to obtain such a sequence with human help.

4.3.1 Generating Sub-models

```
Model:
in == notnull                                          (1)
in.readInt() == 1                                      (2)
this.currentState == null                              (3)
```

(a) A complete model

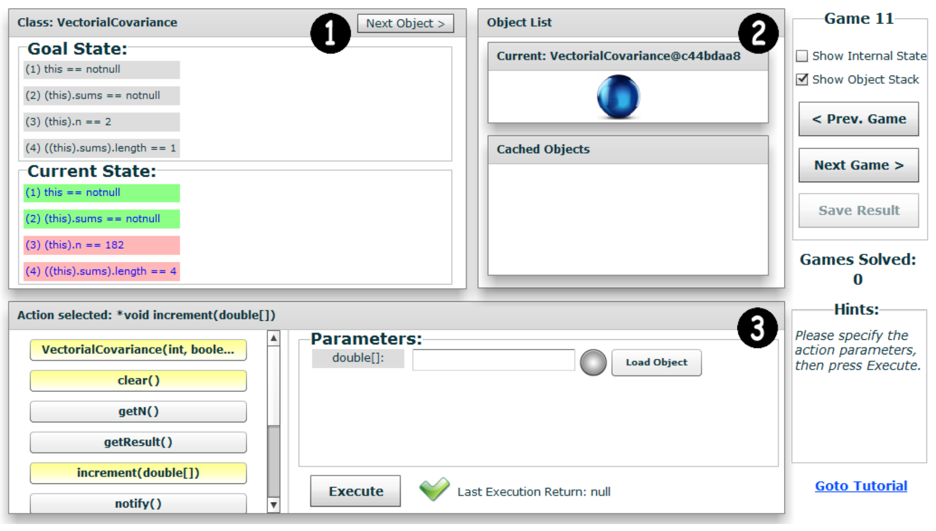
```
Sub−Model 1:
in == notnull                                          (1)
in.readInt() == 1                                      (2)

Sub−Model 2:
this.currentState == null                              (1)
```

(b) The divided sub-models

Figure 7: Sub-model division

Instead of presenting an entire model directly, PAT divides a model into several *sub-models* to make the generated puzzles as simple as possible. A sub-model represents the goal state of only one object reference, such as the callee or a parameter. Figure 7 illustrates one such division. The complete model in Figure 7(a) is divided into two sub-models in Figure 7(b) to represent goal states


Class: VectorialCovariance

Next Object >

Goal State:

```
(1) this == notnull
(2) (this).sums == notnull
(3) (this).n == 2
(4) ((this).sums).length == 1
```

Current State:

```
(1) this == notnull
(2) (this).sums == notnull
(3) (this).n == 182
(4) ((this).sums).length == 4
```

Object List

Current: VectorialCovariance@c44bdaa8

Cached Objects

Game 11

- [ ] Show Internal State
- [✓] Show Object Stack

< Prev. Game

Next Game >

Save Result

Games Solved: 0

Action selected: *void increment(double[])

VectorialCovariance(int, boole...

clear()

getN()

getResult()

increment(double[])

notify()

Parameters:

double[]:

Load Object

Execute

Last Execution Return: null

Hints:

Please specify the action parameters, then press Execute.

Goto Tutorial

Figure 8: Screenshot of mutation puzzles. Panel 1 is the Object State panel showing the goal and the current object state. Panel 2 is the Object panel showing the current mutating object. Panel 3 is the Action List panel that lists available actions with parameters.

of `in` and `this` separately. The solution of the complete model can be retrieved by combining solutions of all individual sub-models. The retrieved solution is correct as long as the solutions to the individual sub-models are independent of each other. This means that the solution to one sub-model should not affect solutions to any other sub-models.

4.3.2 *Prioritizing Sub-models*

Before generating puzzles, PAT first prioritizes all sub-models to make better use of human efforts. The key insight for prioritizing sub-models is that, one sub-model can be shared by many different models. For example, sub-model 2 in Figure 9 is identical to sub-model 2 in Figure 7(b). Sub-models shared by many different models are assigned higher priorities.

```
Sub-Model 1:
in == null                                          (1)

Sub-Model 2:
this.currentState == null                           (1)
```

Figure 9: Another set of Sub-models

4.3.3 *Generating Puzzles for Sub-models*

After prioritizing the sub-models, object mutation puzzles are generated for them. Puzzles generated from sub-models with higher priorities are presented earlier. Figure 8 presents the interface of an object mutation puzzle generated from a sub-model. When designing the puzzle interface, we want to make it as simple to play as possible, such that humans can learn to play it within a few minutes. On the puzzle interface, there are three different panels, namely, 1) the Object State panel, 2) the Object panel, and 3) the Action List panel. The Object State panel comprises two sections. The upper section shows the goal object state that should be satisfied in the puzzle. The bottom section shows the current object state. The Object panel shows the object currently being mutated. Users can also save any intermediate object instances in the Object Cache section of this panel. Finally, the Action List panel lists all public member methods (actions) in button style. Thus, users can invoke different member methods by pressing the corresponding buttons.

Initially, PAT tries to instantiate a set of objects of the type specified by the sub-model. There are two different ways by which PAT can instantiate objects: 1) load from the set of previously saved objects, or 2) find and invoke the constructor that takes no parameter. If both fail, PAT tries to instantiate objects of the sub-classes until a set of objects is instantiated or there are no more sub-classes to try. If a set of objects is successfully instantiated, a random object in the set is used as the current object. Users can easily change the current object by pressing the “Next Object” button.

To perform an action, users can press any action button from the Action List panel. Parameters are assigned in the following manner: if it is a primitive type parameter, users can simply type in the primitive value such as `1`, `0.5` or `true`; if it is not a primitive type parameter, users can press the “Load Object” button to instantiate an object of the corresponding type and use it as the input parameter. The mechanism of instantiating a parameter object is the same as the mechanism of instantiating a current object for a mutation puzzle, as previously described. If no object is loaded, a `null` value is used. After filling in all parameters, users can press the “Execute” button to execute the action.

After an action is executed, PAT immediately re-evaluates the satisfaction of each model line in the Object State panel. Once a model line has been satisfied by the current object state, PAT turns its color to green in the Current State section. The puzzle is considered solved if all model lines of the sub-model have turned green, and the solution is automatically saved into the server database. The solution of a puzzle consists of all previously executed actions (i.e. method calls) along with the input parameters.

4.3.4 *Providing Hints*

Since there can be many available actions in the Action List panel, we want to further simplify the solution of puzzles by recommending a set of potential actions to humans. PAT conducts a static analysis to identify all actions containing statements which can change the value of at least one field in the sub-model. All such actions are recommended to users by highlighting them in yellow. Figure 10 demonstrates the simplified pseudo-code for finding recommended actions in a target class.

For each method in the target class, we analyze each of the instructions in the method. If the current instruction is an assignment

```
Input the target class of the mutation puzzle.
Input the target fields which we want to change.
Conduct points-to analysis and build call graphs.
for each method in the target class:
  for each instruction in the method:
    if it is an assignment instruction:
      if the assignee is one of the target fields:
        Mark current method as recommend.
        break
    if it is an invocation instruction:
      if callee has not been analyzed before:
        Recursively analyze if callee is recommend.
      if the callee method is recommend:
        Mark current method as recommend.
        break
Output all methods marked as recommend.
```

Figure 10: Pseudo-code for finding recommending methods

instruction which assigns values to one of the fields appearing in the sub-model, we mark the current method as *recommend*. If the current instruction is an invocation instruction, and the callee method has not been analyzed before, PAT recursively analyzes the callee method of this invocation. The current method is marked as *recommend* if the callee method has been marked *recommend*. We limit the maximum recursion level of the analysis to 10 to avoid infinite recursion in the presence of cyclic calls. Global lists of *recommend* and *analyzed* methods are maintained during the process to avoid redundant analysis.

In Figure 8, actions `VectorialCovariance(int, boolean)`, `clear()` and `increment(double[])` are highlighted because they can change either `this.n` or `this.sums` or both.

4.3.5 *Solving the Puzzles*

A typical scenario to solve the puzzle presented in Figure 8 is as follows. A player observes the list of available actions. Judging from the action names, argument types, as well as the highlighted hints, the user should be able to infer a group of candidate actions quite easily. After a few attempts on the candidate actions, the player may soon find out that model line (4) can be satisfied by invoking action `VectorialCovariance(int, boolean)` with 1 as the first parameter value. Similarly, invoking the highlighted action `increment(double[])` with a double array increases `this.n` from 0 to 1 (this field has already been reset to 0 in the first action). Thus, model line (3) can be satisfied by invoking this action twice. In this way, the presented sub-model is satisfied, and all actions are recorded automatically.

4.4 Constraint Solving Puzzles

In case of branches for which PAT cannot find any feasible paths to cover in the path computation phase, PAT goes on to examine whether the computation failures are due to SMT solver limitations. As indicated in Section 3, even though some particular forms of path conditions, such as those containing non-linear arithmetic, are not solvable by the SMT solver, it does not necessarily mean that they are naturally difficult for humans. When such path conditions are found, PAT extracts the not solvable constraints from these path conditions, decomposes them, and presents them to humans in the form of constraint solving puzzles.

4.4.1 *Extracting Error Related Constraints*

Given a target branch, if PAT is not able to obtain any feasible execution path with path condition, it examines all path conditions which have been input into the SMT solver for satisfiability checks. Usually, if a path condition cannot be handled by the SMT solver, the solver outputs an error message indicating the cause of the error, as shown in Figure 11.

```
Path conditions:
this == notnull                                        (1)
this.sums == notnull                                   (2)
this.sums.length * this.sums.length <= 4096            (3)
this.sums.length > 0                                   (4)
this.n > 1                                             (5)
Error: feature not supported: non-linear problem.
```

Figure 11: An example of error-causing path conditions

Once PAT has found path conditions which cause SMT errors, it saves them for future puzzle generations. Since PAT performs SMT checks along many different paths for a given branch, there can be many different path conditions that cause errors. Currently, PAT saves only the first 10 erroneous path conditions it finds for one branch. Saving more path conditions for puzzle generations can increase the probability of finding a satisfiable model but it also requires more humans to solve them.

For the saved path conditions, it might still be too long to present them directly. By observing the path conditions in Figure 11, we notice that constraints (1), (2) and (5) are independent of the error related constraints (i.e., (3) and (4)). Therefore, PAT performs an additional SMT check to obtain a partial model which satisfies all the non-error related constraints (Figure 12(a)). Thus, only the error related constraints are left for puzzle generation (Figure 12(b)). Note that even though constraint (4) in Figure 11 is not an error-causing constraint, it is related to the error-causing constraint (i.e., (3)) in that they are both constraining the same variable. Therefore, we still regard it as an error related constraint.

```
A partial model:
this == instanceof VectorialCovariance                 (1)
this.sums == instanceof double[]                       (2)
this.n == 2                                            (3)
```

(a) A partial model satisfying the non-error related constraints

```
Error related constraints:
this.sums.length * this.sums.length <= 4096            (1)
this.sums.length > 0                                   (2)
```

(b) Error related constraints left for puzzle generation

Figure 12: Extracted error related constraints

The partial model in Figure 12(a) and a model obtained from public puzzle solving can be joined together to form a complete model, such as the one in Figure 4.

4.4.2 *Prioritizing Constraint Sets*

Before generating constraint solving puzzles, PAT groups and prioritizes the error-related constraint sets first. The key idea of the grouping schema is that even though many constraint sets are literally different, some are actually semantically equivalent. For example, even though both constraint sets in Figure 13 are extracted from different path conditions, and they are literally different, it is easy to notice that these constraint sets are actually semantically equivalent. By solving only one of them, PAT can retrieve models for both constraint sets. Hence, PAT identifies all constraint sets which are different only by their corresponding variable names as semantically equivalent. These constraint sets are put into the same group, and PAT generates only one representative puzzle from a random constraint set in this equivalent group.

```
Error related constraint set  1:
this.sums.length * this.sums.length <= 4096          (1)
this.sums.length > 0                                 (2)
```

(a) Error related constraint set

```
Error related constraint set  2:
this.covarianceImpl.sums.length *
    this.covarianceImpl.sums.length <= 4096          (1)
this.covarianceImpl.sums.length > 0                  (2)
```

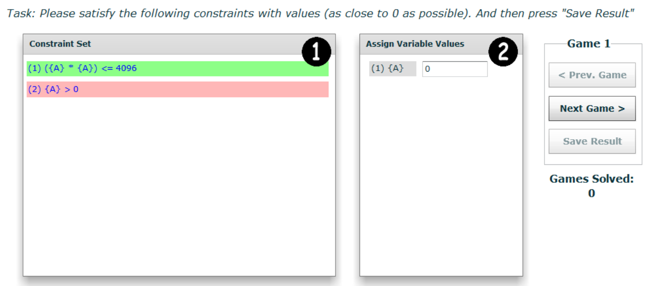
(b) Error related constraint set

Figure 13: Two semantically equivalent error related constraint sets

Then, PAT prioritizes the groups of constraint sets to make the best use of human efforts. Since solving each error-related constraint set may lead to the solutions of multiple models for different branches, the goal of the prioritization schema is to maximize the number of additional branches potentially coverable by the models.

To achieve this, PAT dynamically computes the potential gain of branch coverage for each constraint set group and selects the one with the highest potential gain to present first. If there are multiple groups with the same highest potential gain, PAT selects the one with the fewest number of constraints.

4.4.3 *Generating Puzzles for Constraint Solving*


Task: Please satisfy the following constraints with values (as close to 0 as possible). And then press "Save Result"

Constraint Set

(1) ({A} * {A}) <= 4096

(2) {A} > 0

Assign Variable Values

(1) {A} 0

Figure 14: Screenshot of constraint solving puzzles. Panel 1 presents the constraints being solved in the puzzle. Panel 2 is the input area where users assign concrete values to variables.

After the error related constraint sets are grouped and prioritized, they are presented as constraint solving puzzles by PAT. Figure 14 is a screenshot of the PAT constraint solving puzzle. Two different panels are constructed, with the left panel presenting the constraint set required to be satisfied and the right panel providing areas for humans to input concrete values. Whenever a value is changed by the user, PAT immediately re-evaluates the satisfaction of each constraint line. Constraint lines satisfied by the current set of concrete values are highlighted in green. A puzzle is considered solved when all constraint lines have turned green.

Since each puzzle is actually a representation of a group of constraint sets that might be constraining different variables, all variable names in the constraint lines are masked with symbols (i.e. {A}, {B}, etc.).

4.5 Constructing Test Cases from Solutions

The last phase of PAT is to generate test cases at the server side from puzzle solutions.

Solutions of constraint solving puzzles can be directly used as primitive type arguments or they can be assigned to publicly accessible fields based on the solved models. In case the related fields are not directly assignable, these solutions can be used as new models to generate additional object mutation puzzles.

For object mutation puzzles, the complete action sequence taken by a player to solve a puzzle is directly translated into a corresponding method call sequence. The method call sequence can generate test inputs satisfying a target model. For example, Figure 15 shows the method call sequence generated from the action sequence described in Section 4.3.5. The generated method call sequence produces a `VectorialCovariance` object satisfying the target model in Figure 4. It is possible that a player can perform some trial actions during puzzle solving which are not relevant to the solution. In such case, the resulting method call sequence can contain methods whose side-effects do not contribute to the test input generation. However, since the test inputs generated by the whole method call sequence can still satisfy the target model, these irrelevant method calls do not affect the test case generation process.

To construct the final test case for the model, PAT invokes the target method which includes the target branch, with test inputs generated by the method call sequences.

```
VectorialCovariance var0 =
    new VectorialCovariance(1, true);
double[] var1 = (double[])ObjLoader.loadObjectFromFile(
    "/bigstore/pat/Captured/double[]/hash_10032989");
var0.increment(var1);
double[] var2 = (double[])ObjLoader.loadObjectFromFile(
    "/bigstore/pat/Captured/double[]/hash_10032989");
var0.increment(var2);
```

Figure 15: An example of generated method call sequence

5. EVALUATION

5.1 Evaluation Setup

In this section, we investigate the usefulness of PAT by addressing the following research questions:

RQ1 How many object mutation puzzles and constraint solving puzzles can be solved by humans?
RQ2 How many people would play PAT voluntarily?
RQ3 How much is the test coverage improved by the puzzle solutions of PAT?
RQ4 How much manual test case writing effort can be saved with the help of PAT?

5.1.1 *Subjects*

For evaluation, we deployed the PAT framework to the Apache Commons Math (ACM)[1] library version 2.1 and the Apache Commons Collections (ACC)[2] library version 3.2.1.

Table 1: Subjects: ACM and ACC

Name	Version	# of Lines	# of Branches
Commons Math	2.1	20,605	7707
Commons Collections	3.2.1	14,261	5242

ACM is a lightweight, self-contained, mathematics and statistics library. ACC is a data container library which provides more

[1]http://commons.apache.org/math/
[2]http://commons.apache.org/collections/

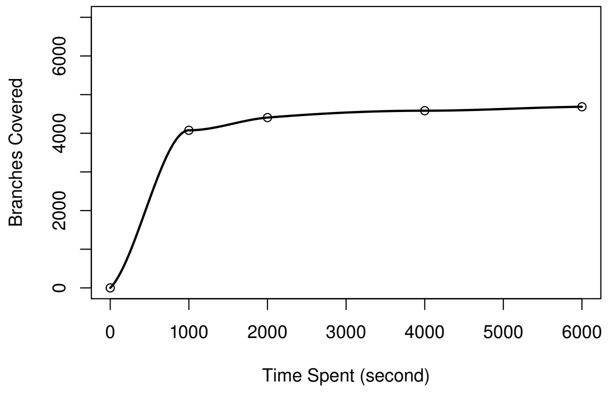


Figure 16: The Randoop coverage result for ACM

data structures over JDK. Both subjects are popular medium size libraries, and have been commonly used in software engineering literature [8, 17, 24].

5.1.2 Up-Front Testing Runs And Baseline Techniques

Before PAT actually generates puzzles for the subjects, a set of automatically generated up-front test cases needs to be executed, as described in Section 4.1. This phase removes branches easily coverable by automatic test generation techniques. Thus, PAT generates puzzles only for branches not coverable by automatic test generation techniques.

In the evaluation we generated the up-front test cases with two types of automatic test generation techniques. These two techniques are also used as baseline techniques when we compare the coverage improvements achieved by PAT in RQ3.

The first up-front testing technique we used was *Randoop* [22], a state-of-the-art feedback directed random test generation technique. Figure 16 presents the coverage results of Randoop. The X-axis in the figure represents the test case generation time (in seconds) spent by the technique. For example, 1000 means the technique was set to generate test cases for 1,000 seconds. The Y-axis represents the total number of branches covered by test cases generated by the technique. Figure 16 shows that the coverage is basically saturated after 4,000 seconds. Thus, we used 6,000 seconds as the time setting for Randoop to generate the up-front test cases.

The second up-front testing technique we used was a symbolic execution based automatic test generation technique (*Symbolic*). We had adapted and implemented a simplified version of the symbolic execution based test generation module. This module uses the same path computation algorithm described in Section 4.2 to obtain the path conditions for each branch not covered by Randoop. To construct test inputs satisfying the path conditions, this symbolic execution module attempts to mutate objects automatically without breaking any class invariants [2]. More specifically, it assigns model values only to primitive type arguments and to member fields that are publicly accessible. It is possible to assign values to non-public fields through Java reflection but our symbolic module does not do that since this may easily break class invariants and produce many objects with invalid states.

In total, 64.4% of branches were covered with test cases generated by the two up-front testing techniques for ACM. For ACC, 56.7% of branches were covered with test cases generated by the two techniques. In our study, we used these two up-front testing techniques. However, besides these two techniques, other automatic test generation techniques can also be deployed as the up-front testing runs to generate PAT puzzles.

5.2 RQ1: Puzzle Solving Results

To investigate the effectiveness of humans in solving the puzzles, we conducted one small scale study targeting the graduate computer science (CS) students. We invited a group of eight graduate CS students to play both kinds of puzzles to investigate how effective they were in solving the puzzles. All participants were presented with the top 100 object mutation puzzles prioritized by PAT as described in Section 4.3.2 and top 100 constraint solving puzzles prioritized as described in Section 4.4.2. For evaluation purposes, all participants were presented with the same set of 100 object mutation puzzles and 100 constraint solving puzzles. However, in the actual deployment, PAT does not present already solved puzzles again. Table 2 shows the overall puzzle solving results.

Table 2: Puzzle Solving Results for ACM

Puzzle Type	Total Presented	Total Solved	Avg. Time
Mutation	100	51	1 min
Constraint	100	72	1 min

We investigated the effectiveness of participants in solving the presented puzzles in two aspects: 1) how many puzzles can they solve; and 2) how long does it take for them to solve the puzzles. Note that since the lists of puzzles solved by participants can have overlaps, redundantly solved puzzles are counted only once.

In total, 51 of the 100 object mutation puzzles were successfully solved by our participants. On average, a participant spent one minute on a puzzle, before solving it or moving to the next puzzle.

The evaluation results show that the participants were able to handle the mutation puzzles quite effectively, with a 51% puzzle solving rate and one minute time spent on each puzzle.

Table 2 presents the overall results for constraint solving puzzles. In terms of puzzle solving, after removing the redundant ones, 72 of the 100 constraint solving puzzles were successfully solved by our participants. On average, a participant spent one minute on a puzzle, before solving it or moving to the next puzzle.

The evaluation results indicate that our group of participants can also solve the constraint solving puzzles quite effectively, with a 72% puzzle solving rate and one minute time spent on each puzzle.

Overall, the group of participants (i.e. CS graduate students) which we investigated could solve both types of puzzles effectively.

5.3 RQ2: Voluntary Participation

The next research question we want to investigate is: how many people would play PAT voluntarily?

To investigate this question, we setup and deployed PAT on the second subject, ACC, and generated puzzles for it. Then, we posted the link to the puzzles on Twitter and encouraged people to participate on a voluntary basis. Our tweet was retweeted several times.

Table 3: Puzzle Solving Results for ACC

Puzzle Type	Total Presented	Total Solved	Avg. Time
Mutation	100	24	1 min
Constraint	100	84	1 min

In total, 120 people volunteered to play the puzzles. Table 3 presents the overall results of puzzle solving by these participants. Overall, 24 of the top 100 object mutation puzzles and 84 of the top 100 constraint solving puzzles were successfully solved. In terms of time spent, on average, a participant spent one minute on both kinds of puzzles before solving it or moving to the next puzzle.

Comparatively, the solving rate for object mutation puzzles was lower, but the solving rate for constraint solving puzzles remained high comparing to puzzle solving by CS graduate students (in RQ1).

To encourage more people to participate and play the PAT puzzles, we can further introduce a competition and reward mecha-

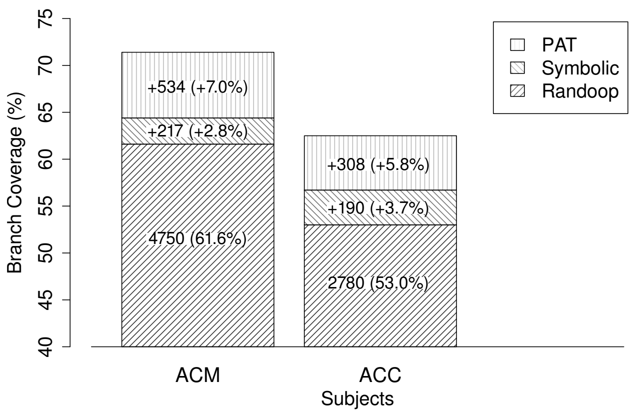


Figure 17: Coverage improvements of ACM and ACC. The numbers on the bar plots show the amount and percentage of additional branches covered by each technique.

nism. For instance, humans can compete for coming up with the first solution of a puzzle. Points or prizes can be awarded to them as stimulation. Implementing such a mechanism can attract more people to play the PAT puzzles, but it is more of an engineering issue. A competition and reward mechanism will be introduced in future versions of PAT.

5.4 RQ3: Coverage Improvement

We have shown that PAT puzzles can be effectively solved by humans and people are willing to participate in solving puzzles. The next research question we want to investigate is: how much can these solved puzzles improve test coverage. For this question, we generated test cases using the puzzle solutions from RQ1 and RQ2 for the two subjects, ACM and ACC. We investigated the coverage improvements achieved by these PAT test cases and compared them with the two baseline techniques, Randoop and Symbolic.

We first used the two baseline techniques in up-front testing runs as shown in Section 5.1.2. Only constraints and mutations related to the not-covered branches by these two baseline techniques were presented to humans as puzzles (Sections 5.2, 5.3), since human effort should not be wasted on branches coverable by automatic techniques. We investigated how much PAT improves coverage, on top of these two baseline techniques.

Figure 17 presents the numbers and percentages of branches covered by the baseline techniques and PAT for the two subjects. *Randoop* represents the branch coverage of running test cases generated by Randoop. *Symbolic* shows the additional coverage that can be achieved with Symbolic. Finally, *PAT* is the number of additional branches covered with test cases generated by PAT.

For ACM, Randoop achieved a coverage of 4750 (61.6%) branches. As shown in Figure 16, coverage of this technique had reached saturation after running for 4,000 seconds. Symbolic covered an additional 217 (2.8%) branches on top of Randoop. Overall, the two baseline techniques together achieved 64.4% branch coverage.

By running test cases generated from 123 puzzle solutions obtained in RQ1, PAT improved the coverage of the two baseline techniques by 534 (7.0%) branches. Considering the saturated coverage of the two baseline techniques, and the relatively small scale of the study (solving only 123 PAT puzzles), a 534 (7.0%) branch coverage improvement is non-trivial. The coverage improvement was achieved by humans without domain knowledge of the subjects.

Similarly for ACC, the two baseline techniques had achieved a branch coverage of 56.7% together. On top of that, PAT improved the coverage by 308 (5.8%) branches, leveraging only 108 puzzle solutions.

Overall, after solving only 123 puzzles of ACM and 108 puzzles of ACC, PAT successfully generated test cases to cover 534 (7.0%) and 308 (5.8%) additional branches for the two subjects, on top of the two saturated baseline techniques. This shows that PAT solutions were efficient in generating test cases to cover more branches. In addition, it shows that PAT's prioritization approaches described in Section 4.3.2 and Section 4.4.2 were effective, and participants' efforts were well spent on the most important puzzles. On average, each puzzle solution helped cover three to four branches, while one puzzle took about one minute to solve, as shown in RQ1 and RQ2.

5.5 RQ4: Effort Reduced

We have shown that PAT puzzles can be solved by humans and the puzzle solutions can improve test coverage significantly. Since the puzzle solving process includes human effort, one could argue that humans can directly write the test cases manually without playing PAT puzzles.

To evaluate this, a graduate student randomly selected ten not covered branches from each subject and tried to write test cases manually to cover them. He had six years of Java programming experience, and certain degree of domain knowledge of the subjects.

Table 4: Manual effort (in Minutes) required to write test cases for ten randomly selected branches in ACM and ACC

	ACM Branch	Time	ACC Branch	Time
1	PolynomialSplineFunction:147	4m	AbstractHashedMap:624	22m
2	VectorialCovariance:93	6m	AbstractMapBag:263	12m
3	MicrosphereInterpolatingFunction:220	13m	BoundedBuffer:164	7m
4	FractionFormat:244	2m	BoundedFifoBuffer:284	3m
5	StepNormalizer:152	17m	InstantiateTransformer:62	4m
6	DirectSearchOptimizer:201	5m	LazyList:117	4m
7	LeastSquaresConverter:180	9m	ObjectGraphIterator:189	11m
8	LoessInterpolator:308	12m	LinkedMap:171	5m
9	RotationOrder:802	4m	Flat3Map:478	3m
10	AbstractRealMatrix:885	13m	CompositeCollection:285	4m
	Average	9m	**Average**	8m

Table 4 shows the time in minutes spent on writing test cases for ten randomly selected branches for each project. Mostly, it took 3 to 9 minutes to write a test case to cover one branch. Some branches such as *AbstractHashedMap:624* took more than 20 minutes as the pre-condition for covering the branch was complex and involved several different methods.

Overall, it took 8 minutes on average to manually write one test case to cover a branch. These results indicate that manual test writing to cover even one branch requires significant time and effort.

However, PAT, with its prioritization approaches (Section 4.3.2 and Section 4.4.2), covered 534 branches for ACM and 308 for ACC leveraging only 123 puzzle solutions from ACM and 108 from ACC. On average, each PAT puzzle helped cover three to four branches. Note that each PAT puzzle was solved in one minute on average, as shown in Section 5.2 and Section 5.3.

In addition, the time spent by participants on solving PAT puzzles and time spent by developers writing test cases should not be directly compared. In general, much more programming and domain knowledge is required to write test cases for a given subject. On the other hand, the general public can play the puzzles with little or no domain knowledge of the subject. Therefore, the use of PAT makes the allocation of resources more effective as the effort from the general public can be utilized to test many branches. Thus, developers' limited time can be focused on writing test cases for branches not coverable by both automatic test generation techniques and PAT. Furthermore, developers can also play the puzzles themselves. With their knowledge in the subject programs, they may solve puzzles that are difficult for most ordinary participants.

The puzzle solutions they provide may help them cover more program branches without the need of writing test cases manually.

Overall, results in this section show that PAT can effectively reduce the time and effort required by developers to write test cases.

6. THREATS TO VALIDITY

Threats to external validity. In our evaluation, we used two medium size open source libraries, ACM and ACC, which are widely used in the literature as subjects [8, 17, 24]. Since evaluation requires participant by many users and is time consuming, we evaluated PAT on only these two subjects. However, these subjects might not be representative of other kinds of systems such as closed-source systems.

PAT uses Yices as its SMT solver in puzzle generation. It is possible that constraints unsolvable by Yices may be solvable by other SMT solvers. We plan to use more SMT solvers to further reduce the number of puzzles needed to be solved by humans.

Threats to internal validity. A threat to internal validity comes from captured objects used in mutation puzzles. Currently, PAT only captures objects from the up-front test running phase (Section 4.1). The quality and quantity of the captured objects may affect the results of PAT.

Threats to construct validity. The major threat to construct validity comes from the measurement of usefulness. In the evaluation, we used the branch coverage criterion to evaluate the usefulness of PAT. There are other metrics that can be used to evaluate usefulness. Therefore, conclusions obtained from our measurement criterion might not be representative of other measurement metrics.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented PAT, which decomposes object mutation and complex constraint solving problems into small puzzles for humans to solve. Our evaluation study shows that humans can efficiently solve these mutation and constraint problems by voluntarily playing the puzzles. The solved puzzles yield non-trivial test coverage improvement. Overall, PAT successfully leverages crowd-sourcing to assist test input generation and automatic software testing.

We anticipate that future software engineering approaches will leverage more crowd-sourcing to address challenges in software engineering research. PAT is a first step in this direction. In future, we expect PAT to become an easily extensible platform to facilitate more types of puzzles related to software testing. The PAT puzzles and our experiment data are publicly available at `http://pat.cse.ust.hk:8080/`.

8. REFERENCES

[1] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *Proc. ICSE*, New York, NY, USA, 2008. ACM.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, 2002.

[3] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proc. PLDI*, 2009.

[4] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[5] O. Consortium. Asm. `http://asm.objectweb.org`.

[6] S. Cooper, A. Treuille, J. Barbero, A. Leaver-Fay, K. Tuite, F. Khatib, A. C. Snyder, M. Beenen, D. Salesin, D. Baker, and Z. Popović. The challenge of designing scientific discovery games. In *Proc. FDG*, 2010.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[8] B. Daniel and M. Boshernitsan. Predicting effectiveness of automatic testing tools. In *Proc. ASE*, ASE '08, pages 363–366, Washington, DC, USA, 2008.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[10] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[11] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. Verification games: Making verification fun. In *FTfJP'2012: 14th Workshop on Formal Techniques for Java-like Programs*, Beijing, China, June 2012.

[12] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.

[13] B. Dutertre and L. de Moura. System description: Yices 1.0. In *Proc. SMT-COMP*, 2006.

[14] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proc. CSTVA*, pages 310 –315, March 2011.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[16] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. ISSTA*, ISSTA '98, pages 53–62, 1998.

[17] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and java. In *SLE*, volume 5969 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2009.

[18] IBM. T.J. Watson Libraries for Analysis (WALA). Online manual. `http://wala.sf.net`.

[19] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object Capture-based Automated Testing. In *Proc. ISSTA*, July 2010.

[20] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27:1085–1110, December 2001.

[21] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proc. ICSE*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[23] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proc. ICSM*, Sept. 2010.

[24] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proc. FSE*, SIGSOFT '08/FSE-16, pages 135–145, New York, NY, USA, 2008. ACM.

[25] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proc. ISSTA*, ISSTA '11, pages 34–44, New York, NY, USA, 2011. ACM.

[26] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. CAV*, 2006.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[28] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, August 2009.

[29] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[30] N. Tillmann, J. de Halleux, and T. Xie. Pex4fun: Teaching and learning computer science via social gaming. In *Proc. CSEET*, CSEET '11, 2011.

[31] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *Proc. CHI*, April 2004.

[32] E. Y. C. Wong, A. T. S. Chan, and H.-V. Leong. Efficient management of XML contents over wireless environment by Xstream. In *Proc. SAC*, pages 1122–1127, 2004.

[33] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, May 2011.

[34] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, Sept. 2010.