The Dart, the Psyco, and the Doop

Concolic Execution in Java PathFinder and its Applications

Marko Dimjašević
School of Computing, University of Utah, USA
marko@cs.utah.edu

Dimitra Giannakopoulou
NASA Ames Research Center, Moffett Field, CA, USA
dimitra.giannakopoulou@nasa.gov

Falk Howar*
IPSSE, TU Clausthal, Goslar, Germany
falk.howar@tu-clausthal.de

Malte Isberner*
TU Dortmund University, Dortmund, Germany
malte.isberner@udo.edu

Zvonimir Rakamarić*
School of Computing, University of Utah, USA
zvonimir@cs.utah.edu

Vishwanath Raman*
FireEye Inc., USA
vishwa.raman@gmail.com

ABSTRACT

JDart is a concolic execution extension for Java PathFinder. Concolic execution executes programs with concrete values while recording symbolic constraints. In this way, it combines the benefits of fast concrete execution, with the possibility of generating new concrete values, triggered by symbolic constraints, in order to exercise additional, potentially rare, program behaviors. As is typical with concolic execution engines, JDart can be used for test-case generation. Beyond this basic mode, it has also been used as a component of other tools. In this paper, we describe the main features of JDart, provide usage examples, and give an overview of applications that use JDart. We particularly concentrate on our efforts into making JDart robust enough to handle large, complex systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification, Experimentation

Keywords

Concolic Execution, White-box Testing

1. INTRODUCTION

In this paper we present JDart, a concolic execution extension for Java PathFinder. Concolic execution [10, 16] executes programs with concrete values while recording symbolic constraints for execution paths. In this way, it combines the benefits of fast concrete execution, with the possibility of generating new concrete values, triggered by symbolic constraints, in order to exercise additional, potentially rare, program behaviors.

We describe the main features of the tool, and its usage modes, through examples. We particularly concentrate on our efforts into making JDart robust enough to handle large, complex systems. The AutoResolver software for prediction and resolution of airplane loss of separation developed at NASA Ames has been our main driver for striving towards robustness and efficiency [8].

JDart can be used in many modes. As is typical with concolic execution engines, it can be used for test-case generation. Beyond this basic mode, JDart has also been used as a component of other tools. The Psyco tool uses JDart in combination with automata learning to automatically compute interfaces of software components implemented in Java. The Doop tool uses JDart together with random techniques for more efficient test-case generation.

The paper is organized as follows. Section 2 gives an overview of concolic execution. Section 3 describes the main features of JDart. JDart's role in the Psyco and Doop tools, and in the case study on AutoResolver, is presented in Section 4. Related work is presented in Section 5.

2. CONCOLIC EXECUTION

JDart [9, 12] is a concolic execution [10, 21] engine based on the Java PathFinder[1] framework [23]. It can be used to generate test cases exercising a large number of paths in a program. The tool executes Java programs with concrete and symbolic values at the same time, and records symbolic constraints describing the decisions (i.e., conditional branches taken) along a particular execution path. Combined, these path constraints form a *constraints tree*. The constraints tree is continually augmented by trying to exercise paths to unexplored branches. Concrete data values for exercising these paths are generated by a constraint solver. We will explain the central ideas of concolic execution as we use it in JDart on our running example shown in Fig. 1.

All parameters of the analyzed method are treated symbolically. This means that their values, as well as all decisions involving them, are recorded during execution. In the example, parameter `i` will hence be treated symbolically. For the initial concrete execution of the analyzed method `test()`, JDart uses the value found on the stack, which is `0` in the example. Instance fields are not treated symbolically in the default configuration of JDart.

Executing the method with a value of `0` for `i` does not trigger the assertion failure as `i <= 100`. Because `i` is symbolic, we still record this check, and add it to the *constraints tree*. The resulting partial constraints tree is shown in Fig. 2 (left): the *false* branch of the condition $i > 100$ (note that x is not being treated symbolically) contains the result "ok", and a valuation of the symbolic

*These authors did this work while at Carnegie Mellon University.

[1] `http://babelfish.arc.nasa.gov/trac/jpf`

```
public class Example {
  private int x;
  public Example(int x) {
    this.x = x;
  }
  public int test(int i) {
    if (i > x)
      assert false;
    int tmp = x;
    x += i;
    return tmp;
  }
  public static void main(String[] args) {
    Example e = new Example(100);
    System.out.println(e.test(0));
  }
}
```

Figure 1: Simple Java example. Method `test()` compares parameter `i` to field `x` and can lead to an assertion failure.

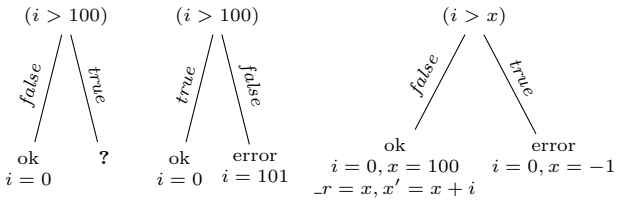

Figure 2: Different constraints trees for our running example. Leafs show program state, pre- and postcondition.

variables that allows exercising the corresponding path (in this case the initial configuration, $i = 0$).

However, the constraints tree also contains an unexplored branch, namely the *true* branch. Concolic execution now attempts at exercising this branch, by generating a valuation satisfying the *path constraint* $i > 100$, usually using a Satisfiability Modulo Theories (SMT) solver. SMT solvers provide decision procedures for first-order logical formulas of predicates from different theories (e.g., integer numbers or bit vectors). Such a solver could generate the valuation $i = 101$. The program is now rewound to the state were the analyzed method `test()` was entered. As the parameter `i` is treated symbolically, the corresponding stack contents are now changed to the value `101`, and the method is executed again. This time, the assertion failure is triggered. We augment the constraints tree by recording the outcome "error" along with the corresponding valuation $i = 101$ (Fig. 2 (middle)). As the constraints tree now no longer contains any nodes labeled by "?", concolic execution terminates.

Instance Fields and Return Values. By default only parameters are treated symbolically. However, the symbolic treatment can be extended to instance fields (e.g., `this.x`) and return values as well. Fig. 2 (right) shows the resulting constraints tree for symbolic values of `i` and `this.x`. The valuations labeling the leaves now contain values for i and x. Furthermore, the return value $\_r$ as well as the postcondition (the state of the instance after execution of the method) are given as symbolic expressions over i and x.

3. JPF-JDART

An architectural overview of JDART is depicted in Fig. 3. As mentioned before, JDART is built on top of the Java PathFinder framework. Through a mechanism called *instruction factory*, Java PathFinder allows for defining custom semantics of JVM bytecode instructions. JDART's instruction factory takes care of keeping track of symbolic expressions for primitive stack values and object fields. For example, if one of the operands of an `iadd` instruction (addition of two integers on the stack) is annotated with a symbolic expression (e.g., x), the result of that operation that is pushed on the operand stack is also annotated with a symbolic expression (e.g., $x+5$). Additionally, conditional branch instructions (e.g., `ifge` — branch if stack value is greater or equal to zero) are recorded and used to extend the constraints tree on-the-fly. Note that during a single execution of the application no exploration is performed. Therefore, every newly added branch not corresponding to the branch taken in the current run will be flagged as "unknown" ("?", cf. Fig. 2).

After a single run of the application has finished, JDART begins *exploring* the constraints tree. It does so by selecting a node flagged as "unknown", and aggregating the *path constraint* for this node, i.e., conjoining all conditions encountered on the path from the root of the tree to this node (note that the condition is negated if the branch taken is labeled by *false*). For example, the path constraint for the single unexplored node in the constraints tree depicted in Fig. 2 (left) is $\neg(i > 100)$. Such a path constraint is then passed to an SMT solver in order to obtain a satisfying valuation. By default, JDART uses Microsoft's Z3 solver [4], but its lightweight solver abstraction layer allows for easy integration with other solvers.

In case a satisfying valuation is obtained, the program is rewound to the entry of the method to be analyzed, using Java PathFinder's state-saving functionality. This program state is modified by changing the concrete values of all symbolic inputs to their respective values in the obtained valuation. Execution of the program in the above-described fashion is then repeated with the new concrete values. If no satisfying assignment is found, either because the path constraint is unsatisfiable, or because the solver failed (due to incompleteness), the node to be explored is labeled accordingly, and exploration chooses a different node to continue with. The whole procedure is iterated until there are no more nodes flagged "unknown" in the constraints tree, or until some user-defined termination criterion (e.g., timeout, number/percentage of branches covered) holds. Note that such an explicit termination criterion might become necessary in the presence of loops. In this case, the constraints tree might grow to infinite depth (although each single execution still terminates if the program itself does not get stuck in an infinite loop). JDART supports specific termination criteria for these cases. For instance, it can be instructed to make no further attempts to find a satisfying valuation for a loop condition if the number of loop iterations has reached a given threshold. This is similar in style to the common technique of *bounded model checking* [2].

The following paragraphs briefly describe some key features of JDART. As our main case study that guided the development of JDART was an actual flight-critical system (AUTORESOLVER; we discuss the corresponding case study in Section 4), our main goals were *robustness* and *scalability*.

Approximating Floating-Point Arithmetic. Flight-critical systems make heavy use of floating-point calculations, which at the time of publication are not properly and scalably supported in any SMT solver available to us. Therefore, similarly to previous work [13, 1], JDART approximates floating-point values using reals while at the same time employing additional measures to get

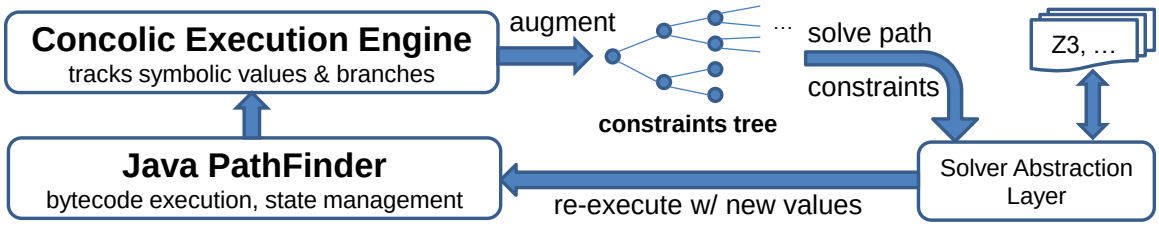

Figure 3: Architectural overview of JDart.

maximum benefits from incorrect solutions due to the unsound approximation.

Handling Native Calls. Native calls to methods outside of the JVM (e.g., in C libraries) can be handled by Java PathFinder thanks to the recent `jpf-nhandler` extension.[2] This, however, incurs a loss of symbolic information, which JDART handles using several strategies—representing the results as uninterpreted functions, merely concretizing them, or translating their effect to user-defined constraints.

Dealing with Incorrect Solutions. Simply discarding incorrect solutions that are caused by mismatches between the program semantics and solver theory would be a huge waste of computation time. Instead, JDART analyzes every solution and tests if it can nonetheless be used to exercise *some* (even if not the originally intended) path that is still unexplored in the program.

Selective Exploration. JDART can be configured to explore the path-space *selectively* by suspending (resp., resuming) exploration upon method entry (resp., exit). Exploration can be specified to start when a method of a particular class of interest is invoked, and suspended again during methods of uninteresting classes. This mechanism is complemented by the possibility of specifying sets of input values for replay; JDART replays these cases and explores according to the above criteria.

Using JDart. JDART comes with a number of examples that highlight the implemented features and configuration options. All examples are in the `src/examples/` folder of the JDART repository. Each example comes with the corresponding `.jpf` file to be run with Java PathFinder.

Here, we present three small examples, showing the behavior of JDART on a simple method, on a method with a native trigonometric function, and when producing method summaries.

```
public void foo(int i, boolean b) {
  if (i > 200000)
    if (b == false)
      assert false;
}

-('i' <= 200000)
  |-[+]_/OK: [  ]
  +-[-]-'b'
        |-[+]_/OK: [  ]
        +-[-]_/ERROR: java.lang.AssertionError
```

Figure 4: Java method with `int` and `boolean` parameters and the constraints tree produced by JDart.

[2] `https://bitbucket.org/nastaran/jpf-nhandler`

Fig. 4 shows a Java method with two primitive parameters and two simple tests on these parameters, as well as the corresponding constraints tree produced by JDART. The constraints tree has one inner node for each condition in the code and three leaves. Since the method is of type `void`, ok-paths do not show post-conditions. The error-path is annotated with the class of the found error. We do not show the concrete inputs that were used to exercise these paths; JDART can be configured to display these values or generate unit-tests based on them.

```
public void bar(double d) {
  if (1.1 <= Math.sqrt(d))
    if (Math.sin(d) > 0.0)
      assert false;
}

-(1.1 > 'sqrt'('d'))
  |-[+]_/OK: [  ]
  +-[-]-('sin'('d') <= 0.0)
        |-[+]_/OK: [  ]
        +-[-]_/ERROR: java.lang.AssertionError
```

Figure 5: Java method with native math operations and the corresponding constraints tree produced by JDart.

Fig. 5 shows a Java method that calls `sqrt` and `sin` as subroutines. In this example, JDART was configured to use linear interpolations of these functions when searching for concrete values for new execution paths. This heuristic works for this example as the constraints tree shows. JDART can be configured to use other strategies for finding solutions in the presence of native method calls: one can, e.g., use bounds on the domain of functions (as assumptions), bounds on the range of functions, or simply ignore these parts of constraints. Other strategies can be implemented easily in JDART as customized solvers (a detailed example can be found in the repository).

Fig. 6 shows the input and output of JDART when generating method summaries. In this case, the fields of the analyzed class are annotated as symbolic, indicating that those fields are to be included in the symbolic analysis. As a result, the fields become visible in the path constraints and postconditions in the constraints tree. Note that the `@Symbolic` annotation in the above example is a convenient way to flag instance fields as symbolic. However, this can also be specified in configuration files, eliminating the need to modify the source code of the programs to be analyzed.

JDART has many more features and configuration options, such as for controlling conditional exploration. It can deal with simple JAVA objects and arrays of fixed size symbolically. All primitive types are supported. The project repository contains a Wiki that documents its stable features and options.

4. APPLICATIONS

In this section, we introduce several tools and case studies in which JDART is used.

```
public class Protocol {
  @Symbolic("true") private int buffer_empty = 1;
  @Symbolic("true") private int expect = 0;
  public void recv_ack(int value) {
    if (buffer_empty==1)
      assert false;
    else
      if (value == (((expect-1) + 2) % 2))
        buffer_empty = 1-buffer_empty;
      else
        assert false;
  }
}

-('this.buffer_empty' != 1)
  |-[+]-('value' != (('this.expect' - 1) + 2) % 2))
  |      |-[+]_/ERROR: java.lang.AssertionError
  |      +-[-]_/OK: [ this.buffer_empty:=
                        (1 - 'this.buffer_empty')  ]
  +-[-]_/ERROR: java.lang.AssertionError
```

Figure 6: Simple protocol class with symbolic fields and the constraints tree produced by JDart for method `recv_ack(int value)`.

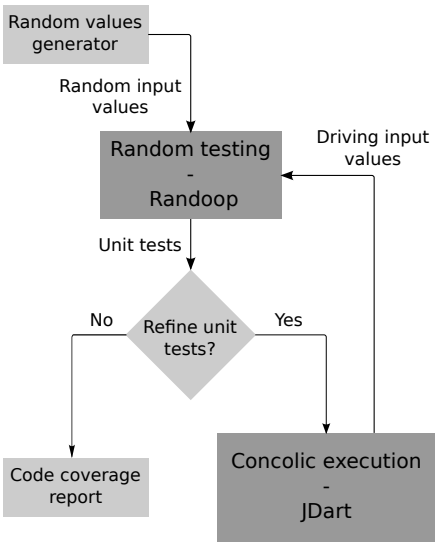

Figure 7: The workflow of Doop. The algorithm in Doop combines concolic execution and feedback-directed random testing.

Doop. Doop [5] is a testing tool with the workflow illustrated in Fig. 7. The tool combines two approaches: feedback-directed random testing [17] (also called Randoop) and concolic execution. The reason the two approaches are combined is to give better code coverage than each approach does on its own by mutually overcoming their respective shortcomings. For example, Randoop can generate heap objects, which is very important for an object-oriented language like Java, but it performs poorly in generating primitive values that would drive a program execution through branching statements. JDart, as noted above, cannot generate heap objects, but systematically finds primitive values that drive the execution through branches Randoop is unlikely to cover.

In our combination of the approaches, Randoop is used for global exploration (at the level of objects/classes) of the program state space, while JDart is used for local exploration (at the level of methods). The random testing part of Doop generates sequences of method calls and object constructors called drivers. Drivers reach a broad set of states, but miss numerous local states because they are generated in a random fashion. In turn, Doop takes each such driver and replaces concrete primitive input parameter values of methods with symbolic values. The replacement enables JDart to perform concolic execution on the drivers, exploring hard-to-reach local sites of the state space and yielding values that lead to them.

The described combination is repeated in a loop by providing primitive values obtained from JDart as input to another round of Randoop. Now the new round has the primitive values specific to the drivers of the previous round, which makes it possible for Randoop to reach even deeper states. The final output of Doop are unit tests generated from the mentioned drivers and input parameter values.

Psyco. In our previous work [9, 12], we present Psyco, a tool for generating temporal interfaces for components that include methods with parameters. The generated interfaces are finite-state automata whose transitions are labeled with method names and guarded with constraints on the corresponding method parameters. The guards partition the input spaces of parameters, and enable a more precise characterization of legal orderings than was previously possible in a fully automatic fashion. Psyco uses automata learning to create sequences of calls to methods of a component; these sequences are then turned into programs. JDart is used to analyze these programs. From the resulting path constraints, Psyco can extract guards for transitions in the generated interfaces.

Testing Flight-Critical Systems. In recent work [8], we present a case study in which we used JDart to generate test cases for the proto-typical implementation of a flight-critical system. This case study was the main driver for the development of the particular set of features that are unique to JDart—scalability and selective exploration of sequential programs where heap does not play a major role.

The Next Generation Air Transportation System (NextGen) advocates the use of innovative algorithms and software to address the increasing load on air-traffic control. AutoResolver [6] is a large, complex NextGen component that provides separation assurance between multiple airplanes up to 20 minutes ahead of time. The prototypical implementation of AutoResolver, developed at NASA Ames, consists of 199 Java classes with 1698 methods. That amounts to over 43 KLOC. The average McCabe cyclomatic complexity is 3.661; central methods for conflict resolution have the complexity of up to 82.

The input space of AutoResolver consists of airplane trajectories, each trajectory being a sequence of hundreds of points in the three-dimensional space. Generating meaningful test cases for AutoResolver that cover its behavioral space to a satisfactory degree is a major challenge.

We have developed a framework for testing AutoResolver that provides parameterized test drivers. These test drivers create conflict scenarios (e.g., trajectories of two airplanes in level flight). Each scenario has some primitive parameters to configure the details of a test case, such as the altitudes and speeds of the two airplanes. Automated test case generation tools exercise AutoResolver through these scenarios.

In the first phase of this project, we were mainly interested in generating test inputs of high quality. Using JDart, we gener-

ated test cases that reached branches in the code that were not reached by the black-box method we also used for generating test cases. However, JDART covered only a few additonal branches since we restricted the domains of parameters to values common in practice, e.g., altitudes that are multiples of 1,000 ft. Detailed results are presented in our recent paper [8].

We are currently developing test oracles for the AUTORESOLVER testing framework. Moreover, we are adding infrastructure for checking oracles at run time, and for generating reports of violations for the AUTORESOLVER developers. This will allow for finding unexpected system behavior with JDART.

5. RELATED WORK

Concolic execution [10, 21] is a well-known technique implemented by many automatic testing tools (e.g., [3, 7, 11, 22]). For example, SAGE [11] is a white-box fuzzer for security testing based on concolic execution, and applied successfully to large Microsoft systems such as media players and image processors.

There have been previous attempts to build a scalable concolic execution engine for Java. jCUTE [20] is the first implemented concolic execution engine for Java. Similarly to JDART, jFuzz [14] is a concolic white-box fuzzer implemented on top of Java PathFinder. More recently, LCT [15] implemented concolic execution via Java bytecode instrumentation, and the tool also supports distributed exploration. Unfortunately, none of these tools are under active development, and they also do not support advanced features required for handling AUTORESOLVER such as floating-point arithmetic. Hence, to be able to successfully apply concolic testing on AUTORESOLVER, we implemented JDART.

Symbolic PathFinder [19] is a Java PathFinder extension similar to JDART. At its core, the tool implements symbolic execution, albeit is still capable of switching to concrete values in the spirit of concolic execution [18]. That enables it to deal with limitations of constraints solvers (e.g., non-linear constraints).

To conclude, we implemented JDART as a true concolic engine targeting the types of applications we are dealing with; we have had full control over its development and hence we created a robust, efficient, and flexible tool that we leveraged in several related projects.

6. WORKING WITH JDART

JDART is currently in the process of being made available as open source software. Until it is released, it is possible to obtain access to JDART by contacting the authors of this paper. JDART is currently hosted as a private project on Bitbucket.[3]

Acknowledgement

This research was partly sponsored by United States National Aeronautics and Space Administration (NASA) under Prime Contract No. NNA10DE60C.

7. REFERENCES

[1] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560, 2013.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[5] M. Dimjašević and Z. Rakamarić. JPF-Doop: Combining concolic and random testing for Java. In *JPF Workshop*, 2013. Extended abstract.

[6] H. Erzberger, T. A. Lauderdale, and Y.-C. Chu. Automated conflict resolution, arrival management and weather avoidance for ATM. In *International Congress of the Aeronautical Sciences*, 2010.

[7] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *ICSE*, pages 132–141, 2013.

[8] D. Giannakopoulou, F. Howar, M. Isberner, T. Lauderdale, Z. Rakamarić, and V. Raman. Taming test inputs for separation assurance. In *ASE*, 2014.

[9] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *SAS*, pages 248–264, 2012.

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

[11] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[12] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *ISSTA*, pages 268–279, 2013.

[13] F. Ivančić, M. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, pages 49–58, 2010.

[14] K. Jayaraman, D. Harvison, and V. Ganesh. jFuzz: A concolic whitebox fuzzer for Java. In *NFM*, 2009.

[15] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä. LCT: An open source concolic testing tool for Java programs. In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pages 75–80, 2011.

[16] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[17] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[18] C. S. Pasareanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA*, pages 34–44, 2011.

[19] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26, 2008.

[20] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

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

[22] N. Tillmann and J. d. Halleux. Pex—white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[23] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[3] `https://bitbucket.org/psycopaths/jpf-jdart`