JDart: A Dynamic Symbolic Analysis Framework*

Kasper Luckow[1], Marko Dimjašević[2], Dimitra Giannakopoulou[3], Falk Howar[4], Malte Isberner[5], Temesghen Kahsai[1,3], Zvonimir Rakamarić[2], and Vishwanath Raman[6]

[1] Carnegie Mellon University Silicon Valley, Mountain View, CA, USA
[2] School of Computing, University of Utah, Salt Lake City, UT, USA
[3] NASA Ames Research Center, Moffett Field, CA, USA
[4] IPSSE, TU Clausthal, Goslar, Germany
[5] TU Dortmund University, Dortmund, Germany
[6] StackRox Inc, Mountain View, CA, USA

Abstract. We describe JDart, a dynamic symbolic analysis framework for Java. A distinguishing feature of JDart is its modular architecture: the main component that performs dynamic exploration communicates with a component that efficiently constructs constraints and that interfaces with constraint solvers. These components can easily be extended or modified to support multiple constraint solvers or different exploration strategies. Moreover, JDart has been engineered for robustness, driven by the need to handle complex NASA software. These characteristics, together with its recent open sourcing, make JDart an ideal platform for research and experimentation. In the current release, JDart supports the CORAL, SMTInterpol, and Z3 solvers, and is able to handle NASA software with constraints containing bit operations, floating point arithmetic, and complex arithmetic operations (e.g., trigonometric and nonlinear). We illustrate how JDart has been used to support other analysis techniques, such as automated interface generation and testing of libraries. Finally, we demonstrate the versatility and effectiveness of JDart, and compare it with state-of-the-art dynamic or pure symbolic execution engines through an extensive experimental evaluation.

1 Introduction

JDart is a dynamic symbolic analysis framework for Java, under development at CMU and NASA Ames Research Center since 2010. Our main goal in developing JDart has been to build a dynamic symbolic analysis tool that can be applied to industrial scale software, including complex NASA systems. To reach this goal, we faced challenges that required a significant amount of design and engineering effort by several researchers over multiple years.

Our main design guideline has been to strive for a modular and extensible architecture. As such, our vision has been for JDart to be a platform for experimentation not only in symbolic analysis, but also in other areas of research

* Based upon work funded and supported by NASA Contract No. NNX14AI09G.

that may use symbolic analysis as a component. JDart has now reached a level of robustness and efficiency that makes it ready for use by a wider community of researchers and practitioners. With the opportunity of JDart's recent open sourcing [1], this paper describes the characteristics of the tool that make it unique in its field. Moreover, it presents an extensive experimental evaluation of JDart, comparing it with state-of-the-art tools on a variety of benchmarks, in order to provide interested users with an understanding of its strengths and weaknesses relative to other similar frameworks.

As mentioned, the key distinguishing feature of JDart is its modular architecture. The two main components of JDart are the *Executor* and the *Explorer*. The *Executor* executes the analyzed program and records symbolic constraints on data values. It is currently realized as an extension to the Java PathFinder framework [19, 33]. The *Explorer* determines the exploration strategy to be applied. It uses the constraints library JConstraints (developed as part of the JDart project) as an abstraction layer for efficiently encoding symbolic path constraints and provides an interface for a variety of constraint solvers. JDart's current release supports the CORAL [30], SMTInterpol [4] and Z3 [22] solvers. Furthermore, JDart provides several useful extensions, such as method summarization and jUnit test case generation, that leverage the results of dynamic symbolic analysis. Note that all these components of JDart can be configured, extended, or replaced.

In addition to being easily extensible and configurable, JDart can also be used as a symbolic execution component within other tools. In particular, we discuss two such uses of JDart: Psyco [13, 16] and JPF-Doop [7]. The former is a tool that uses automata learning and dynamic symbolic execution to automatically generate extended interfaces for Java library components. The latter is a tool that combines random feedback-directed generation of method sequences with dynamic symbolic execution for automatic testing of Java libraries.

Among benchmarks that we use to showcase the capabilities of JDart, we emphasize a NASA case study that has been our main challenge and driver for its development over the years. JDart has been used to generate tests for the AutoResolver system — a large and complex air-traffic control tool that predicts and resolves loss of separation for commercial aircraft [9, 12]. Within this context, JDart has demonstrated the capability to handle programs with more than 20 KLOC containing bit operations, floating point and non-linear arithmetic operations (e.g., trigonometric), and native methods from `java.lang.Math`. Our experimental evaluation also demonstrates that, from the set of available and maintained symbolic execution tools, JDart is the most stable and robust.

Note that a preliminary version of JDart was presented earlier in [6]. Since then, we have added support for additional constraint solvers and exploration strategies, which are included in the open source release and discussed in this paper. We have also conducted a thorough evaluation of JDart on multiple benchmarks and compared it to state-of-the art tools.

[1] JDart is available on GitHub: `https://github.com/psycopaths/jdart`

```
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));
}
```

Fig. 1: Simple JAVA software under test (SUT) example. Method `test()` compares parameter `i` to field `x` and can lead to an assertion failure.

Synopsis. The rest of the paper is organized as follows. Sec. 2 introduces dynamic symbolic execution. Sec. 3 describes the architecture of JDART and its usage in other analysis techniques. Sec. 4 discusses features of JDART and related tools. Sec. 5 gives an extensive experimental evaluation with benchmarks that include NASA examples. Our conclusions are discussed in Sec. 6.

2 Dynamic Symbolic Execution

Dynamic symbolic analysis is a program analysis technique that executes programs with *concrete* and *symbolic* inputs at the same time. It maintains a *path constraint*, i.e., a conjunction of symbolic expressions over the inputs that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Combined execution paths form a *constraints tree*, which 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 explain how this works in JDART using the example shown in Fig. 1.

Dynamic symbolic execution treats some (or all) parameters of an analyzed method symbolically. This means that their values, as well as all decisions involving them, are recorded during execution. In the example of Fig. 1, parameter `i` is treated as a symbolic value. For the initial concrete execution of the analyzed method `test()`, JDART uses the value found on the stack, which is `0`. 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 because `i <= 100`. Since `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 variables that allows exercising the corresponding path (in this case the initial configuration, $i = 0$).

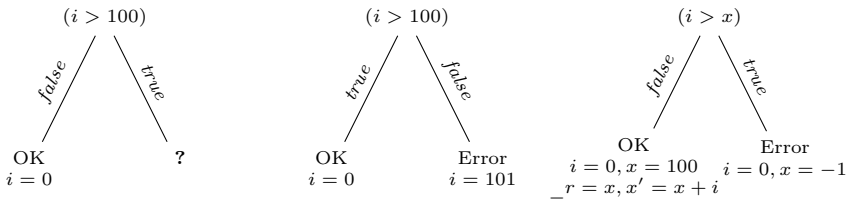

Fig. 2: Different constraints trees for the example in Fig. 1. Leafs show program states as well as the pre- and post-conditions of paths.

However, the constraints tree also contains an unexplored branch, namely the *true* branch. Dynamic symbolic execution now attempts to exercise this branch, by generating a valuation satisfying the *path constraint* $i > 100$, usually using an SMT solver. SMT solvers provide decision procedures for first-order logical formulas of predicates from different theories (e.g., integer numbers, bit vectors or arrays). Given a set of constraints, the solver will generate a satisfying assignment that makes the constraint satisfiable. In our example it could generate the assignment $i = 101$. The program is now rewound to the state where the analyzed method `test()` was entered. As 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. JDart augments 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 "?", dynamic symbolic execution terminates.

By default, JDart treats only parameters symbolically. However, the symbolic treatment can be extended to *instance fields* (e.g., `this.x`) and *return values* as well. For example, Fig. 2 (right) shows the resulting constraints tree for symbolic values of `i` and `this.x`. The return value $\_r$ as well as the post-condition (the state of the instance after execution of the method) are given as symbolic expressions over i and x.

3 JDart

The development of JDart has been driven by two main goals. The primary goal has been to build a symbolic analysis framework that is robust enough to handle industrial scale software. More precisely, it has to be able to execute industrial software without crashing, deal with long execution paths and complex path constraints. The second objective has been to build a modular and extensible platform that can be used for the implementation and evaluation of novel ideas in dynamic symbolic execution. For example, JDart is designed to allow for easy replacement of all of its components: it supports different and combined constraint solvers, and several exploration strategies and termination criteria.

This section presents the modular architecture of JDart, and discusses its main components and extension points. It subsequently describes existing uses of JDart as a component within other research tools.

```
target=Example                              // SUT class with main
concolic.method.test=Example.test(i:int) // Method declaration
concolic.method=test                        // Selection of target method
concolic.method.test.constraints=(i>=0)  // Assumptions on inputs
symbolic.dp=z3                              // constraint solver
```

Fig. 3: Configuration of JDART for the Example from Fig. 1.

3.1 Architecture

JDART executes JAVA Bytecode programs and performs a dynamic symbolic analysis of specific methods in these programs. JDART also implements extensions that build upon the results of a dynamic symbolic analysis:

- The *Method Summarizer* generates fully abstract method summaries for analyzed methods. In the generated summaries, class members, input parameters, and return values are represented symbolically.
- The *Test Suite Generator* generates JUNIT test suites that exercise all the program paths found by JDART.

Fig. 3 illustrates a basic configuration of JDART (no extensions included) for the example of Fig. 1. The configuration sets the system under test to class `Example`, and specifies method `test(i:int)` of the same class as the target of the analysis. The last two lines tell JDART to explore the target method only for parameter values $i >= 0$ and to use Z3 for solving constraints.

During dynamic symbolic analysis, JDART uses two main components to iteratively execute the target method, to record and explore symbolic constraints, and to find new concrete data values for new executions: Fig. 4 depicts the modular architecture of JDART. The basis (at the bottom) is the *Executor* that executes the analyzed program and records symbolic constraints on data values. The *Explorer* organizes recorded path constraints into a constraints tree, and decides which paths to explore next, and when to stop exploration. The Explorer uses the JCONSTRAINTS library to integrate different constraint solvers that can be used in finding concrete data values for symbolic paths constraints.

3.2 Executor

The Executor runs a target program and executes an analyzed method with different concrete data values for method parameters and class members. It also records symbolic constraints for program paths. Currently, JDART uses the software model checker Java PathFinder (JPF) for the execution of JAVA Bytecode programs. JDART uses two extension points of JPF.

Setting Concrete Values. JPF uses "choice generators" to mark points in an execution to which JPF back tracks during state-space exploration. JDART implements a choice generator that sets parameter values of methods that are analyzed symbolically.

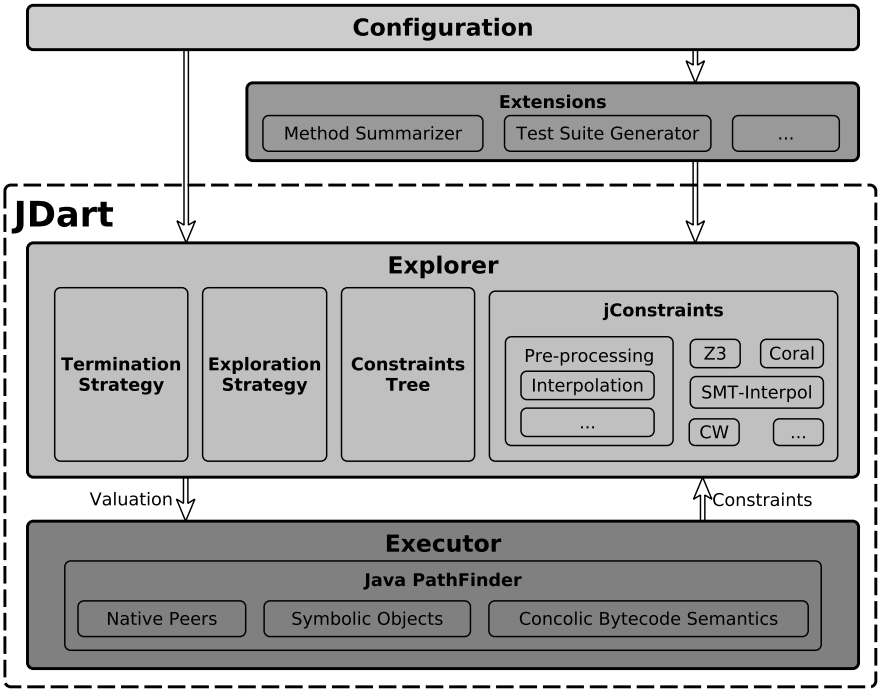

Fig. 4: Architecture of JDart.

Recording Symbolic Constraints. JPF extensions can provide custom bytecode implementations. JDart adds concolic semantics to the Java Bytecodes that perform concrete and symbolic operations simultaneously, while also recording path constraints. Using JPF as an execution platform has several benefits. For example, is easy to integrate other JPF extensions in JDart (e.g., for dealing with native code, or for recording test coverage). Moreover, JPF provides easy access to all objects on the heap and stack, as well as to many other elements and facilities of the JVM such as stack frames and class loading. On the other hand, using a full-blown custom JVM for execution has an impact on performance. This is one of the reasons why we are keeping the integration with JPF as loose as possible. JDart has been built with the possibility of changing the underlying execution environment from JPF to more light-weight instrumentation, as is the case with other similar frameworks, such as PEX [32] or JCute [26].

3.3 Explorer

The Explorer organizes recorded constraints into a constraints tree, decides which parts of the program to explore, when to stop, and how to solve constraints for new concrete input values.

Exploration. In order to hit interesting paths quickly when analyzing large systems, JDart needs to be able to limit exploration to certain paths. JDart provides configuration options for specifying multiple pre-determined vectors of input values from which the exploration is started. It also allows the user to specify assumptions on input parameters as symbolic constraints. JDart will then only explore a method within the limits of those assumptions. Finally, JDart can be configured to simply skip exploration of certain parts of a program (e.g., after entering a specific method) — i.e., it supports suspending/resuming exploration based on method level descriptions. It also allows skipping exploration after a certain depth.

Termination. For industry-scale systems, it is often not possible to run an analysis to completion. Sometimes one may even be interested in recording the path constraint of a single program path (cf., e.g., Microsoft's SAGE [15]). JDart provides an interface for implementing customized termination strategies. So far, it provides strategies for terminating after a fixed number of paths, or for terminating after a fixed amount of time.

Constraint Solvers. In real world systems, path constraints can be long and complex and may contain trigonometric or elementary functions, which may challenge any state-of-the-art constraint solver. JDart provides several techniques and extension points for optimizing constraints, e.g., by simplifying path constraints, by adding auxiliary definitions and/or interpolation that help solving complex constraints, and by using specialized solvers.

Constraints Tree. Finally, it is important to guarantee that progress is made when only approximating Java semantics in solvers. Sometimes a solution suggested by a solver may not be valid for a Java Bytecode program. JDart tests all valuations produced by a decision procedure on the constraints tree by evaluating path constraints with Java semantics before re-executing the program with a new valuation (this is a feature provided by JConstraints, as explained later in this section).

Potential Extensions. In extending the explorer, we are considering to implement concolic heuristics for dealing with complex constraints, and to use coverage metrics (e.g., branch coverage or MC/DC) to prioritize exploration of decisions that may increase the selected coverage. Using JPF, in the future it will also be possible to add support for concurrent programs.

3.4 JConstraints

JConstraints is a constraint solver abstraction layer for Java. It provides an object representation for logic expressions, unified access to different SMT and interpolation solvers, and some useful tools and algorithms for working with constraints. While JConstraints has been developed for JDart, it is maintained as a stand-alone library that can be used independently. The use of a constraint solver abstraction layer for a programming language has several advantages, as

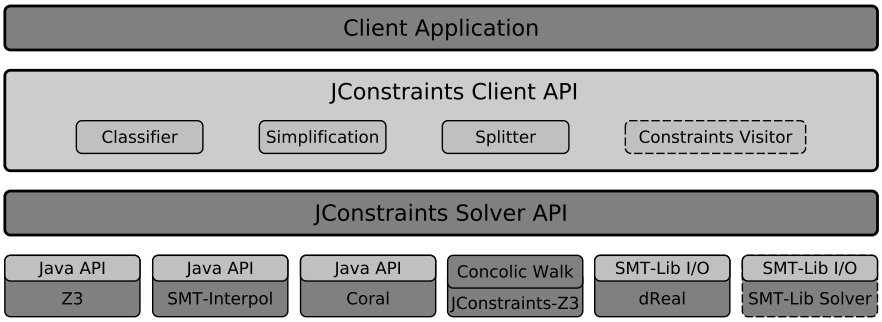

Fig. 5: The JConstraints architecture.

discussed here. This idea has also been explored by others, such as PySMT [11], which has recently been developed for Python.

The architecture of JConstraints is shown in Fig. 5: It consists of the basic library providing the object representation of logic and arithmetic expressions, the API definitions for solvers (for SMT solving and interpolation, or for incremental solving), and some basic utilities for working with expression objects (basic simplification, term replacement, and term evaluation). Plugins for connecting to different constraint solvers can be added easily by implementing a solver interface and taking care of translating between a solver-specific API and the object representation of JConstraints.

Currently, plugins exist for connecting to the SMT solver Z3 [22], the interpolation solver SMTInterpol [4], the meta-heuristic based constraint solver CORAL [30], and a solver that implements the Concolic Walk algorithm [8]. JConstraints uses the native interfaces for these solvers as they are much faster than file-based integration. It can also parse and export constraints in its own format and supports a subset of the SMT-LIB format [28] which enables connection to many constraint solvers that understand this format. For example, through the SMT-LIB format, we were able to experiment with using the dReal solver [10] for non-linear constraints in JDart.

JConstraints supports both Java and user-defined types for expressions. This enables it to record path constraints directly in terms of the analyzed program types and semantics, as opposed to the types supported by the constraint solver to be used. An advantage of this feature is that it is easy to validate solutions returned by constraint solvers by simply evaluating the path constraint stored by JConstraints with Java semantics.

3.5 Leveraging JDart

JDart is a mature and easy to use framework that has so far been leveraged in several tools.

Automatic Testing of Libraries. Previous work on Randoop [23] has shown that software libraries can often be effectively explored using *feedback-directed*

random testing, which generates test cases in the form of reasonable sequences of public method invocations. However, while Randoop excels at generating method sequences, its heuristic for selecting inputs for arguments of primitive data types is simplistic — these inputs are selected from a small pool of mostly randomly chosen values. This heuristic is often inadequate for reaching deep into the code of methods with many conditionals over primitive types such as integers. On the other hand, JDart's capabilities are orthogonal: it cannot generate sequences of method invocations, but it can explore deep code paths by leveraging the power of SMT solving. Hence, we implemented JPF-Doop to combine the two approaches [7].

JPF-Doop leverages Randoop to generate a collection of method sequences. Next, JPF-Doop converts all primitive-type input parameters into symbolic inputs in every generated method sequence. This in turn enables JDart to be executed on such method sequences, and its dynamic symbolic execution algorithm reaches deep paths within each method in a sequence. As a result, more paths, and consequently branches and lines of code, are often explored by JPF-Doop than by using the two tools in isolation [7].

Generating Interfaces of Software Components. Performing compositional software verification is key to achieving scalability to large systems. Generating interfaces for software components is an important sub-task of compositional software verification. In our previous work [13, 16], we introduced an algorithm (implemented in a tool called Psyco) for automatic generation of precise temporal interfaces of software components that include methods with parameters. Psyco generates interfaces in the form of finite-state automata, where transitions are labeled by method names as well as guarded by symbolic constraints over their parameters. It relies on JDart's capability for computing method summaries for the public methods of the analyzed component.

4 JDart and Related Frameworks

Dynamic symbolic execution [14, 27] is a well-known technique implemented by many automatic testing tools (e.g., [3, 15, 26, 32]). For example, SAGE [15] is a white-box fuzzer based on dynamic symbolic execution. SAGE has been routinely applied to large Microsoft systems, such as media players and image processors, where it has been successful in finding critical security bugs.

Several symbolic execution tools specifically target Java Bytecode programs. A number of them implement dynamic symbolic execution via Java Bytecode instrumentation. JCute [26], the first concolic execution engine for Java, uses Soot [29] for instrumentation, and uses lp_solve as a constraint solver. jCUTE is no longer maintained and the developers now seem to focus on a new tool, CATG [31]. CATG uses ASM [1] for instrumentation, and CVC4 [5] as a constraint solver. Another concolic engine, LCT [20], additionally supports distributed exploration. It uses Boolector and Yices for solving constraints, but does not currently have support for float and double primitive types.

Tool	Basic Technology	Symbolic Exploration of						Practicality							Solvers		
		Primitive	Arrays	Fields	Objects	Shape	Concurrency	Retains SUT	Parallel	Native Code	Long Paths	Replay	Summaries	Test Suites[5]	String	Non-linear	Trig./Elem.
JDart	Custom VM	✓	(✓)[2]	✓	✓			✓		✓	✓	✓	✓	✓		✓	✓
SPF		✓	(✓)[2]	✓	✓	✓	✓	✓		✓		✓		✓	✓	✓	✓
jFuzz †		✓	(✓)[23]	✓			✓	✓		✓				✓	✓	✓	
JCute †	Code Instr.	(✓)[1]	(✓)[2]	(✓)[4]	✓		✓				✓	✓		✓			
CATG		(✓)[1]	(✓)[2]	(✓)[4]							✓			✓	✓		
LCT		(✓)[1]	(✓)[2]	(✓)[4]	✓				✓		✓			✓			

[1]No float and double. [2]Only fixed size.
[3]Only `char[]`. [4]Symbolic inputs are injected by modifying SUT.
[5]Only for sequential programs. †No longer maintained.

Table 1: Comparing the features of JDart to other symbolic analysis tools.

A drawback of instrumentation-based tools is that instrumentation at the time of class loading is confined to the SUT. LCT for example does not by default instrument any of the standard Java libraries thus limiting concolic execution to the application classes. However, the instrumentation-based tools discussed above provide the possibility of using symbolic (and/or simplified) models for non-instrumented classes or using pre-instrumented core Java classes.

Several dynamic symbolic execution tools for Java are not based on instrumentation. For example, the concolic white-box fuzzer jFuzz [18] is based on Java PathFinder (as is JDart) and can thus explore core Java classes in the analysis without any extra prerequisites. Finally, Symbolic PathFinder [25] is a Java PathFinder extension similar to JDart. In fact, jFuzz reuses some of the core components of (albeit an older version of) SPF, notably the solver interface, and its implementations. While at its core SPF implements symbolic execution, it is also capable of switching to concrete values in the spirit of concolic execution [24]. That enables it to deal with limitations of constraints solvers (e.g., non-linear constraints).

Table 1 summarizes the main features of the tools discussed in this section. It can be seen that JDart supports a large number of features that are desirable in symbolic execution engines to accommodate analysis of industrial scale systems. On the other hand, JDart does not currently support programs with concurrency in contrast to SPF and JCute. Also, JDart does not feature a mechanism for dealing with unbounded input data structures such as lists and trees. SPF supports this through its lazy initialization mechanism [21]. Finally, JDart does not currently support a parallel exploration of the constraints tree. However, JDart's architecture provides a solid basis for future extensions towards supporting such features. In particular, some of the distinctive features of SPF are relatively easy to port to JDart given the common foundation of the two tools on JPF. In general, we expect that open sourcing will expedite extensions of JDart in new directions.

5 Experimental Evaluation

We base our evaluation of JDart on a comparison with SPF, CATG, LCT, and random testing. Random testing provides a baseline, while the other tools are representative of the state-of-the-art in symbolic analysis of Java Bytecode. We were not able to properly set up JCute and jFuzz despite investing a significant amount of effort. We note that they are no longer actively maintained. Our evaluation is performed on the following benchmarks:

AutoResolver is a sophisticated automated air-traffic conflict resolution system developed at the NASA Ames Research Center. It is envisioned to be a component of the Next Generation Air Transportation System (NextGen) for the US airspace. It features complex constraints arising, among others, from spherical geometry and great circle distance computations. We focus JDart on a single conflict scenario, using the test driver developed in previous work [12] that exposes a double-precision floating-point type controlling the heading difference between two aircraft at a collision point. Note that our coverage metrics take into account the entire AutoResolver code base consisting of approximately 20 KLOC of Java code.

MER Arbiter is derived from a flight component for the Mars Exploration Rover developed at NASA JPL. The arbiter module is based on a Simulink/Stateflow model translated into Java using the Polyglot framework [2].

TSAFE is a flight-critical system that seeks to provide separation assurance for multiple aircraft. It features complex, nonlinear floating-point arithmetic and constraints with transcendental functions.

TCAS is a component of a traffic collision avoidance system installed in aircraft; its operation is controlled by 12 inputs.

Raytracer is a component for rendering shades on surfaces. It performs a number of calculations on 3D vectors taking into account light and color objects.

WBS has 18 integer and boolean inputs controlling the update operation in a wheel brake system.

Minepump is a classic real-time system that performs monitoring and controlling of the fluid level and methane concentration in a mine shaft.

We use the following metrics: (i) analysis time; (ii) the quality of the symbolic exploration of a benchmark in terms of multiple coverage criteria, such as general coverage metrics (branch, instruction, line, method) and *behavioral coverage* captured by the absolute number of paths exercised; (iii) the *quality* of the test suite produced by the tools, i.e., the ratio of paths exercised while running the suite to the number of tests. Table 2 gives our experimental results.

Evaluation of Symbolic Analysis Tools. With a time cap of 1 hour, we monitor the analysis time and peak memory consumption for each tool to terminate and return a collection of input valuations. For a consistent comparison, we measure coverage of the generated valuations, as opposed to using output statistics of the tools. For each tool, we construct a jUnit test suite based on the produced valuations, which is then analyzed by the JaCoCo [17] coverage

Example	Tool	Solver	Run time [s]	Memory [MB]	Branch Cov. [%]	Instr. Cov. [%]	Line Cov. [%]	Method Cov. [%]	#Tests	#Sat. Paths	#OK	#Err.	#D/K
AutoResolver	**Totals**				**10,896**	**96,304**	**19,695**	**1,941**		**229**[§]			
	JDart	Z3	80	1,483	15	24	27	35	3	3	3	0	6,898
	SPF *	CORAL	2	215	0	0	0	0	0	0	0	0	0
	CATG [†]	CVC4	-	-	-	-	-	-	-	-	-	-	-
	LCT [†]	Boolector	-	-	-	-	-	-	-	-	-	-	-
	Random	-	80	-	16	24	27	35	1,969	229	229	0	214,982
MER	**Totals**				**576**	**11,047**	**2,234**	**635**		**1,248**			
	JDart	Z3	69	340	50	81	79	77	1,248	1,248	1,248	0	0
	SPF	Z3	90	761	50	81	79	77	1,248	1,248	1,248	0	0
	CATG	CVC4	crash	-	-	-	-	-	-	-	-	-	-
	LCT	Boolector	2,839	51	50	81	79	68	1,256	1,248	1,248	0	0
	Random	-	69	-	38	71	72	62	8,259	81	81	0	231
TSAFE	**Totals**				**20**	**137**	**26**	**4**		**21**[§]			
	JDart	CORAL	3	415	90	89	96	75	21	21	21	0	26
	SPF	CORAL	23	727	90	89	96	75	58	21	21	0	26
	CATG [†]	CVC4	-	-	-	-	-	-	-	-	-	-	-
	LCT[†]	Boolector	-	-	-	-	-	-	-	-	-	-	-
	Random	-	3	-	70	85	92	75	54,318	9	9	0	13
TCAS	**Totals**				**74**	**216**	**48**	**9**		**68**			
	JDart	Z3	<1	119	93	96	96	89	68	68	68	0	0
	SPF	Z3	5	118	61	72	92	78	68	36	36	0	80
	CATG	CVC4	33	88	93	96	96	89	68	68	68	0	0
	LCT	Boolector	32	77	93	96	96	89	68	68	68	0	0
	Random	-	1	-	5	19	29	22	19,067	1	1	0	3
Raytracer	**Totals**				**44**	**798**	**119**	**18**		**83**[§]			
	JDart	CORAL	51	414	86	94	92	94	80	80	80	0	64
	SPF [‡]	CORAL	1,524	350	82	94	92	94	548	83	83	0	158
	CATG [†]	CVC4	-	-	-	-	-	-	-	-	-	-	-
	LCT [†]	Boolector	-	-	-	-	-	-	-	-	-	-	-
	Random	-	51	-	68	79	81	89	186,953	80	80	0	26
WBS	**Totals**				**90**	**356**	**149**	**3**		**24**			
	JDart	Z3	<1	119	67	77	76	67	24	24	24	0	0
	SPF	Z3	1	118	67	77	76	67	24	24	24	0	0
	CATG	CVC4	8	62	67	77	76	67	24	24	24	0	0
	LCT	Boolector	6	68	67	77	76	67	24	24	24	0	0
	Random	-	1	-	39	50	50	67	190,715	5	5	0	6
Minepump	**Totals**				**100**	**552**	**139**	**43**		**1,200**			
	JDart	Z3	2	150	52	68	79	74	1,200	1,200	952	248	0
	SPF	Z3	48	212	52	682	79	74	1,200	1,200	952	248	0
	CATG	CVC4	420	64	52	68	79	74	1,200	1,200	952	248	0
	LCT	Boolector	522	83	52	68	79	74	1,729	1,200	952	248	0
	Random	-	2	-	52	68	79	74	39,966	1,200	952	248	0

*SPF with Z3 crashes. CORAL does not handle constraints with DOUBLE.ISNAN(DOUBLE), thus ignoring them.
†Does not support constraints with floating points.
‡With PSO heuristic (used with JDART), SPF does not finish <1h. Instead, the AVM heuristic is used.
§Max # of sat paths explored by any tool. The total # of sat paths might be higher due to *Don't know* paths.

Table 2: Experimental results. Numbers in bold font denote the total number of units (e.g., instructions or branches) for the respective benchmark. "-" represents when values do not apply, e.g., when an example is not supported by a tool.

measuring library. JACOCO generates a detailed report containing branch, instruction, line, and method coverage. On the other hand, behavioral coverage is not reported by standard code coverage libraries, so we measure it by replaying valuations with JDART, where JDART is run without a solver. JDART tracks the number of *unique satisfiable paths* that are exercised, as well as whether a path yields normal termination (*OK*) or an error state (*Error*) — assertion violation or uncaught exception. We chose to use dynamic symbolic analysis for this purpose because it additionally checks for validity of the valuations; as seen in the

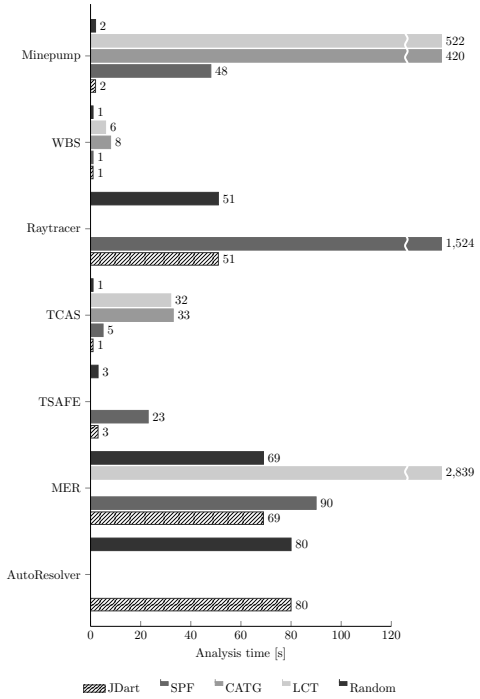

Fig. 6: Analysis time.

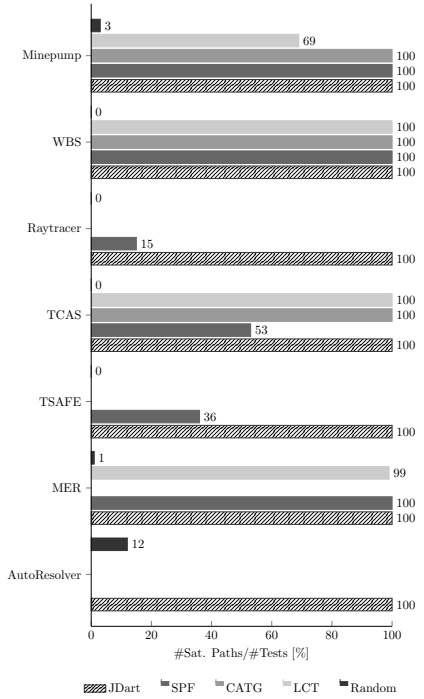

Fig. 7: Test suite quality.

TCAS example results for SPF, 68 valuations are produced, but only 36 of them are valid and contribute to path coverage.

We also keep track of the number of *potentially* unexplored subtrees/decisions (*D/K*, short for *Don't Know*); *D/K*s represent decisions in the constraints tree that are not covered by the test suite. For symbolic analysis tools, when they terminate, it means that the used solver was *inconclusive* as to the satisfiability of these decisions. Such situations arise due to, e.g., insufficient solver capabilities, constraints that are computationally intractable, or constraints from undecidable theories (containing non-linear or transcendental functions).

For JDart, we select the solver and configuration that yields the best test suite defined in terms of the above coverage metrics. Unless we found a better configuration for SPF, the same configuration is used for SPF. Other tools do not expose such rich set of configuration parameters or solver options.

Evaluation of Random Testing. For random test case generators, we set time out to match the analysis time of JDart. Input values are randomly selected from a uniform distribution from the value range of a particular parameter's data type. Note that our implementation of random testing is rather simplistic: constraining the input ranges according to domain knowledge and picking values from non-uniform distributions (e.g., from a known "usage profile") would likely increase its applicability.

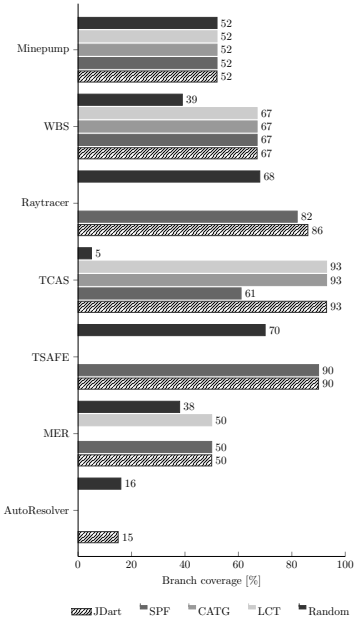

Fig. 8: Branch coverage.

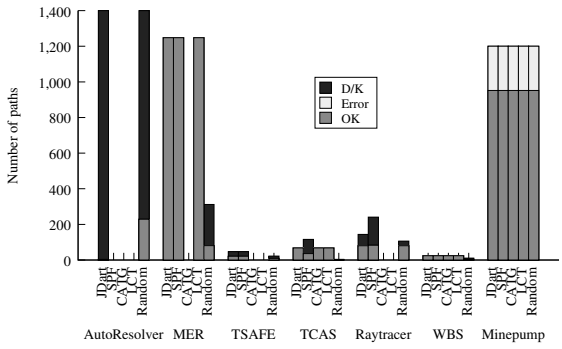

Fig. 9: Path coverage. JDART and random testing on AUTORESOLVER have 6,898 and 214,982 *Don't Know* paths, respectively.

Observation 1: Analysis Time and Path Coverage. JDART outperforms SPF, CATG, and LCT on all benchmarks in terms of analysis time — often by an order of magnitude (see Fig. 6). Furthermore, with path coverage being the primary metric for comparison, JDART provides at least as good results as the other tools, except for the Raytracer benchmark (where SPF performed slightly better) and the AUTORESOLVER benchmark (where Random Testing performed better). Fig. 9 summarizes these results. For Raytracer, SPF found three more *OK* paths. In this particular case, however, SPF was run with a slightly different configuration of CORAL that uses the Alternating Variable Method (AVM) meta-heuristic — JDART uses the Particle Swarm Optimization (PSO) meta-heuristic. If SPF is run with PSO, it does not terminate within 1 hour. On the other hand, if JDART is run with AVM, it performs worse than with PSO and covers only 38 distinct paths. As a side note, the longer analysis times of SPF might be attributed to the significant number of *D/K* paths.

Observation 2: Random Test Case Generation. Our experimental results demonstrate that random test case generation performs poorly on the benchmark suite. In particular, in WBS it covers only 5 different paths with a test suite containing 190,715 test cases. For AUTORESOLVER, random testing slightly outperforms JDART in terms of branch coverage (1 percentage point difference) at the expense of having to run 1,969 test cases (taking 80 seconds). In contrast, the test suite produced by JDART contains only 3 test cases (taking less than a second to run). Note that the coverage results for AUTORESOLVER are so low

because large submodules are not reachable from the entry point that only deals with a single conflict scenario.

Observation 3: Performance of Instrumentation-Based Tools. CATG obtains similar coverage results as JDART on the benchmarks it supports, but is several orders of magnitudes slower. This might be attributed to the concolic execution approach implemented in CATG, which (similar to JCUTE) allocates a process for each execution — CATG reruns the instrumented program for each explored path. JDART, on the other hand, harnesses the JPF infrastructure and perturbation facility to efficiently restore program states and generate new paths. LCT is also comparable to JDART on benchmarks that do not require symbolic floating-point reasoning, but like CATG it is much slower. Note that LCT supports parallel exploration that was not used in our experiments, which is a feature currently not supported in JDART.

All instrumentation-based tools employ a pre-processing step where a benchmark (and classes potentially referenced by it) need to be instrumented before the actual analysis can be performed. Our measured analysis times do not account for this step, which is often significant. For example, the instrumentation of MER with LCT takes 13 seconds. JDART avoids this by leveraging the JPF infrastructure to define a custom interpreter where the standard JAVA Bytecode semantics are replaced with concolic semantics.

Observation 4: Test Suite Quality and Branch Coverage. Fig. 7 presents the quality metric for the generated test suites. Random testing, due to a very large number of generated test cases, has very low quality, often almost 0%. On the other hand, all the dynamic symbolic execution tools typically generate minimal test suites, i.e., those with 100% quality. SPF produces sub-optimal test suites in three cases, with as low as 15% quality for Raytracer. We were not able to find the reason for this unexpected behavior.

We give the usual branch coverage metric in Fig. 8. No tool reaches full branch coverage on the analyzed benchmarks, which is due to infeasible paths, exemplified by TCAS where JDART achieves full path coverage (i.e., #D/K is 0). In other words, JDART explores all possible behaviors of TCAS, and therefore 93% is the highest possible branch coverage, thus indicating the presence of code that cannot be reached from the entry point, i.e. dead code.

6 Conclusions

We presented JDART, a dynamic symbolic analysis framework for JAVA Bytecode programs. We provided a detailed description of its architecture and features, as well as an experimental evaluation of the tool in comparison to other similar frameworks. After several years of development, JDART has reached a level of efficiency, robustness, and versatility that lead to its recent open sourcing by the NASA Ames Research Center. This paper is therefore meant as an introduction of the tool to the research community. We hope that the tool's current capabilities and its existing use cases within other frameworks will inspire the community to experiment and extend it in novel ways.

References

1. ASM: A Java bytecode engineering library. `http://asm.ow2.org`
2. Balasubramanian, D., Păsăreanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.: Polyglot: Modeling and analysis for multiple statechart formalisms. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA). pp. 45–55 (2011)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 209–224 (2008)
4. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proceedings of the 19th International Workshop on Model Checking Software (SPIN). pp. 248–254 (2012)
5. Deters, M., Reynolds, A., King, T., Barrett, C.W., Tinelli, C.: A tour of CVC4: how it works, and how to use it. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD). p. 7 (2014)
6. Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Rakamarić, Z., Raman, V.: The Dart, the Psyco, and the Doop: Concolic execution in Java PathFinder and its applications. ACM SIGSOFT Software Engineering Notes 40(1), 1–5 (January 2015), proceedings of the 2014 Java Pathfinder Workshop (JPF)
7. Dimjašević, M., Rakamarić, Z.: JPF-Doop: Combining concolic and random testing for Java. In: The Java Pathfinder Workshop (JPF) (2013), extended abstract
8. Dinges, P., Agha, G.: Solving complex path conditions through heuristic search on induced polytopes. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 425–436 (2014)
9. Erzberger, H., Lauderdale, T.A., Chu, Y.C.: Automated conflict resolution, arrival management and weather avoidance for ATM. In: International Congress of the Aeronautical Sciences (2010)
10. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proceedings of the 23rd International Conference on Automated Deduction (CADE). pp. 208–214 (2013)
11. Gario, M., Micheli, A.: pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In: Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT) (2015)
12. Giannakopoulou, D., Howar, F., Isberner, M., Lauderdale, T., Rakamarić, Z., Raman, V.: Taming test inputs for separation assurance. In: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 373–384 (2014)
13. Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic learning of component interfaces. In: Proceedings of the 2012 International Static Analysis Symposium (SAS). pp. 248–264 (2012)
14. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 213–223 (2005)
15. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Queue 10(1), 20:20–20:27 (2012)
16. Howar, F., Giannakopoulou, D., Rakamarić, Z.: Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In: Proceedings of the 2013

International Symposium on Software Testing and Analysis (ISSTA). pp. 268–279 (2013)
17. JaCoCo Java code coverage library. http://www.eclemma.org/jacoco
18. Jayaraman, K., Harvison, D., Ganesh, V.: jFuzz: A concolic whitebox fuzzer for Java. In: Proceedings of the 1st NASA Formal Methods Symposium (NFM). pp. 121–125 (2009)
19. Java pathfinder. http://jpf.byu.edu
20. Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., Niemelä, I.: LCT: An open source concolic testing tool for Java programs. In: Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE). pp. 75–80 (2011)
21. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 553–568 (2003)
22. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008)
23. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering (ICSE). pp. 75–84 (2007)
24. Pasareanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA). pp. 34–44 (2011)
25. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA). pp. 15–26 (2008)
26. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV). pp. 419–423 (2006)
27. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE). pp. 263–272 (2005)
28. The SMT-LIB standard. http://smtlib.cs.uiowa.edu
29. Soot: A framework for analyzing and transforming Java and Android applications. http://sable.github.io/soot
30. Souza, M., Borges, M., d'Amorim, M., Păsăreanu, C.S.: CORAL: Solving complex constraints for symbolic Pathfinder. In: Proceedings of the 3rd NASA Formal Methods Symposium (NFM). pp. 359–374 (2011)
31. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: TesMa and CATG: Automated test generation tools for models of enterprise applications. In: Proceedings of the 37th International Conference on Software Engineering (ICSE). pp. 717–720 (2015)
32. Tillmann, N., Halleux, J.d.: Pex—white box test generation for .NET. In: Proceedings of the 2nd International Conference on Tests and Proofs (TAP). pp. 134–153 (2008)
33. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)