# Symbolic Execution for Generating Java Tests from Dafny Models

PI: Eric Mercer, Associate Professor, Computer Science, Brigham Young University
Cash funding needed: $53,420
AWS Promotional Credits needed: none
Amazon Contact: Sean McLaughlin, Aleks Charkarov, Matthias Schalaipfer, {seanmcl, aleksach, schlaipf}@amazon.com

**Abstract**

Dafny is a modeling and specification language with a static verifier to prove that a program implements its specification. Although it includes a compiler to `Java`, the resulting programs lack efficiency, readability, and any guarantee of preserving correctness. AWS is working on a specialized compiler so that Dafny can be used as a modeling language for web-services. This proposal is to generate a `JUnit` test suite from a Dafny model that gives object branch coverage (i.e., it includes short-circuit evaluation) and specification coverage (e.g., negative/positive tests for pre/post conditions). Such a test suite provides some assurance that the compiler preserves the static correctness properties of the Dafny model.

**Keywords –** *Contract programming, automatic test generation, symbolic execution*

## Introduction

*Contract programming* prescribes that designers write a precise interface specification for software components that can be used to reason about implementations. A *contract* is a declarative specification of what the software component computes and under what conditions that computation is correct. It defines required *pre-conditions* for a component's state and input at invocation, and it defines the guaranteed *post-conditions* of the component's state and outputs after invocation. Contract programming reduces defects and increases reliability by giving pause to the designer to think critically about, and state formally, the intent of each software component in context of the system and by providing early in the design process a robust foundation for automated reasoning.

*Dafny* is an expressive programming language for the static verification of programs. It is a sequential imperative object-oriented language with support for generics, inductive data-types, co-inductive data-types, and most importantly, constructs for formal specification. These constructs include contracts, *frames* to define read and write sets, loop-invariants, and ranking functions in the form of *decreases* clauses for termination. The Dafny verifier implements a deductive *weakest pre-condition* calculus to statically prove a Dafny program terminates and adheres to its specification.

The Dafny compiler targets a handful of backend languages, including `Java`, to implement a verified Dafny program. The compiler outputs source level code which can then be compiled to the intended framework. There is no guarantee that the meaning of the original Dafny model is preserved through compilation, and in general, the compiler does not produce readable, efficient, or idiomatic code. Such a final program is not suitable for deployment in any high assurance system. AWS is working on a specialized compiler from Dafny to `Java` for such systems so that Dafny can be used as a modeling language for web-services. The subject of this proposal is how to provide some further affirmation that the resulting compiled `Java` program still conforms to its original Dafny specification.

Proving that compilation exactly preserves the meaning of the original Dafny model, while an interesting and hard problem, is unlikely to be feasible in the near term [5, 10]. An acceptable alternative is to test to some predefined standard suitable for high assurance systems. This proposal is to automatically generate tests from a Dafny program sufficient to give *object branch coverage* and *specification coverage*. Object branch coverage is a common *white-box* criteria that means that the outcome of every branch, including outcomes from branches added by the short-circuit logic evaluation, is exercised by at least one test.

Specification coverage is a *black-box* criteria based on *input partitioning* where there is a negative test for each pre-condition and a positive test for each post-condition in the specification. The two coverage criteria provide some assurance that the compiled `Java` program from Dafny still implements its specification. [1]

It is worth noting that the value of Dafny's static verification is only as good the specification. It is easy to write weak contracts that say little about intent and are trivial to prove. It is also not hard to write strong contracts that are as complex as the actual code and hard to prove. Experience with engineers at Collins Aerospace and with undergraduates at Brigham Young University shows that it is not uncommon to leave out critical details that should be in a contract, believe a contract is saying something that it is not, or simply not realize a contract is vacuously true by virtue of how implications are used. Testing and debugging contracts is an interesting and hard problem.

A step toward testing contracts is to convert manually written `Java` tests to Dafny tests. Suppose there is an existing `Java` implementation of some service that is used as the basis for a Dafny model. The designer writes the specification for the Dafny model to statically prove each software component based on its `Java` counterpart. How does this designer know if the specification is strong enough to be useful? One answer is that the specification should be at least strong enough to discern negative and positive tests from the original `Java` implementation. If it is too weak to discern such tests, then it can be strengthened as needed. In this way, a designer is able to leverage the original `Java` tests in the Dafny model (see [11] for example).

### Related Work

Contracts can be higher-order or first-order depending on the language. Contracts are often used as programming annotations to improve readability of the code and enhance maintainability (e.g. [12, 16, 14]). Contracts form the basis of runtime monitors to check pre-conditions and assert post-conditions [13]. Monitors can be removed by proving an implementation adheres to its contract. Such proofs can be accomplished with static verification [28], model checking [15, 17, 9, 6, 7], or deductive reasoning [20, 18, 29, 1, 3]. Counter-examples from such proofs make good tests [4]. Contracts are equally useful for automated testing [27, 8, 25, 26]. A post-condition is a test oracle. The pre-condition is an input constraint for property based or fuzz testing. Mutation analysis on contracts generates useful tests to relate the contract to the code [22]. There are two well known fully verified compiler frameworks: CakeML [21, 19] and CompCert [23, 24]. Neither is suitable for a `Java`.

## Methods

Automatic test generation in this work is accomplished with either the Dafny verifier using its weakest pre-condition calculus or symbolic execution in Symbolic Java Pathfinder [2]. Both approaches rely on constraint solving with a backend *Satisfiability Modulus Theories* (SMT) solver. This presentation uses *symbolic execution* but it is not clear that one approach is more advantageous than another. Indeed, there is some obvious advantage with integrating test generation into the Dafny Verifier as it already provides much of the needed functionality.

Consider the Dafny program in Fig. 1(a). The program consists of a token class, `T`, and an ID station class, `I`. The token stores a fingerprint (`f`), an access clearance level (`l`), and a bit indicating if the token is valid (`v`). *Function-methods* are read only with the frame making clear what is read. These check a match between a passed in fingerprint to the stored fingerprint (`T.isF(f:int)`) and return the clearance level (`T.getL()`). The last method (`T.setV(f:int)`) side-effects as indicated by the modifies clause. It invalidates the token on a mismatched finger print.

[1] *MC/DC* and *boundary value analysis* are also possible [30].

```
class T {
  var f : int; var l : int;
  var v : bool;

  function method isF(f : int) : bool
  reads `f, `v {
    (v && (f == this.f))
  }

  function method getL() : int
  reads `l {
    l
  }

  method setV(f : int)
  ensures v == old(isF(f))
  modifies `v {
    v := isF(f);
  }
}
```

```
class I {
  var a : bool; var s : bool;
  var l : int;

  method o(t : T, f : int)
  requires !a && !s
  ensures
    s == (old(t.isF(f)) && l >= t.getL())
  ensures !(old(t.isF(f)) && l >= t.getL())
    ==> (a == t.isF(f))
  modifies t, `a, `s {
    if (t.isF(f) && l >= t.getL()) {
      s := true;
    } else {
      s := false;
      t.setV(f);
      a := t.isF(f);
    }
  }
}
```

Figure 1: An illustrative Dafny model for test generation.

```
assume (t.isF(f));
assume (l >= t.getL());
s := true;
```

(a)

```
assume (t.isF(f));
assume !(l >= t.getL());
s := false;
t.setV(f);
a := t.isF(f);
```

(b)

```
assume(!t.isF(f));
s := false;
t.setV(f);
a := t.isF(f);
```

(c)

Figure 2: Test paths for object branch coverage. (a) Both conditions true. (b) Second false. (c) First false.

The ID station class stores the alarm state (`a`), the state of a door, (`s`), and the maximum allowed clearance level to access the door (`l`). The method, `o(t : T, f : int)`, in the class arbitrates access to the door by checking a scanned fingerprint against the stored fingerprint in the token. The specification declares the behavior of the door and alarm. The pre-conditions (`requires`) require the alarm to be off and the door closed. The post-conditions (`ensures`) merit some discussion. The first ensures says the door only opens when the fingerprint matches the token and the token is within the clearance level. The second ensures is more subtle but intuitively the alarm sounds if the fingerprint does not match the token. Dafny is able to prove the code implements the specification with deductive reasoning.

The goal of this proposal is to generate tests for object branch coverage of `I.o(t : T, f : int)` with symbolic execution. Symbolic execution enumerates all the control flow paths in the method, and for each path, it generates a *path constraint* over the object state and primary inputs that must be satisfied to activate the path. Each path constraint is dispatched to a backend SMT solver to find the initial state values and primary inputs, if any, needed to activate the path. Paths are enumerated until there is a sufficient set to satisfy the coverage criteria.

There are three paths needed for object branch coverage for `I.o(t : T, f : int)`. For simplicity, these are shown as *straight line programs* in Fig. 2. A straight line program has no branching and uses the *assume* keyword to assert constraints that must hold for program execution to continue.

Dafny specifications provide important information for simplifying symbolic execution:

- Pre and post conditions are summaries that can be used to replace call sites.
- Frames specify dependencies useful for determining when call-sites need to be replaced.
- Loop invariants and ranking functions in decrease clauses guarantee termination.

Such required specifications in Dafny address the major hurdles to symbolic execution.

The straight line program in Fig. 2(a) is illustrative. Symbolic execution initializes the program state and input with *uninterpreted functions* (i.e., unconstrained variables that are able to assume any value) and executes the program to track the state and variable definition changes in terms of the uninterpreted functions. At the same time, it propagates the path constraint to capture the conditions that must be met by the state and variables to continue along the straight line program.

Symbolic execution begins by initializing a symbol table for variables in the current frame. This table is used to create the initial path constraint and to track changes to variables in the program path. The initial path constraint for Fig. 2(a) is $\neg a_0 \wedge \neg s_0$ to encapsulate the pre-condition in the specification where $a_0$ and $s_0$ are symbolic versions of the fields `a` and `s` in the symbol table. Novel in this proposal is the use of the Dafny frame in the specification for dependency analysis to determine that `t.isF(f)` and `t.getL()` are independent and not side-effecting; as such, these can be represented with uninterpreted functions, $t.isF_0$ and $t.getL_0$, in the symbolic execution. The two methods, with the represented symbolic variables, are added to the symbol table.

Symbolic execution proceeds by building up the path constraint as it encounters the assume statements. The new path constraint after the assume statements needs to express the object state that must exist before the assignment to field `s`: $\neg s_0 \wedge \neg a_0 \wedge t.isF_0 \wedge (l \geq t.getL_0)$. Symbolic execution then applies to the symbol table the assignment to `s` changing its entry to be the value `true`. That value would be used for `s` if the program continued.

The path constraint is passed to the backend SMT solver to generate inputs for the test: $a_0 = \text{false}$, $s_0 = \text{false}$, $l = 0$, $t.isF_0 = \text{true}$, and $t.getL_0 = 0$. The generated `JUnit` test uses a *mock* object for an instance of `T` for the test and directs the mock to return true whenever `T.isF(f)` is invoked and zero for whenever `T.getL()` is invoked.[2]

The post-conditions define the oracle for the test. What is interesting in this test is that the ensures statement in the specification makes reference to the *old* value of `t.isF(f)`. The old value is the value before the method call. The test must capture the value in a local variable before the call to the method under test in order to use that old value in the assertion. Dafny disallows any side-effecting calls in the specification so it is never the case that capturing the old value changes the object state. Fig. 3(a) is the final test expressed as a Dafny program. The assume statements fill the role of mocks. Although a simple example, it illustrates the utility of the frame in the Dafny specification to simplify the path constraint by abstracting out needless details through dependency analysis. This test program can be used to generate a `JUnit` test for the `Java` implementation.

Consider the slightly more complex path in Fig. 2(b). Here the dependency analysis identifies the relation between `t.setV(f)` and `t.isF(f)` with the former modifying a field that the latter reads. As before, `t.getL()` is independent. This proposal takes inspiration from SMT solvers by first solving the constraint problem with uninterpreted functions, and then, with dependency analysis, solve any secondary constraint problems. As such, the path constraint at the end of the method is $\neg s_0 \wedge \neg a_0 \wedge t.isF_0 \wedge \neg(l \geq t.getL_0)$ with a satisfying assignment of $a_0 = \text{false}$, $s_0 = \text{false}$, $l = 0$, $t.isF_0 = \text{true}$, and $t.getL_0 = 1$. The secondary constraint is solved using the specification for `t.setV(f)`: $t.v_0 \wedge (f_0 == t.f_0)$. These assignments are $t.v_0 = 0$, $t.f_0 = 0$, and $f_0 = 0$ for the final test in Fig. 3(b). The test asserts rather than assumes the call to `t.isF(f)` has the correct value.

Not discussed thus far are quantifiers in the specification. These may require more advanced theories in the SMT solver if they appear in the path constraint because of a pre-condition, or they can be flattened.

[2]Mockito is to be used for mocks.

```
method test_path0() {
  var i: I := new I;
  i.a := false; i.s := false;
  i.l := 0;
  var t: T := new T;

  var f: int;
  assume (t.isF(f));
  assume (t.getL() == 0);
  var old_isF := t.isF(f);
  i.o(t, f);
  assert (i.s ==
    (old_isF && i.l >= t.getL()));
  assert(!(old_isF && i.l >= t.getL())
         ==> (i.a == t.isF(f)));
}
```

(a)

```
method test_path1() {
  var i: I := new I;
  i.a := false; i.s := false;
  i.l := 0;
  var t: T := new T; t.v := true;
  t.f := 0;
  var f: int := 0;
  assert (t.isF(f));
  assume (t.getL() == 0);
  var old_isF := t.isF(f);
  i.o(t, f);
  assert(i.s ==
    (old_isF && i.l >= t.getL()));
  assert(!(old_isF && i.l >= t.getL())
         ==> (i.a == t.isF(f)));
}
```

(b)

Figure 3: Generated tests. (a) The true path. (b) A false path.

Quantifiers in the post-condition are flattened or translated into appropriate looping structures. Also, not discussed yet are looping structures, inductive types, and recursion. Loops in Dafny include loop invariants and decreases clauses for termination. These specifications should be sufficient for symbolic execution. Inductive types, recursion, and other advanced Dafny features are not considered in this proposal.

Specification coverage first generates negative tests to violate each of the pre-conditions using the back-end solver. In the running example, it would generate tests where `a` is set and `s` is not and visa versa. It then checks the current test suite to see if tests exists to exercise each way that a post-condition could be satisfied. In the example, at least two tests must exist to exercise each side of the implication defining the behavior of `a` but tests are also required for each value allowed by the equivalence checks (e.g. == in the ensures). Symbolic execution is used to search for paths, if any, that exercise uncovered conditions by including the condition in final path constraint passed to the backend solver. In the running example, the eventual test from the path in Fig. 2(c) covers the missing value of the equivalence check for `a`.

## Expected Results

A tool that generates a `JUnit` test-suite for object branch and specification coverage from a verified Dafny program is the expected outcome with appropriate publications. The first quarter is to generate tests from straight line programs tackling the dependency analysis and symbolic execution. The second quarter addresses loops, and the use of loop-invariants, in the test generation. The third quarter implements path enumeration and specification coverage. The fourth quarter targets case studies and publications.

## Funds Needed

The $53,420 in requested funds provide direct support for two graduate students. The amount is based on both the cost of wages and tuition. Base wage is $20.00 per hour for 20 hours per week for 16 weeks for the fall and winter semesters, and it allocates 40 hours per week for the spring/summer semester for 16 weeks. That is 1,920 total hours for a total wage cost of $38,400. Full-time graduate student tuition for 2020-2021 is $7,510 per student for a total tuition cost of $15,020.

# Appendices

## A References


[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018. Springer International Publishing.

[4] Jürg Billeter. Counterexample Execution. Master's thesis, ETH Zurich, Germany, 2008.

[5] CakeML. `https://cakeml.org/`.

[6] Paul Caspi, Christine Mazuet, Rym Salem, and Daniel Weber. Formal design of distributed control systems with lustre. In Massimo Felici and Karama Kanoun, editors, *Computer Safety, Reliability and Security*, pages 396–409, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[7] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.

[8] Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *In Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 545–557, 2005.

[9] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 126–140, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] CompCert. `https://compcert.org/`.

[11] Dafny-Java Test Generation. `https://github.com/ericmercer/dafny-java-test-generation`.

[12] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). *SIGPLAN Not.*, 51(9):117131, September 2016.

[13] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 214–233, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. *SIGPLAN Not.*, 46(9):176188, September 2011.

[15] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In Eran Yahav, editor, *Static Analysis*, pages 351–368, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):4859, September 2002.

[17] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The jkind model checker. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 20–27, Cham, 2018. Springer International Publishing.

[18] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. *Formal Specification with the Java Modeling Language*, pages 193–241. Springer International Publishing, Cham, 2016.

[19] Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 999–1026. Springer, 2018.

[20] Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering*, pages 284–299, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[21] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: Sel4: Formally verifying a high-performance microkernel. *SIGPLAN Not.*, 44(9):9196, August 2009.

[22] Willibald Krenn and Bernhard K. Aichernig. Test case generation by contract mutation in Spec#. *Electronic Notes in Theoretical Computer Science*, 253(2):71–86, 2009. Proceedings of Fifth Workshop on Model Based Testing (MBT 2009).

[23] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[24] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[25] Marcus Lindner, Jorge Aparicius, and Per Lindgren. No panic! verification of rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 108–114, 2018.

[26] Marcus Lindner, Nils Fitinghoff, Johan Eriksson, and Per Lindgren. Verification of safety functions implemented in rust - a symbolic execution based approach. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 432–439, 2019.

[27] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and František Plášil, editors, *SOFSEM 2007: Theory and Practice of Computer Science*, pages 114–129, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[28] Phúc C. Nguyundefinedn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[29] K. Rustan and M. Leino. Developing verified programs with dafny. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, pages 82–82, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[30] Michael W. Whalen, Suzette Person, Neha Rungta, Matt Staats, and Daniela Grijincu. A flexible and non-intrusive approach for computing complex structural coverage metrics. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 506–516, 2015.

# Eric G Mercer

## CONTACT INFORMATION


Computer Science
Brigham Young University
3361 TMCB
Provo, UT 84602-6576 USA

Phone: +1 801 422 4628
Fax: +1 801 422 0169
eric.mercer@byu.edu
http://vv.cs.byu.edu


### Education degree, discipline, institution, year

**Ph.D.** in Electrical and Computer Engineering. The University of Utah, December 2002.

**M.S.** in Electrical Engineering. The University of Utah, June 1999.

**B.S.** in Computer Engineering. The University of Utah, December 1996.

### Academic experience

**Associate Professor**, 08/2008—present.
Computer Science, Brigham Young University.

**Assistant Professor**, 08/2002—08/2008.
Computer Science, Brigham Young University.

### Publications

- Y. Huang, B. Ogles, and E, Mercer, "A Predictive Analysis for Detecting Deadlock in MPI Programs", *Proceedings of Automated Software Engineering* (ASE), September 2020, ACM.
- Y. Huang, K. Gong, and E. Mercer, An efficient algorithm for match pair approximation in message passing, Journal of Parallel Computing, Volume 91, March 2020, pp. 102585.
- B. Ogles, E. Mercer, P. Aldous, "Proving Data Race Freedom in Task Parallel Programs with a Weaker Partial Order", *Proceedings of Formal Methods in Computer Aided Design* (FMCAD), 22-25 Oct 2019, IEEE.
- R. Nakade, E. Mercer, P. Aldous, K. Storey, B. Ogles, J. Hooker, S. J. Powell, and J. McCarthy, Model-checking task-parallel programs for data-race Innovations in Systems and Software Engineering, Volume 15(3), pp. 289-306, 2019, 10.1007/s11334-019-00343-5, Errata.
- R. Nakade, E. Mercer, P. Aldous, J. McCarthy, "Model-Checking Task Parallel Programs for Data-Race", In: Dutle A., Muoz C., Narkawicz A. (eds) NASA Formal Methods, NFM 2018, Lecture Notes in Computer Science, vol 10811. Springer, Cham.
- B. Hillery, E. Mercer, N. Rungta, and S. Person, "Exact Heap Summaries for Symbolic Execution", in *Proceedings of Verification, Model Checking, and Abstract Interpretation* (VMCAI 2016), Lecture Notes in Computer Science, Volume 9583, pp 206-225, 2016.
- B. Hillery, E. Mercer, N. Rungta and S. Person. "Towards a Lazier Symbolic PathFinder", in *Proceedings of JPF Workshop*, SIGSOFT Software Engineering Notes, Volume 39, Feb 2014. `http://ti.arc.nasa.gov/events/jpf-workshop-2013/`.

**Professional Development**: Collins Aerospace Consultant (present), NASA Visiting Scientist, 2013—2015

# C Previously Funded Project Summary

Nothing to report.