Expanding the Capabilities of Automated Program Verifiers for Testing, Synthesis, and Concurrency

A dissertation

submitted by

Aleksandr Fedchin,

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

February 2026

ADVISOR: Jeffrey S. Foster

Expanding the Capabilities of Automated Program Verifiers for Testing, Synthesis, and Concurrency

Aleksandr Fedchin

ADVISOR: Jeffrey S. Foster

Program verification is a powerful technique that can provide strong guarantees against runtime bugs, yet most mainstream programming languages do not support verification as a feature. This is arguably because implementing such a feature is technically challenging, while the number of developers who would use it is comparatively small. Moreover, even specialized programming languages that do allow verification, such as Dafny, often lack support for common programming paradigms such as concurrency. These limitations may hinder the wider adoption of formal verification in the programming language community.

This dissertation aims to show that program verification has additional applications that may broaden the technique's appeal and justify its integration in existing programming languages. The dissertation focuses on Dafny, a programming language that already allows automated verification by discharging proof obligations to an SMT solver. I show that Dafny can repurpose this functionality to enable additional capabilities, embodied in a test generation framework called DTest, a program synthesis tool called Metamorph, and a library for MPI concurrency called DafnyMPI. DTest uses the verifier to automatically generate test inputs, leading to test suites that

achieve up to 79% statement coverage on industrial benchmarks. Metamorph compliments DTest by allowing generation of complex heap-based values from user specifications by synthesizing sequences of appropriate method calls. DafnyMPI allows reasoning about concurrent message passing code, including proving deadlock freedom, functional correctness, and safe use of read and write buffers. While these extensions are specific to Dafny, I argue that the ideas behind them should be applicable to other programming languages with support for automated verification. Together, these results suggest that automated verification should be seen as a crucial component of the language ecosystem rather than merely an end in itself.

Acknowledgments

I am very fortunate to have had many wonderful people help me get to this point.

I want to thank **Jeff**, who has gone above and beyond what I could have hoped for in an advisor, whose consistent and always helpful feedback and genuine belief in the value of research encouraged me to keep going, and whose work ethic will always set an example for me.

I want to thank **Zvonimir**, who helped me discover what I now love about automated reasoning, allowed me to explore many different research ideas while at Amazon, and supported me in ways that went far beyond any formal role.

As with most dissertations, this text will have few readers, but I am very grateful and could not be happier that **Hari**, **Lenore**, and **Mark** are amongst them.

I would like to thank people who taught and guided me before Tufts: **Vitaly**, for helping me with my high-school projects and much more, **Sven**, who taught me more about Computer Science than any other person, **Rob**, who taught me how to write, **Olga** for helping me throughout my time at Bard and ever since, **Pramit**, **Joseph**, and **Patrick** for guiding me in my first ever publication.

I want to thank all of my co-authors, particularly **Alex** and **Antero**, for often catching issues I missed and offering ways to fix them. Thank you also to all the fellow TuPL PhD students and postdocs for forming a wonderful community and for all the feedback on the presentations

I gave at the reading group over the years: thank you **Milod**, **Jared**, **JM**, **Sam**, **Sankha**, **Nate**, **Fox**, **Mark**, **Dinghong** and **Vadim**.

Throughout my five years at Tufts, I have continually been supported by CS department faculty and staff, who helped me navigate academia, teaching, program requirements, and, most importantly, helped me with every single issue I faced without exception. Thank you **Diane**, **Karen**, **Marty**, **Richard**, **Angelica**, **Ellen**, **Jenny**, **Johnny**, **Sandie**, and **Sarah**.

Much of this dissertation is inspired by the work I did at Amazon, and I would like to thank all the wonderful people whom I met there. Thank you **Aaron**, **Alex**, **Anjali**, **Cody**, **Fabio**, **Giles**, **John**, **Lucas**, **Mikael**, **Neha**, **Remy**, **Robin**, **Rustan**, **Sean**, and **Stefan**.

This dissertation has been partly written in Bishkek, Kyrgyzstan. I want to thank the wonderful AUCA community and particularly **Jonathan** and **Almaz** for inviting me to teach there.

My parents have helped me every step of the way and in every possible way. Thank you **mom** and **dad**!

And thank you **Izzy**, for always believing in me, and for crossing twelve time zones to be with me when I most needed you.

Sasha

Contents

Abstract ii

Acknowledgments iv

List of Tables x

List of Figures xi

Chapter 1 Introduction 1

1.1 Dafny: A Brief Introduction . . . 3

1.2 DTest: Automated Test Generation for Dafny . . . 5

1.3 Metamorph: Program Synthesis for Dafny . . . 5

1.4 DafnyMPI: Concurrency for Dafny . . . 6

1.5 Broader Applicability and Future Work . . . 7

1.6 Attribution and Acknowledgement . . . 7

Chapter 2 A Toolkit for Automated Testing of Dafny 8

2.1 Toolkit Overview . . . 11

2.2 Unit Testing and Mocking Frameworks . . . 14

2.3 Automated Test Generation . . . 16

2.3.1 Custom Dafny to Boogie Translation . . . 16

2.3.2 Trap Assertion Injection . . . 18

2.3.3 Unit Test Generation . . . 18
2.3.4 Limitations . . . 20
2.4 Empirical Evaluation . . . 21
2.4.1 Unit Testing and Coverage . . . 21
2.4.2 Test Suite Size . . . 24
2.5 Related Work . . . 26
2.6 Conclusion . . . 27

Chapter 3 Metamorph: Synthesizing Large Objects from Dafny Specifications 28

3.1 Motivating Example . . . 32
3.2 Synthesis Algorithm . . . 37
3.2.1 Overview . . . 37
3.3 Guiding the Search . . . 42
3.3.1 State Representation . . . 42
3.3.2 Greedy Distance Metric and Best-First Search . . . 44
3.3.3 Piecewise Distance Metric and A* Search . . . 44
3.4 Implementation . . . 49
3.4.1 Solver Timeouts . . . 49
3.4.2 Recursive Object Synthesis . . . 50
3.5 Evaluation . . . 50
3.5.1 Benchmarks . . . 51
3.5.2 Correctness . . . 53
3.5.3 Baseline . . . 53
3.5.4 Synthesizing Large Objects . . . 54
3.5.5 Comparison of Distance Metrics . . . 57
3.5.6 Integration with DTest . . . 60
3.5.7 Synthesizing Virtual Machine Bytecode . . . 61
3.6 Discussion and Future Work . . . 63

3.7 Related Work . . . 65
3.8 Conclusion . . . 66

Chapter 4 DafnyMPI: A Dafny Library for Verifying Message-Passing Concurrent Programs **68**

4.1 Overview . . . 70
4.1.1 Solving PDEs with the Message Passing Interface . . . 71
4.1.2 DafnyMPI . . . 73
4.2 Proving Deadlock Freedom . . . 77
4.2.1 The Core Calculus . . . 79
4.2.2 Operational Semantics . . . 82
4.2.3 Axioms . . . 86
4.2.4 Deadlock . . . 88
4.2.5 The Deadlock Freedom Proof . . . 90
4.2.6 Rely-Guarantee Reasoning for Message Contents . . . 95
4.3 Implementation . . . 95
4.4 Evaluation . . . 100
4.4.1 Benchmarks . . . 100
4.4.2 Proof Engineering Effort . . . 102
4.4.3 Runtime Performance . . . 103
4.5 Discussion and Future Work . . . 104
4.6 Related work . . . 105
4.7 Conclusion . . . 107

Chapter 5 Conclusion and Future Work **109**

5.1 Broader Applicability . . . 110
5.2 Future Work . . . 112
5.2.1 DTest and Metamorph . . . 112

5.2.2 DafnyMPI 113
5.2.3 Counterexample-Centric Verification 113

Bibliography **115**

List of Tables

2.1 Overview of achieved coverage. 23

2.2 Comparison of minimization strategies. 25

3.1 Running time of Metamorph on problems from the Virtual Machine benchmark (median of 3 runs). 62

4.1 Axioms of the core calculus (implicitly parameterized by program state, except for AtStart). 87

4.2 Definition of deadlock an related terms. 88

4.3 Number of lines of code (LOC) and verification time as measures of proof complexity. 103

List of Figures

1.1 An example Dafny program. 3

2.1 Toolflow of the Dafny testing toolkit. 10

2.2 Unit test generation example. 12

2.3 An example use of the :synthesize attribute for mocking (simplified). 15

2.4 Runtime of DTest on methods from ESDK and DUTIL. 22

3.1 A partial API specification written in Dafny. 32

3.2 The synthesis problem (a) and solution (b) for the API in Figure 3.1. 34

3.3 Object states Metamorph explores while solving the problem in Figure 3.2. Boxes corresponds to states and edges between states are annotated with method calls that transform one state into another. Solid boxes are states that lie along the path to the solution. Each state is illustrated with the corresponding social network graph, as well as with the piecewise distance from the initial state as computed by Metamorph. 36

3.4 The synthesis algorithm used by Metamorph. 38

3.5 Example queries that Metamorph makes as part of the PreconditionFor (a) and GeneralizePrecondition (c) subroutines. Dafny verifies (c) but returns a counterexample (b) when asked to verify (a). GeneralizePrecondition proves that the constraint on line 17 is redundant because (c) verifies with line 25 commented out. 40

3.6 The process of computing piecewise distance for the target state in Figure 3.2a. . 45
3.7 Queries Metamorph sends to Dafny when pre-training on the Social Network API in Figure 3.1. Dafny verifies (b) but cannot verify (a), so Metamorph learns that a call to AddUser can affect no more than one constraint of type u in users at a time. 46
3.8 Running time (log scale) of Metamorph, its modifications, and a baseline approach on six benchmarks. 54
3.9 Breakdown of Metamorph's running time. 57
3.10 Running time (log scale) of Metamorph under different distance metrics. 59
3.11 Two equivalent solutions synthesized when using the piecewise distance (a) and greedy distance (b) metrics. 59

4.1 Example Python MPI program (left) and corresponding Dafny proof obligations (right, simplified) 72
4.2 Core calculus syntax. 79
4.3 Core calculus reduction rules. 83
4.4 Two partial reductions for the following program, executed by 2 processes: **set** x 5**; if** rank $=$ 0 **{ isend** t x **} { irecv** t x **}; wait** t. Edges are labeled with abbreviations of reduction rules (SS sands for SeqSkip, etc.). The reduction rule above the line is taken first. 84
4.5 Benchmark PDE simulations. Top row: initial conditions; bottom row: solutions after 100 time steps as computed by DafnyMPI. From left to right: Linear Convection, Poisson, Heat Diffusion 99
4.6 Structure of MPI Communication for the Linear Convection (top left), Poisson (top right, transposed), and Heat Diffusion (bottom, transposed) benchmarks. . . 100
4.7 Running time as function of the process count. From left to right: Linear Convection, Poisson, Heat Diffusion. 103

Chapter 1

Introduction

The modern world increasingly relies on software to automate ever more complex parts of everyday infrastructure: cloud services, delivery logistic chains, etc. Faults in such software can have dramatic consequences. Traditionally, developers use testing to catch such faults before the software is deployed in production. However, testing has a major limitation in that it only provides certainty about the program's behavior for a small subset of possible inputs. This dissertation argues for wider adoption of software verification, an alternative to testing that provides stronger correctness guarantees. To that end, the dissertation presents several new ways in which program verification can be made more accessible to software developers.

The key advantages of verification is that it allows proving properties of a program's runtime behavior prior to execution and for an arbitrarily large range of inputs. Success stories range from verified compilers [76] to optimized cloud authorization engines running in production [23]. Compared with testing, verification provides much stronger correctness guarantees. For example, one study of C compilers found hundreds of bugs in LLVM and GCC but only two bugs in the verified CompCert compiler [118]. The ever-changing nature of software restricts the set of problems where verification is worthwhile [33], but the above examples demonstrate that this level of assurance is indeed helpful in such domains as compilers and authorization systems, where small mistakes can have significant consequences.

However, despite these successes, and even within the specialized domains mentioned above, program verification remains a relatively niche field of Computer Science. In practice, the difficulty of adoption stems, at least in part, from two seemingly irreconcilable problems. On the one hand, using verification tools is technically challenging. This means that software engineers may be unwilling to use specialized verification software unless doing so offers an immediate benefit; verification is sometimes contrasted with testing in this respect [99]. On the other hand, perhaps to reduce the difficulty of use, verification tools often restrict the set of programming paradigms they support, with concurrency in particular often not being supported by default. Without concurrency, however, a verification tool risks undermining the promise of enabling safe optimization, which is one of verification's main selling points (as in [23]). This has driven the development of concurrent verification systems such as Chalice [74] and VCC [28]. However, by supporting concurrency, these tools become more specialized and inherently harder to use than their sequential counterparts, leading back to the first problem. To get the best of both worlds, one would want a tool that (i) offers additional benefits that justify adopting verification, and (ii) supports concurrency when needed without complicating sequential verification.

This dissertation aims to achieve both things by extending Dafny, a programming language that comes with a built-in automated verifier [72, 73]. We focus on Dafny because it is one of the more widely used verification languages [23, 22, 20] yet, prior to this work, it lacked concurrency support and test generation infrastructure. The dissertation aims to show that Dafny's automated verification pipeline can be repurposed to enable test generation, message-passing concurrency, and program synthesis. These additional capabilities, which should make Dafny more appealing to software engineers, are primarily possible because of Dafny's SMT-based approach to verification, where low-level proof details are automatically filled in by the verifier. For a detailed introduction to Dafny, see Section 1.1. The ideas presented in this dissertation should transfer to other SMT-based verification systems similar to Dafny (e.g., Viper [89], Frama-C [30]).

In summary, this dissertation aims to demonstrate that:

> *Programming languages that support automated program verification can offer additional capabilities that rely on the pre-existing verification pipeline. First, the verifier can be reused as a backend for test generation and program synthesis. Second, verification of select concurrent programs may be possible without the need to introduce concurrency-specific logics. Automated verification should therefore be seen as a crucial part of the language ecosystem rather than an end in and of itself.*

The remainder of this chapter introduces Dafny and the key ideas behind each of the extensions to the language mentioned above.

1.1 Dafny: A Brief Introduction

Dafny is an imperative and functional programming language that also supports object-oriented constructs. Dafny programs can be compiled to Java, C#, Python, Go, or JavaScript for execution. Figure 1.1 shows an example Dafny method, called Reciprocal, that takes a real number as input and returns its reciprocal as output. In most programming languages, the user has to be careful to never call such a method with zero as its argument to avoid a division-by-zero error at runtime. In contrast, the Dafny verifier immediately flags this as an issue and says that a division by zero is possible. To make verification go through, the user must write a *precondition* (the requires clause on line 3) that promises that the argument is never zero. This precondition is then checked by Dafny on every call to the method. The user can also annotate methods with *postconditions* (ensures clauses), which are properties that Dafny checks to always hold at the end of the method execution. The postcondition on line 4 ensures that the

```
1 method Reciprocal(n:real)
2   returns (r:real)
3   requires n ≠ 0.0
4   ensures n > 1.0 ⟹ r < n
5 { r := 1.0 / n; }
```

Figure 1.1: An example Dafny program.

output of the method is less than its input, provided that the input is greater than 1.0. In this case, Dafny automatically proves this property, without requiring the user to explicitly call a lemma about reciprocals as may be necessary when using proof assistants. Instead, Dafny performs verification by translating the program to Boogie, a lower-level verification language that, in turn, invokes an SMT solver to check that the program is correct. In this case, the SMT solver is able to prove the postcondition on the method automatically. Dafny can then assume this postcondition to be true any time the method is called.

The purpose of pre- and postconditions is to allow modularity: Dafny verifies each method one at a time and does not expose the body of one method to another—all properties necessary for verification must be encoded in the method's pre- and postconditions, which can also be referred to together as the method's *specification*. Aside from writing specifications, the programmer may also aid the verifier by adding assertions and loop invariants. To capture complex properties about a program's behavior, the user may also define *functions* and call out to them from within the specification. Unlike methods, all Dafny functions are pure: they always return the same output for the same input, and they never have side-effects. A Dafny programmer would often write executable code in imperative fashion using methods and then write a functional specification to be used for verification, e.g., to prove that one method always returns the same value as another.

In principle, a Dafny program can never lead to a runtime error and will always terminate—Dafny requires the user to prove that every loop or recursive function decreases some value that is bound below by zero. In practice, however, a Dafny program may call out to unverified libraries via its foreign function interface, and errors could also hypothetically be introduced by the compilers that translate Dafny to Java or another language. The possibility of such errors motivates the work on automated testing described in the next section.

1.2 DTest: Automated Test Generation for Dafny

Chapter 2 introduces DTest, a tool for automated generation of tests for Dafny programs. The motivation is twofold. On the one hand, as mentioned above, the promise of automated testing may help convince developers to switch to a verification-aware system. Testing can be performed before a program is fully verified, so the programmer may gain an initial assurance of their code's correctness before proceeding to the costly step of writing a formal proof. On the other hand, DTest may help detect bugs introduced by unverified components of the Dafny pipeline, including its compilers and linked external libraries.

DTest works by modifying the way Dafny code is translated to the Boogie intermediate verification language [8]. These changes enable DTest to query the verifier for counterexamples that drive execution towards specific statements or branches in the code. We evaluate DTest on two real-world Dafny codebases: the Dafny utilities library [77] and the AWS Encryption SDK [5], achieving up to 79% statement and 84% branch coverage. Prior work such as Delfy [103, 26] explored concolic testing for Dafny but was limited in scope and does not support the modern version of the language. In summary, DTest shows that a verification-aware language can provide automated testing support, lowering the adoption barrier for software engineers.

1.3 Metamorph: Program Synthesis for Dafny

One of the major limitations of DTest is its inability to produce complex heap-based test inputs. DTest can fully specify an object that it needs to generate but it does not know which Dafny methods to call to construct such an object. In Chapter 2, we partially mitigate this problem by introducing DMock, a mocking framework that allows creating mock objects at runtime. However, DMock is specific to a compilation backend (C#) and is constrained by what runtime mocking makes possible—the objects it creates are essentially static. These limitations motivate the creation of a synthesis tool that can take a specification of an object as input and generate a sequence of method calls that produces such an object.

To that end, Chapter 3 introduces Metamorph, a tool for the synthesis of method call sequences that construct an object according to a user-provided specification. Metamorph relies on the Dafny verifier to infer method arguments that transform an object from one state to another. This functionality then allows Metamorph to perform goal-directed search over possible object states. Metamorph incorporates several distance metrics to guide the synthesis process. Depending on the distance metric used, the algorithm uses breadth-first, best-first, or A* search. We evaluate Metamorph in several dimensions, including by integrating it with DTest and by generating programs that require up to 57 method calls. The fact that Metamorph can synthesize such long sequences of method calls sets it apart from prior work, which typically limits the number of method calls to, e.g., five [84] or eight [120]. Overall, we believe that Metamorph demonstrates that automated verifiers are well-positioned to empower program synthesis.

1.4 DafnyMPI: Concurrency for Dafny

The correctness of parallel scientific software is critical, yet there are few practical formal verification techniques for such systems. Model checking and dynamic verification approaches [110, 109, 93, 112] can suffer from the state explosion problem and typically require assuming a fixed number of processes (see, for example, the evaluation of the CIVL-C model checker on MPI programs [79]). Proof assistants, e.g., those using concurrent separation logic [64, 54], can perform arbitrarily complex reasoning, but verification of even small programs requires substantial effort by a team of specialists. Automated verifiers with concurrency support (e.g., Chalice [74]) focus primarily on low-level primitives such as locks and thread fork/join, which can make it difficult to model more complex communication patterns. One goal of this dissertation is, therefore, to show how to verify concurrent programs in Dafny without complicating the use of the language.

Chapter 4 introduces DafnyMPI, a Dafny library for message passing concurrency. DafnyMPI consists of a number of Dafny methods that can be used for both point-to-point and collective communication between processes. The user of DafnyMPI must specify the topology of the com-

munication upfront; the preconditions on the library methods enforce an ordering property on message tags to guarantee absence of deadlocks and allow reasoning about return values of each method call. At compile time, the calls to DafnyMPI methods are translated to the corresponding method calls in the mpi4py library for Python. We demonstrate the use of DafnyMPI by implementing sequential and parallel versions of three common partial differential equation solvers, verifying that the two implementations are functionally equivalent, and testing them at runtime. In summary, DafnyMPI presents a new approach to modelling concurrency within the constraints of a sequential program verifier.

1.5 Broader Applicability and Future Work

While this dissertation focuses on Dafny, many of the underlying ideas could be applied in the context of other verification-aware languages and tools. Chapter 5 discusses this in detail, focusing in particular on Viper [89], which shares many similarities with Dafny, and Frama-C [30], which targets low-level C code but where similar considerations still apply. Chapter 5 also outlines possible directions for future work and summarizes the contributions of this dissertation.

1.6 Attribution and Acknowledgement

Portions of this dissertation are based on joint work, some of which has been previously published. For consistency, I will be using first person plural throughout this text. The material in Chapter 2 appeared in the 2023 Proceedings of the 15th International Symposium on NASA Formal Methods (NFM), co-authored with Tyler Dean, Jeffrey S. Foster, Eric Mercer, Zvonimir Rakamarić, Giles Reger, Neha Rungta, Robin Salkeld, Lucas Wagner, and Cassidy Waldrip [37]. The material in Chapter 3 appeared in the 2025 Proceedings of the ACM on Programming Languages, OOPSLA issue, co-authored with Alexander Y. Bai and Jeffrey S. Foster [36]. The material in Chapter 4 appeared in the 2026 Proceedings of the ACM on Programming Languages, POPL issue, co-authored with Antero Mejr, Hari Sundar, and Jeffrey S. Foster [38].

Chapter 2

A Toolkit for Automated Testing of Dafny

It is common for a program written in a verification-ready language, such as Dafny, to first be compiled into a traditional programming language, such as C#, before being deployed to production. This way one can leverage the extensive compiler optimizations and libraries that have already been developed for popular programming languages. At the same time, one also needs to guarantee the correctness of the final deployed program. First, it is necessary to ensure that the Dafny compilers, such as the Dafny to C# compiler, do not introduce unexpected behavior [59]. One approach to ensure the correctness of a compiler would be to verify it end-to-end. There have been such efforts in the past for other languages [76]. While successful, these efforts took years of manual human effort, and Dafny supports compilation to several different languages making verification of the entire toolchain very difficult. Second, many Dafny programs use external libraries, which are another potential source of bugs since they are not written in Dafny and hence are not verified to match their specifications. Incorrect specification of an external library may introduce bugs even if the library itself and the entire compilation pipeline are verified to be correct.

In this chapter, we propose to increase assurance of the correctness of the compiled Dafny program by leveraging automated testing. More specifically, we introduce a toolkit for automated testing of Dafny programs, consisting of DUnit (unit testing framework), DMock (mocking frame-

work), and DTest (automated test generation). The main purpose of the combined toolkit is to ensure that the guarantees provided by verified Dafny programs are preserved when those programs are executed via compilation to a different programming language, such as C#. The main component of the toolkit is DTest, a tool for automated generation of tests that achieve high coverage of Dafny programs. The tests themselves are written in Dafny and compiled to use testing frameworks in selected target languages, including C#. The tests assert that method postconditions verified in Dafny hold at runtime. Thus, we can use DTest to (i) generate tests to ensure a compiled program preserves the behavior verified in Dafny; (ii) increase confidence in specifications of external libraries that cannot be verified; and (iii) increase assurance that a Dafny program is functionally equivalent to an existing implementation that may be written in another language.

To compile tests to the target programming language, we introduce DUnit and DMock, unit testing and mocking frameworks for Dafny. DUnit extends Dafny with a method attribute :test, which signals the compiler to mark the corresponding method as a unit test in the testing framework of the target language. To support DUnit, DMock facilitates generation of complex heap structures as test inputs by adding mocking capabilities to Dafny. We introduce a new Dafny attribute (:synthesize) for tagging of mock methods, which have no body in Dafny but instead describe their return values with postconditions. Tests produced by DTest rely on mock methods to bypass the need to infer how to use existing constructors to create objects with specific field values. Instead, DMock automatically compiles mock methods to code (using the popular Moq mocking framework for C#) that returns objects that comply with the corresponding postconditions. Currently, DMock can produce mock implementations for a specific but broadly useful set of postconditions: one can supply concrete values for constant instance fields or redefine the behavior of instance functions.

Fig. 2.1 shows the typical toolflow of the Dafny testing toolkit. DTest is implemented as an extension to Dafny and uses the existing Dafny verifier, which works by translating the Dafny program to the Boogie intermediate verification language [8, 16]. Boogie, in turn, proves each as-

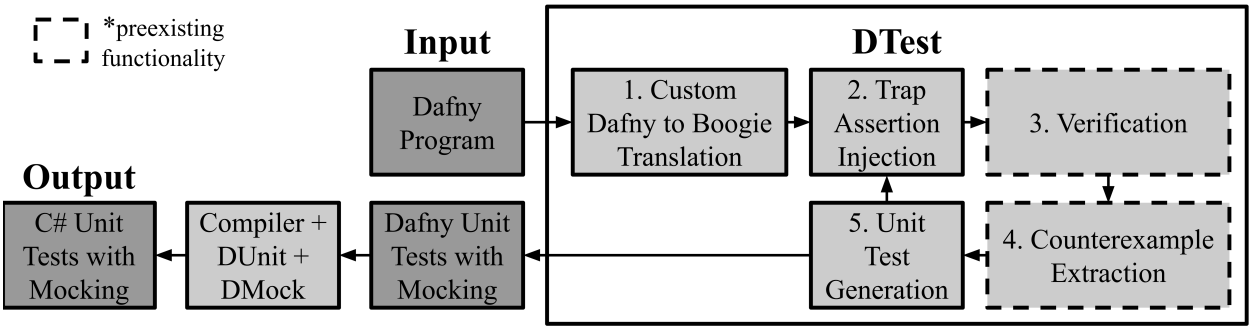


Figure 2.1: Toolflow of the Dafny testing toolkit.

sertion with Z3 [88, 119]. DTest starts test generation by translating Dafny to Boogie (step 1 in the figure), including several changes to the existing translation pipeline (see Sec. 2.3.1). Next, DTest enters a loop where it systematically injects trivially failing *trap assertions* (meaning assert false) into the Boogie code and uses the Boogie verifier and counterexample extractor [24] to generate counterexamples that reach the assertions (steps 2–4). Then, DTest translates counterexamples into Dafny tests (step 5) using unit testing and mocking attributes understood by DUnit and DMock, and converts method postconditions into runtime oracles (see Sec. 2.3.3). We then compile the Dafny program and the generated tests to C# using the Dafny compiler augmented with the functionality that DUnit and DMock provide.

We evaluated our toolkit across two dimensions. First, we used DTest to generate unit tests for the Dafny utilities library (DUTIL) [77] and the portion of the AWS Encryption SDK (ESDK) that is implemented in Dafny [5]. We then compiled each library and its tests to C# and measured the coverage of the tests on the C# code: the tests produced by DTest achieved 79% (resp. 62%) statement and 84% (resp. 58%) branch coverage on DUTIL (resp. ESDK). This is promising since the ESDK developers target 80% statement and 35% branch coverage for their manually written unit tests as part of their wider testing strategy. Second, we compared the number of tests DTest generates to achieve full coverage to the number of tests generated by a brute-force algorithm that can optimally minimize the number of tests. We found that DTest often generates close-to-the-minimal number of tests, with the worst observed case (for some of the methods with the most complex control flow) being three times the optimal.

In summary, the main contributions of this chapter are as follows:

- We introduce DTest, a tool that uses the Dafny verifier to automatically generate unit tests for preexisting Dafny programs.
- We develop DUnit and DMock, unit testing and mocking frameworks for Dafny that support automated compilation of tests and construction of objects based on a formal description of their behavior.
- We evaluate the toolkit on a set of real-world Dafny programs and show that the generated tests achieve coverage expected by the developers.
- We released our toolkit with Dafny [31] and made the persistent artifact available at https://doi.org/10.5281/zenodo.7310719.

Overall, our results show that DUnit, DMock, and DTest are a promising toolkit for automatically generating high coverage tests for Dafny. More broadly, our work should be useful to researchers and practitioners working in verification-ready ecosystems other than Dafny, as we provide solutions for critical pain points in test generation, including dealing with pre- and postconditions, mocking in their presence, and leveraging the verifier for automatic test generation.

2.1 Toolkit Overview

Fig. 2.2a gives the example Dafny method LexLeq (LexicographicByteSeqBelowAux originally) we extracted from the ESDK to illustrate how DUnit and DTest work. It takes byte sequences x and y and an index n as input, and returns a Boolean indicating whether x is equal to or precedes y in lexicographic order starting at position n. The core logic of the method (lines 5–7) is a disjunction of conditions that would make this true: either we have reached the end of x, or the byte at position n in x comes before y, or the two bytes are equal and x is before y lexicographically at position n+1. Otherwise, x is greater than y at n. Because the method is recursive, it is accompanied by a decreases clause (line 4), which allows Dafny to prove termination by stating that at each recursive call the value of |x| – n decreases. The method also has a precondition (line 2) requiring that n is

```
function LexLeq (x: seq⟨uint8⟩, y: seq⟨uint8⟩, n: nat): (result: bool)
  requires n ≤ |x| ∧ n ≤ |y|
  ensures result ⟹ n = |x| ∨ n ≠ |y|
  decreases |x| - n {
    n = |x|
  ∨ (n ≠ |y| ∧ x[n] < y[n])
  ∨ (n ≠ |y| ∧ x[n] = y[n] ∧ LexLeq(x, y, n + 1))}
```

(a) An example Dafny method from the ESDK.

```
implementation LexLeq (x: Seq Box, y: Seq Box, n: int)
  returns (result: bool) {
  var tmp: bool;
  Entry: goto A, B;
  A: assume n != |x|;
    goto C, D;
  B: assume n == |x|;
    goto E;
  // blocks C to K go here...
  L: assume n != |y| && x[n] == y[n];
   call tmp := LexLeq(x, y, n + 1);
   assert false; // Added by DTest
   goto Return;
  Return:
   result := n == |x| || (n != |y| && x[n] < y[n])
                      || (n != |y| && x[n] == y[n] && tmp);
   return;}
```

(b) (Simplified) Boogie translation with a trap assertion.

```
method {:test} test() {
  var d0: seq⟨int⟩ := [0, 133, 188];
  var d1: seq⟨int⟩ := [0, 133, 187];
  var r0 := LexLeq(d0, d1, 1);
  expect r0 ⟹ 1 = |d0| ∨ 1 ≠ |d1|;}
```

(c) Generated Dafny unit test.

```
[Xunit.Fact]
public static void test() {
  var d0 = Sequence⟨int⟩
    .From(0, 133, 188);
  var d1 = Sequence⟨int⟩
    .From(0, 133, 187);
  bool r0 = LexLeq(d0, d1, 1);
  if (r0 ∧
     d0.Count ≠ 1 ∧
     d1.Count = 1)
   throw new Dafny.Exception();}
```

(d) (Simplified) generated C# unit test.

Figure 2.2: Unit test generation example.

within a valid range for x and y, and a postcondition (line 3) ensuring that if the result is true then either we have reached the end of x or we have not reached the end of y. Note that the postcondition was not present in the original code, but we added it to more fully illustrate the features of DTest.

Dafny verifies programs by translating them to the Boogie intermediate verification language [8, 16] and then verifying the Boogie code. For our example, DTest translates the Dafny code in Fig. 2.2a to the Boogie implementation in Fig. 2.2b. Note that this translation differs from one the regular Dafny to Boogie translator would produce—we discuss the differences in Sec. 2.3.1. The code in Fig. 2.2b takes three input parameters that directly map to the parameters in the Dafny code. The parameters x and y have type Seq Box, which is the type that the Dafny translator uses to encode sequences in Boogie. For clarity, we use Dafny notation in place of Boogie function calls for element access, a[i], and sequence length, |x|. On entry to the implementation, the Boogie program proceeds to either block A or B, each corresponding to one of the two possible values of the Boolean expression on line 5. Note that in Boogie, control flow is captured by non-deterministic branches to blocks guarded by assume statements. For example, here, block A is guarded by an assumption $n \neq |x|$ falsifying the condition on line 5 in Fig. 2.2a. Thus, the Boogie code has a block for each term of the Boolean expression in the original Dafny. Therefore, block coverage of the Boogie code essentially corresponds to branch coverage of the original Dafny code.

Recall that DTest finds inputs that reach target branches by iteratively inserting assert false in each block and then extracting a counterexample from the verifier. We call such assertions *trap assertions* because we do not expect the prover to successfully verify them. Here, DTest has added a trap assertion on line 19 in the Boogie code, with the goal of covering block L.

When we ask Boogie to verify the code in Fig. 2.2b, the verifier produces a counterexample. The counterexample itself is not human readable, but recent work [24] allows us to infer counterexample arguments x=$[0, 133, 188]$, y=$[0, 133, 187]$, and n=1. DTest then uses these arguments to produce the unit test in Fig. 2.2c. Here the :test annotation signals to the compiler that this

method should be compiled as a unit test in the target language of choice. The body of the test begins by constructing sequences d1 and d2 which, along with the literal 1, are the counterexample arguments. Next, the test case calls LexLeq with these arguments. The expect statement on line 29 is a runtime assertion. Here we check that the result satisfies the postcondition. Thus, the test not only covers block L, but also adds a level of assurance to the emitted code by introducing runtime checks.

Finally, Fig. 2.2d shows the C# unit test DUnit generates for the example Dafny unit test. Lines 32–35 correspond to lines 26–27 in Dafny and construct the counterexample arguments later used in the method call on line 36. The conditional on lines 37–40 throws an exception in case of a postcondition violation. Note that DUnit converts the :test annotation in Dafny to XUnit.Fact, which allows us to run the resulting test using .NET's XUnit framework [117].

2.2 Unit Testing and Mocking Frameworks

To support DTest, over the span of several years we developed DUnit and DMock, unit testing and mocking frameworks for Dafny. In this section, we describe the new unit testing and mocking constructs we introduced to the Dafny language as well as how we compile them into C#, the target compilation language used by most open-source Dafny projects.

As we discuss in Sec. 4.1 (see Fig. 2.2c), DUnit introduces the :test attribute for annotating unit tests. Within a unit test, we introduce expect statements to specify runtime assertion checks. In contrast to standard Dafny assert statements, the Dafny verifier does not prove expect statements but instead assumes they hold. Dafny compiles expect statements into runtime assertions in the target language, whereas assert statements are removed from compilation.

In addition to this basic unit testing support, we also introduce support for runtime *mocking*, which allows creation of objects based on a description of their behavior. When compiling to C#, we translate Dafny mocks into code that uses the popular Moq library [87]. (Note that DMock also supports compilation to Java using the Mockito library [86], but we focus on C#.)

```
1  method {:synthesize} getKeyring
2    (client: ManagementServiceClient, key: String, arn: Identifier, tok: TokenList)
3    returns (o: AwsKmsKeyring)
4    ensures fresh(o)
5    ensures o.client=client ∧ o.Key=key ∧ o.Arn=arn ∧ o.Tokens=tok
```

(a) Dafny.

```
6  public static AwsKmsKeyring getKeyring
7    (ManagementServiceClient client, String key, Identifier arn, TokenList tok) {
8    var mock = new Mock⟨AwsKmsKeyring⟩();
9    mock.CallBase = true;
10   mock.SetupGet(x ⇒ x.client).Returns(client);
11   mock.SetupGet(x ⇒ x.Key).Returns(key);
12   mock.SetupGet(x ⇒ x.Arn).Returns(arn);
13   mock.SetupGet(x ⇒ x.Tokens).Returns(tok);
14   return mock.Object;}
```

(b) C#.

Figure 2.3: An example use of the :synthesize attribute for mocking (simplified).

The key reason we developed DMock is to support the creation of heap-based structures (i.e., objects), which DTest heavily relies on. In particular, mocking solves the problem of having to synthesize a sequence of calls to constructors and other API methods to put a given object into the required state. In DMock, we introduce the :synthesize attribute for annotating mock methods, which is accompanied by postconditions describing the method's return value. We can use such postconditions to specify mocking behavior of constant instance fields and functions. DTest can infer from counterexamples the arguments with which to call mock methods, and we do not allow mocking of side-affecting properties, which ensures that the objects are consistent with preconditions and type invariants.

Fig. 2.3a gives an example mock method that generates a new AwsKmsKeyring object and sets its instance fields to values given to the method call as arguments. DTest automatically produced this method (simplified) and relevant arguments to call it with while generating tests for the ESDK. First, line 4 uses Dafny's fresh keyword to ensure the returned object is new and not aliased by an existing variable. The subsequent postconditions specify the values of each of the object's constant fields. DMock compiles this method into the code in Fig. 2.3b. On line 8, we

use Moq to create a new class that extends AwsKmsKeyring. On the next line, we ensure that by default the mocked class behaves exactly as the original class it extends. Then, we override the field getter methods to return the values provided as arguments. Finally, we return a new instance of the mocked class that, by construction, behaves exactly as specified by the postconditions in Fig. 2.3a.

DMock also supports mocking of instance functions by redefining their behavior with arbitrary expressions. For example, we can add the following postcondition to the mock method in Fig. 2.3a to ensure that a call to the Identity instance function simply returns its argument: ensures ∀ arg:int • o.Identity(arg) = arg. DMock compiles this postcondition into the C# statement below (which would be added to Fig. 2.3b) to override the behavior of the Identity function:[1]

```
mock.Setup(x ⇒ x.Identity(It.IsAny⟨BigInteger⟩())).Returns((BigInteger arg) ⇒ arg);
```

This functionality is particularly useful for instantiating traits, which are Dafny types similar to interfaces in Java that also cannot be instantiated directly. A method annotated with :synthesize can both return an object extending a given trait and ensure the instance functions of that object behave as the postconditions dictate. Note that we can only mock instance functions, not methods, since method calls cannot appear inside postconditions, which are expressions. However, Dafny programs are typically written in a functional style, and hence DTest can still handle most real-world uses of traits.

2.3 Automated Test Generation

In this section, we describe DTest's test generation approach (steps 1-5 in Fig. 2.1).

2.3.1 Custom Dafny to Boogie Translation

DTest customizes (step 1 in Fig. 2.1) Dafny's standard translation to Boogie with two key modifications to support automated test case generation.

[1] Dafny's int is compiled to C#'s BigInteger because in Dafny integers are unbounded.

Preprocessing to Support Inserting Trap Assertions. The Dafny code we have analyzed makes extensive use of functions. Function bodies are syntactically expressions and are translated as such into Boogie. However, an assertion is a statement, and cannot be inserted into the body of a Boogie function. To address this issue, DTest preprocesses the Dafny code to turn functions into function-by-methods, which are functions with an equivalent imperative definition provided as a method. In our case, we wrap the original expression in a return statement, which then prompts the translator to create an imperative Boogie implementation. Hence, for each input function-by-method, Dafny emits both a standard Boogie function—used for verification—and an imperative implementation, as in Fig. 2.2b. DTest can then insert trap assertions into implementations' bodies.

Inlining. If we are using DTest to generate unit tests of individual methods, no further translation steps are needed. However, an issue arises if we wish to generate system-level tests via calls to a `main` method entry point or similar. The challenge is that Boogie verifies methods one at a time, and any callee methods are represented by their specifications. Any trap assertions aside from those in a `main` method will essentially be "hidden" behind the specifications of the methods they are inside of.

Our solution to this problem is inlining: DTest can optionally inline the program into a user-specified `main` method before proceeding with test generation. Recursive methods can also be inlined (unrolled) up to a manually chosen bound. This way, DTest can provide coverage of the entire Dafny program. Boogie supports inlining, but to take advantage of this support, we have made several changes to the Boogie code emitted by Dafny. These changes allow translating functional-style code, such as conditional expressions, to their imperative equivalents, such as conditional statements, which makes the code more amenable to trap-assertion injection and inlining.

2.3.2 Trap Assertion Injection

DTest generates tests while iterating over the basic blocks of the Boogie representation. Iteration happens in reverse topological order of the control-flow graph in order to greedily reduce the number of tests by generating tests that are likely to cover multiple blocks at a time. For each block that has not yet been covered, DTest inserts a trap assertion (step 2 in Fig. 2.1) and queries the Boogie verifier for a counterexample (step 3). Alongside the counterexample, the verifier also reports the error trace, i.e., blocks leading up to the trap assertion that the counterexample also covers. We use the error trace to prune away already covered blocks. DTest then uses previous work [24] to extract the counterexample to a Dafny-like format and then concretizes the result (step 4 in Fig. 2.1).

One can construct more complex trap assertions that fail when a program takes a specific path through the control flow graph. We can, therefore, use DTest to generate test suites with path-coverage guarantees, although we do not fully explore this use case here and only apply this version of DTest to study the sizes of potential test suites (see Sec. 2.4.2.)

Note that a successfully verified trap assertion serves as proof that no input can cause a given block to be visited, i.e. it signifies the presence of dead code. This also allows us to uncover dead code using DTest, which is an option we implemented but have not experimented with extensively.

2.3.3 Unit Test Generation

The key challenge DTest faces when generating unit tests (step 5 in Fig. 2.1) involves selecting concrete values that are not constrained by the counterexample because they are irrelevant to a particular assertion failure. For example, consider the method in Fig. 2.2a. A counterexample returned by the solver may suggest that calling the method with $n = 1$ and x being a one-element sequence covers block B. To generate a unit test, DTest also has to emit a value for x's single element. DTest is free to choose any value assuming it satisfies the corresponding type constraint if any such constraint is present.

To generate such values, DTest relies on *witnesses*—user-supplied (using the witness keyword in Dafny) or sometimes automatically inferred values that Dafny uses to prove a given type is nonempty. We define such values for all primitive types and collections (e.g., 0 for integers), and user-defined witnesses are typically available for subset types, i.e., types that are constrained with arbitrary predicates. In the rare case that a user does not suggest a witness for a given subset type, DTest will emit a default value for the corresponding supertype, which may lead to a test that violates a type constraint. We call any such test that violates the specification of the target method or a type constraint *unreliable* and discuss all cases in which DTest might generate such tests in Sec. 2.3.4.

One way to exclude unreliable tests would be to verify them in Dafny. However, DTest might generate a correct test while at the same time failing to find the right value for a ghost variable — irrelevant at execution — to make the test verify. Moreover, some tests may be unreliable yet still explore the targeted branch when compiled to C#. Therefore, to allow more flexibility, we aim to filter unreliable tests at runtime with checks that preemptively terminate execution if a test violates a method precondition over non-ghost fields, violates a type constraint, or calls a trait instance method that is not explicitly mocked by DTest. For example, in Fig. 2.2c we add such a check after the last local variable initialization on line 27 as the runtime assertion expect $1 \leq |\mathsf{d0}| \wedge 1 \leq |\mathsf{d1}|$, "Unmet precondition". In our evaluation, such runtime checks terminate 94% of all unreliable tests that would otherwise have led to difficult-to-interpret failures.

To strengthen the assurance provided by the tests, DTest converts postconditions of methods under test into runtime assertions such as the one on line 29 in Fig. 2.2c. We support all specification constructs allowed by Dafny, provided they are not ghost. Whenever DTest encounters ghost specifications, which are infrequent in our evaluation, it does not create a corresponding runtime check; this, of course, does not affect coverage.

2.3.4 Limitations

DTest has several limitations that can either prevent it from being able to generate a test for every block in the Boogie representation of a given procedure or might cause DTest to occasionally produce unreliable tests (Sec. 2.3.3).

- **Solver Timeouts.** As is the case with any tool that relies on an SMT solver, timeouts may occur, in which case DTest might not cover some blocks. We currently set the timeout to 5 seconds, and our empirical evaluation shows that increasing the timeout does not make a significant difference.

- **Spurious Counterexamples.** Dafny might generate a spurious counterexample (i.e., one that does not in fact lead to a trap assertion violation) due to several reasons. First, specifications, such as post-conditions or loop invariants, might be over-approximations that under-constrain the program state. Second, the Dafny translation into Boogie might not provide a complete axiomatization of some features, such as set cardinality. Third, the back-end SMT solver itself is incomplete in the presence of quantified formulas, which Dafny always generates. This can lead to a counterexample that does not expose a trap assertion or may even violate method preconditions.

- **Information Elided in the Counterexample.** If the user does not provide a witness for a certain subset type (Sec. 2.3.3), DTest may not be able to generate a value that satisfies the corresponding type constraint.

- **Ghost Specifications.** DTest cannot compile ghost specifications into runtime checks, so there could be unreliable tests we fail to identify.

- **Unsupported Language Features.** DTest does not support tuples, arrays, infinite maps, infinite sets, or multisets. These Dafny types and collections are rarely used in practice (sequences are used instead of arrays; finite maps and sets are preferred to their infinite counterparts) and only appear in a handful of methods in our benchmarks. Moreover, DTest

does not fully support traits and function types. For any argument of a function type, DTest synthesizes a lambda expression with a matching type signature. For example, for a function that maps an integer to an integer, DTest synthesizes a lambda expression that always returns 0. Given that function types are rarely constrained, this approach works in the majority of cases. For a discussion of traits, see Sec. 2.2.

2.4 Empirical Evaluation

We perform two experiments to evaluate the testing toolkit. First, we evaluate DTest's running time and coverage achieved on two preexisting Dafny projects. Second, we compare the number of tests DTest generates to the minimal number of tests required for full coverage.

The subject programs for our evaluation are two real-world projects: the Dafny utilities library (DUTIL) [77] and the AWS Encryption SDK (ESDK) [5]. DUTIL spans 1382 lines of code and presents a collection of useful methods for non-linear arithmetic; manipulating Dafny maps, sequences, and sets; and performing miscellaneous operations. The ESDK comprises 4596 lines of code and implements a Dafny-verified encryption library, which provides an interface between encryption backends and consumer applications. The two projects are, to the best of our knowledge, the largest open-source Dafny programs, with the exception of Ironclad [51] which, despite manually updating it to the latest Dafny syntax, we were unable to get to verify (and hence use in our experiments). While our benchmarks are small by industry standards, they are representative of how Dafny is used in practice and showcase most of Dafny's features.

2.4.1 Unit Testing and Coverage

In our first experiment, we measure the statement and branch coverage on the binary obtained by compiling DUTIL and the ESDK to C#. To maximize DTest's performance, we augmented the ESDK with about two dozen *witnesses* (Sec. 2.3.3). Doing so took us less than an hour of manual work. We find that DTest can quickly generate tests that provide sufficient coverage to identify

unexpected behavior in an external library.

Performance. DTest took 158 minutes to generate 918 tests for the 436 methods in the two benchmarks. This does not include methods that exist only to aid verification and are not compiled, methods that have no body in Dafny (external methods), and methods introduced by the Dafny compiler (e.g., ToString). Fig. 2.4 shows that the runtime it takes DTest to process one method is close to linear in the number of blocks in the Boogie representation of that method (Pearson's coefficient ≈ 0.86, $p < 0.0005$). The outliers are methods for which DTest can cover multiple blocks with one test, such as the methods we analyze in Sec. 2.4.2. Methods that cause solver timeouts are also a source of outliers.

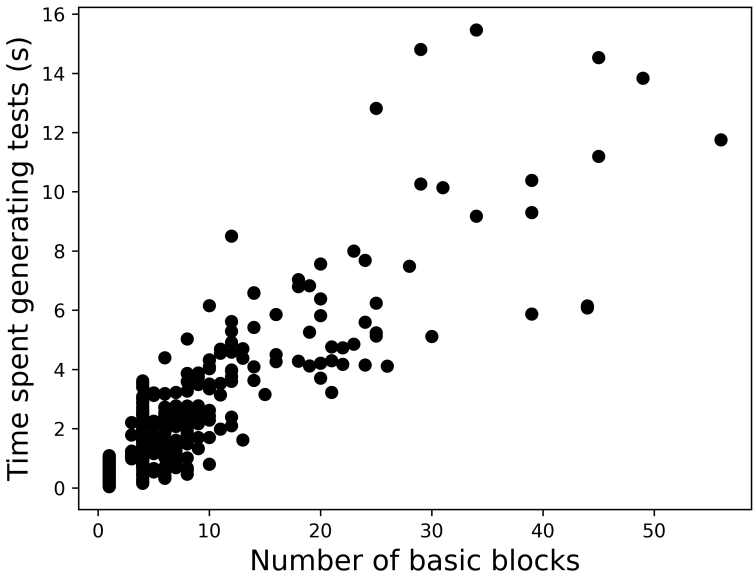


Figure 2.4: Runtime of DTest on methods from ESDK and DUTIL.

Coverage. Table 2.1 shows the coverage that the tests achieve on the compiled C# code, as measured with the Coverlet framework [29]. We give the results for each source folder in DUTIL and ESDK, including the lines of code (LOC) count, target method count, and the number of generated tests for each entry. Each folder consolidates files with similar functionality. The table includes a *Block* column showing the fraction of Boogie basic blocks for which DTest reports that it successfully generated tests. We report the actual statement and branch coverage in the last two columns, with unreliable tests (Sec. 2.3.4) not contributing to the result. We took all methods compiled from Dafny to C#, even those for which DTest fails to generate tests, into account when measuring coverage. We achieve 62% statement and 58% branch coverage on the ESDK. The lower coverage of the SDK folder is

Table 2.1: Overview of achieved coverage.

	Source Folder	LOC	Methods	# Tests	Boogie Block	C# Statement	C# Branch
DUTIL	Maps	55	7	13	100%	98%	88%
	Sequences	1114	131	169	67%	76%	84%
	Sets	42	4	8	100%	83%	69%
	Nonlinear Arithmetic	78	7	11	72%	91%	88%
	Misc.	93	16	22	90%	98%	88%
	Total	**1382**	**165**	**223**	**72%**	**79%**	**84%**
ESDK	Crypto Material Providers	1871	67	270	87%	62%	50%
	Crypto	153	13	25	90%	82%	70%
	Generated	61	4	8	83%	100%	100%
	SDK	1878	116	205	58%	51%	45%
	Standard Library	414	45	121	90%	93%	93%
	Util	219	26	66	74%	75%	91%
	Total	**4596**	**271**	**695**	**77%**	**62%**	**58%**

due to extensive use of traits, which DTest only partially supports (Sec. 2.3.4). These results are comparable to the thresholds that the ESDK developers set as a minimum bar for manual tests as part of their overall testing and verification strategy, with DTest scoring above the 35% threshold for branch coverage but below the 80% threshold for statement coverage.

We observe in the table that the Boogie basic block coverage does not match the C# statement coverage, but is either an over- or under-approximation. The difference is due to two key factors. First, a C# test generated for one method might cover code in a method invoked by it, which at the Boogie level we cannot observe since we are doing intraprocedural test generation in the experiments (no inlining). This leads to the Boogie coverage being an under-approximation of the C# coverage. Second, some tests may be identified as unreliable at runtime, and so they contribute to the Boogie coverage but not to the C# coverage, leading to the former being an over-approximation of the latter.

Overview of Tests. Of the 918 tests DTest generated for DUTIL and ESDK, 85 are unreliable. Of these, our runtime checks preemptively terminated 80 (see Secs. 2.3.3 and 2.3.4): 11 violate preconditions, 40 violate type constraints, and 29 call methods on traits that are not mocked.

Five more tests fail because DTest does not fully support function types. As we describe above (Sec. 2.3.4), arguments of function types are generated from type signatures rather than counterexamples. Overall, half of the generated tests have no return value check due to absence of postconditions. However, even such tests check for runtime errors, which is valuable as shown by the success of black-box fuzzers (that do not check return values either) [80]. In our experiments, the outcomes of two tests are worth noting: one test causes an external method to throw an exception, which, however, is allowed by that method's signature. Another test causes the execution to continue for an indefinite amount of time (we killed the process after two hours). The developers identified this case to be from a particular internal test method (not part of code exposed to the user) for an external .NET RSA library. Thus, our test uncovered an external library behaving differently from the developers' expectations, and they fixed the test method accordingly.

2.4.2 Test Suite Size

In our second experiment, we evaluate the size of the test suite DTest generates by comparing it to a minimal number of tests required to achieve full coverage. Test suite size is an important factor for software development and has prompted significant research effort [57, 50].

For the purpose of this comparison, we designed an algorithm to enumerate sets of control flow paths of a Boogie procedure in order of increasing set cardinality, terminating when we find a set of paths that guarantee full coverage, are all feasible, and that we can generate tests for. We determine the feasibility of a path via a query to the SMT solver, in which a trap assertion is added that fails only if all the blocks along the path are visited. We then generate a test for each path in the same way that DTest would generate a test for a given block. This approach is exponential in the number of SMT queries (running on all benchmarks as in Table 2.1 would take weeks), but we do allow the users of DTest to optionally use this costly method since reducing the number of tests is sometimes of utmost importance (e.g., if tests are being executed over and over again as a part of continuous integration).

Table 2.2: Comparison of minimization strategies.

Method	# blocks	Running time (s)		# Tests	
		Default	Optimal	Default	Optimal
UTF8.Uses3Bytes	53	18	1860	13	12
UTF8.Uses4Bytes	53	19	422	13	12
UTF8.ValidUTF8Range	21	12	36	9	6
Base64.IsBase64Char	19	16	112	8	3
AwsKmsArnParsing.AwsArn.Valid	16	17	30	6	2
Base64.IndexToChar	16	14	19	5	5
Base64.Is1Padding	16	13	24	6	2
Base64.Is2Padding	16	14	25	6	2
Sorting.LexicographicByteSeqBelowAux	14	8	8	5	3

To compare the default (greedy) and optimal approaches, we selected nine methods from DUTIL and ESDK with the most complex control flow as determined by the number of basic blocks in their Boogie representation. Such methods present high potential for test minimization, since one carefully chosen test could cover many blocks; we expect differences between the two approaches to be less pronounced when viewed in a broader context. We omit any methods for which DTest could not reach all blocks, or for which it generated unreliable tests.

Table 2.2 summarizes the results of this comparison. We give the number of basic blocks in the Boogie representation of each method in a separate column. For each Dafny method and minimization technique, we report the time that the test generation process takes (mean of three runs), and the number of tests in the resulting suite. The default (greedy) algorithm appears to generate more tests whenever there are independent branching points in the control flow of the method, i.e., when the choice of the path at one branching point does not dictate the choice at the next one. Even so, the greedy algorithm is within two tests of minimal for four of the methods, and it never generates more than three times the minimal number of tests. It is also several times faster than the alternative.

2.5 Related Work

Testing of Verification-Ready Languages. Test generation has been explored in the context of Dafny by Delfy [103, 26], a concolic test generation tool. Delfy has not been updated for nearly ten years and only supports a limited subset of Dafny, and hence a direct comparison was not possible. In contrast to Delfy, DTest fully supports the features commonly used by Dafny programmers, such as algebraic datatypes, sequences, sets, and maps. Unlike Delfy, which relies heavily on compilation to C# for both concrete and symbolic execution, DTest is independent of the target language since it generates tests (in Dafny) from counterexamples provided by the Dafny verifier itself.

Another tool for automated testing of Dafny is XDSmith [59], which randomly generates Dafny programs with known verification outcomes and uses these programs to test the Dafny verifier and compilers. XDSmith is complementary to DTest—the former focuses on testing the Dafny toolchain itself in isolation, while the latter helps to increase assurance that the compiled target programs are correct, particularly in their interaction with external libraries.

Concrete execution of verifier-produced counterexamples has been explored in the Why3 verification environment [10, 40]. The goal of this work is to ascertain the validity of a counterexample by observing the runtime behavior it triggers under various assumptions. DTest, by contrast, relies on the correctness of counterexamples to generate tests. This technique and, more broadly, the use of a verifier to generate tests has been explored in the context of other languages, such as C [12] and B [6], but Dafny presents a particular challenge due to its verification-related features (e.g., rich type system, specifications).

To the best of our knowledge, neither Boogaloo [95], a nondeterministic interpreter for Boogie, nor Symbooglix [78], a symbolic execution engine, have been used to generate test suites for Boogie or Dafny programs although both tools can be used to explain a failing assertion. Symbolic-execution-based program exploration algorithms and the verifier have different trade-offs, especially in the presence of loops. We plan to develop, as future work, a portfolio-based approach, similar to CoVeriTest [13], with several backend reachability analyses.

Automated Software Testing. There is a large body of work on automated software testing, involving techniques such as fuzzing (see [80] for a survey), symbolic execution [7], and others. We might augment DTest with some of these techniques in the future since, for example, the approach used by QuickCheck [27] and the related family of fuzzers for generating values of function types offers more flexibility than DTest's current implementation. One of the challenges often accompanying automated testing is object initialization. Our approach to this problem is close to lazy symbolic initialization [65], a process whereby an object is initialized on an "as-needed" basis—we similarly override the value of an object's field only if it is constrained by the counterexample or a precondition.

Mocking. A number of mocking frameworks exist for various languages, of which Mockito [86] and Moq [87] are some of the most popular options for Java and C#, respectively. For our purposes, it is crucial that a mocked object behaves exactly like an instance of the corresponding type unless an instance field or function is specifically redefined by the user. DMock relies on both Mockito and Moq to support this functionality, which is sometimes also referred to as *spying*.

2.6 Conclusion

In this chapter, we presented a toolkit for automated testing of Dafny programs: DUnit (unit testing framework), DMock (mocking framework), and DTest (automated test generation). The main component of the toolkit, DTest, works with the Boogie representation of a Dafny program to generate tests that (i) target branch coverage of the compiled code and (ii) contain runtime assertions extracted from method specifications in the Dafny code. We evaluated the coverage DTest achieves on several preexisting Dafny programs, showed that it can help identify unexpected behavior in external libraries, and compared it to an alternative more costly solution that optimally minimizes the number of tests. Overall, our results show that DUnit, DMock, and DTest are a promising toolkit for automatically generating high coverage tests for Dafny.

Chapter 3

Metamorph: Synthesizing Large Objects from Dafny Specifications

Program synthesis aims to produce executable code that adheres to a specification written by the user. Synthesis tools make use of large language models [90], Petri nets [39], refinement types [96], program sketches [102], evolutionary algorithms [42], abstract domains [48], and other techniques. Synthesis has many applications, including, among other things, in automated test generation, where a test engine like DTest might require generating an object with certain properties to trigger a given execution path. In this chapter, we focus on the problem of synthesizing sequences of Dafny method calls that set up an object to a certain requested state. This work is partially motivated by limitations of DTest and aims to provide a more robust alternative to DMock, which is described in the previous chapter.

In our approach to object synthesis, we seek to take advantage of user-provided specification of an API, including method pre- and postconditions and an object invariant. This setting, sometimes referred to as *specification-guided synthesis*, is becoming increasingly relevant with the availability of fully verified software and libraries that contain such specifications [5, 20, 22]. While there is a breadth of work on applying proof theory to program synthesis in general [97, 60, 104] and on the synthesis of Dafny code in particular [114, 75], the synthesis of method call

sequences in this setting remains an open challenge. Analyzing the effects that multiple consecutive method calls have is computationally expensive, especially when symbolically reasoning about method arguments. In part for this reason, existing work on synthesizing sequences of method calls typically puts an upper bound on the number of calls in the solution, which can be as low as five [84] or eight [120],[1] and is, therefore, insufficient to solve certain problems. For example, synthesizing instances of various data structures, which is the task we focus on in this chapter, may require dozens of method calls (up to 57 in our benchmarks suite). To compound the problem, Dafny's expressiveness comes at the cost of a 1500-line long prelude with axioms and definitions, which makes even simple queries to the underlying SMT solver take longer than in verification tools that have a narrower scope (e.g., Verus [71]). To synthesize Dafny programs of the size that we need, we must, therefore, limit both the complexity and the number of SMT queries that we make. (In Section 3.1, we give an example of the kind of synthesis problem that we aim to solve.)

In this chapter, we introduce a new deductive, goal-directed synthesis approach that uses counterexamples returned by the solver to reason about the effects of method calls one at a time, limiting the complexity of the underlying SMT queries. We begin the search for a solution in the target object state that the user specifies. We then ask the Dafny verifier to prove that this state cannot be reached by calling a certain method. From the counterexample that the verifier returns, if any, we reconstruct a new state in which we can call that method and get to the target state. For example, if our goal is to synthesize a list with three elements, this procedure helps us learn that we can reach the goal by pushing an element onto a list that previously only had two elements. We recursively repeat this process from newly discovered states until we reach the initial state that the object must be in upon construction (e.g., an empty list). While this approach limits us in that we do not support non-linear control flow, it does allow us to synthesize programs that make dozens of calls to external library methods. (We describe the synthesis algorithm in Section 3.2.)

[1]The authors of MSeqMynth [120] also conduct an experiment in which they do not put an explicit limit on the number of method calls but use a timeout to bound the state exploration process. Even so, their approach relies on reasoning about object states symbolically, which we found to be infeasible in the context of Dafny, as we describe below.

To combat the exponentially growing search space, we guide synthesis using one of two metrics that estimate the distance between object states. We represent object states as conjunctions of atomic constraints, so a simple way to estimate distance is by counting the number of constraints that hold in one state but not the other (see, e.g., [17]). However, this greedy approach may guide the search towards a local minimum, such as when two states differ by only one constraint but require many method calls to transition between them. To address this, we propose a novel *piecewise distance metric*, which puts a lower bound on the number of method calls needed to transform one state into the other, allowing us to view the synthesis as weighted A* search. To enable the piecewise distance metric, Metamorph goes through a *pre-training* phrase, where it analyzes the given API to identify which methods available to the synthesis engine may satisfy which constraints. During synthesis, Metamorph combines this information using integer programming to measure distances between specific states. For example, we learn that a call to `Push` may add an element to the list while a call to `Pop` does the opposite, so, in the absence of other methods, synthesizing a list with three elements requires at least three calls to `Push`. While it comes with an optimality guarantee, the piecewise distance requires pre-training on each new API, introducing a trade-off where one sacrifices some of the immediate flexibility of the greedy approach for an optimality guarantee. (We describe the possible distance metrics in Section 3.3.)

We implement these ideas in a tool called Metamorph. Metamorph uses the Dafny verifier as a backend and can, therefore, be easily extended to support future versions of the language. Each program that Metamorph synthesizes consists of calls to available library methods, where synthesized objects can be passed as arguments to downstream method calls to produce more complex objects. (We discuss some of the implementation details in Section 3.4.)

We evaluate Metamorph in four complementary ways. First, we test Metamorph's ability to generate large objects on six benchmarks, each of which defines a data structure, such as a doubly linked list, an array, a queue, a binary tree, or a graph. The benchmarks come with specifications for every method that manipulates the data structure as well as with an object invariant that must hold between method calls. We extract one benchmark from an open-source Dafny project,

adapt two other benchmarks from prior literature, and develop three benchmarks to test our approach on other data structures. Metamorph can successfully construct programs that require up to 57 method calls per instance and compares favorably to a naive baseline approach, where we use Dafny itself as a synthesis tool. Second, we compare the two distance metrics available to Metamorph on these same benchmarks. We show that while the two metrics yield similar performance, the piecewise distance metric provides an edge in cases when the API has an underspecified object invariant or the solution to the synthesis problem requires a call to a method that might temporarily add additional constraints to the object's state. Third, we integrate Metamorph with DTest, Dafny's automated test generation toolkit, and show that Metamorph can synthesize test inputs for parts of the AWS Cryptographic Material Providers Library [4] that DTest alone is not able to cover. Fourth, to demonstrate Metamorph's broader applicability to problems beyond the synthesis of data structure instances, we use Metamorph to generate executable bytecode for a simple virtual machine, which is inspired by the Dafny Ethereum Virtual Machine project [21]. (Section 4.4 describes the experiments we conducted in more detail.)

In summary, the main contributions of this chapter are as follows:

- We introduce a new counterexample-driven synthesis technique, where we reason about the effects of method calls one at a time and recursively reconstruct object states from Dafny counterexamples. By relying on counterexamples, we can view object states as conjunctions of atomic constraints, allowing us to design two distance metrics that help guide the search.
- We implement our counterexample-driven approach in a tool called Metamorph, which can synthesize objects in the Dafny programming language.
- We evaluate Metamorph by showing that it can synthesize large objects and comparing it to a baseline technique. Furthermore, we integrate Metamorph with Dafny's automated testing engine to generate tests for code that the engine has previously been unable to cover.
- We make Metamorph and associated benchmarks available as an artifact on Zenodo [91].

```
41 datatype User = User(friends:set<nat>, requests:set<nat>)
42
43 class SocialNetwork {
44
45   ghost var users:map<nat,User>
46
47   constructor {:use} ()
48     ensures this.users = map[]
49
50   method {:use} AddUser(id: nat)
51     requires IsValid() ∧ id ∉ users
52     modifies this
53     ensures IsValid()
54     ensures ∀ u • u ∈ old(users) ⟺ u ∈ users ∧ u ≠ id
55     ensures ∀ u • u ∈ old(users) ⟹ old(users)[u] = users[u]
56     ensures id ∈ users ∧ users[id] = User({}, {})
57
58   predicate IsValid() reads this {
59     ∧ ∀ user • user ∈ users ⟹
60       (∀ r • r ∈ users[user].requests ⟹ r ∈ users ∧ r ≠ user) ∧
61       (∀ c • c ∈ users[user].friends ⟹  c ∈ users ∧ c ≠ user) ∧
62       (∀ u • u ∈ users ⟹ (user ∈ users[u].friends ⟺ u ∈ users[user].friends) ) }
63
64   method {:use} RemoveUser(id: nat) // Removes a user from the network. . .
65   method {:use} FriendRequest(from: nat, to: nat) // Sends a friend request..
66   method {:use} RequestResponse(from: nat, to: nat, answer: bool) // Responds to a request. . .
67 }
```

Figure 3.1: A partial API specification written in Dafny.

3.1 Motivating Example

In this section, we illustrate Metamorph on one of our benchmarks, which defines an API for manipulating a hypothetical social network. Figure 3.1 shows a partial specification of the API. The social network is a graph where the nodes are uniquely indexed users. Undirected edges connect users who are friends, and directed edges indicate incoming friend requests from one user to another. The state of the social network is fully modeled by the map users (line 45), which is a *ghost* field, meaning that it is used in specifications but cannot be modified or even referenced once the code is compiled. The actual implementation of the data structure is written in C# and is not available to Metamorph, so the only way it can modify the network's state is through four

different API methods: AddUser adds a new user to the network, RemoveUser removes an existing user, FriendRequest sends a request from one user to establish a friendship with another user, and RequestResponse either rejects or accepts an incoming request.

We show the full specification of the AddUser method; the specifications (not shown) of the other methods follow a similar pattern. The precondition (the requires clause) states that the user ID provided as an argument does not yet exist in the database. The modifies clause indicates that the state of the network changes following the method call. The postcondition (the ensures clauses) asserts that (i) the set of users in the database remains the same before and after the method call modulo the user being added (line 54), (ii) all existing connections and friendship requests are preserved by the method (line 55), and (iii) the new user is initialized with no friends or friend requests (line 56). Additionally, the AddUser method maintains the object invariant IsValid, which must hold between calls to any API methods. The invariant guarantees the internal representation of the graph is consistent with itself: no edge can point to a user that does not exist (lines 60–61), and all friendship edges must be present in both directions (line 62). Such object invariants are a common paradigm in verification, and we assume that the user provides one as part of the API.

To invoke Metamorph, one must define the synthesis goal as a predicate that describes the desired object. Such predicates can also be emitted by other programs. For example, in Section 3.5.6 we use DTest to produce tests for a portion of the AWS Cryptographic Material Providers Library [4], where the test inputs are synthesized by Metamorph.

For the current demonstration, we will explore the problem of constructing a social network with three users that form a clique. Figure 3.2a shows a Dafny predicate Target that encodes this synthesis goal, and Figure 3.2b shows the corresponding solution that Metamorph generates. The synthesis goal specifies a network with three users (line 3) who are all friends with each other (lines 4–6). The solution is to add users to the network one by one (lines 5, 6, and 9 in Figure 3.2b), and then to initiate and accept three friend requests (lines 7, 8, 10–13).

The key idea behind Metamorph is to reason about method calls one at a time. This approach allows Metamorph to avoid having to infer arguments for multiple method calls at once, which would be computationally expensive. As an illustration of this reasoning, consider the program in Figure 3.2b that Metamorph synthesizes in under a minute to solve the problem in Figure 3.2a. If we abstract away all concrete arguments in the solution and ask Dafny to infer such values that fulfill the synthesis goal, it takes the solver about 12 minutes to find a solution. Note that this already considerable running time only accounts for inferring primitive arguments and not for the search over possible method call sequences. To synthesize a solution from scratch, we would have to repeat this procedure multiple times for different sequences of methods until we found one that works. In practice, this means that we cannot reason about method arguments symbolically if we intend to synthesize the solution quickly and the solution consists of more than a few method calls. For this reason, Metamorph reasons about the effects of one method call at a time and uses Dafny counterexamples to reconstruct the state that the object must be in before the given method call.

```
predicate Target(s: SocialNetwork)
  reads s
{ s.users.Keys = {0, 1, 2} ∧
  1 ∈ s.users[0].friends ∧
  1 ∈ s.users[2].friends ∧
  2 ∈ s.users[0].friends ∧
  IsValid() }
```

(a)

```
method Solution()
  returns (s: SocialNetwork)
  ensures Target(s)
{ s := new SocialNetwork();
  s.AddUser(0);
  s.AddUser(1);
  s.FriendRequest(0, 1);
  s.RequestResponse(1, 0, true);
  s.AddUser(2);
  s.FriendRequest(1, 2);
  s.RequestResponse(2, 1, true);
  s.FriendRequest(2, 0);
  s.RequestResponse(0, 2, true); }
```

(b)

Figure 3.2: The synthesis problem (a) and solution (b) for the API in Figure 3.1.

Figure 3.3 visualizes the search Metamorph performs to synthesize the program in Figure 3.2b. We annotate each search state with the social network graph to which it corresponds. Red undirected edges stand for friendship connections between users, and blue directed edges represent friend requests. The leftmost state has three fully connected users and corresponds to the target state predicate in Figure 3.2a. Metamorph performs goal-directed synthesis, meaning it starts

with the target state and attempts to find a sequence of method calls that leads to that state. We annotate each edge between states with a method call that Metamorph infers can transform one state into the other.

When Metamorph expands a previously unexplored state (let us call it S), it enumerates all available API methods and asks Dafny if there exists a state $S' \neq S$ such that a call to a given method can transform S' to S. In our running example, two such methods can achieve this for the target state. A call to RemoveUser transforms the state labeled S_2 into the target state by removing a redundant user from the network. A call to RequestResponse similarly transforms the state S_1 into the target state by responding to a friend request from user 0 to user 2. Metamorph infers concrete arguments for these method calls and proves each state transition with the Dafny verifier. Note that Metamorph may have chosen to explore different states in place of S_2 and S_1 in Figure 3.3 because multiple states can be transformed into the target state by calling RemoveUser and RequestResponse. However, the choice that Metamorph makes in selecting S_2 and S_1 does not matter in this case, since all the valid options eventually lead to the same solution modulo the order in which specific users and connections are added to the network. For situations in which the choice does matter, we develop a technique for weakening state predicates to capture multiple options at the same time, as we discuss in Section 3.2.1. As to our present example, after discovering states S_1 and S_2, Metamorph expands these newly found states in the same fashion until, eventually, it finds a path from the initial state to the target state, which is formed by states outlined with a solid line in the figure. This path corresponds to the solution that we show in Figure 3.2b.

A crucial part of Metamorph is the distance metric that determines which state the tool should explore next. Metamorph incorporates two alternative distance metrics, a simpler greedy metric and the *piecewise distance metric*. The latter comes with an optimality guarantee in that it never overestimates the number of methods necessary to transform one object state into another. In the interest of brevity, we give a preview of how Metamorph computes the piecewise distance here, and we discuss both metrics in more detail in Section 3.3. The target predicate in Figure 3.2a

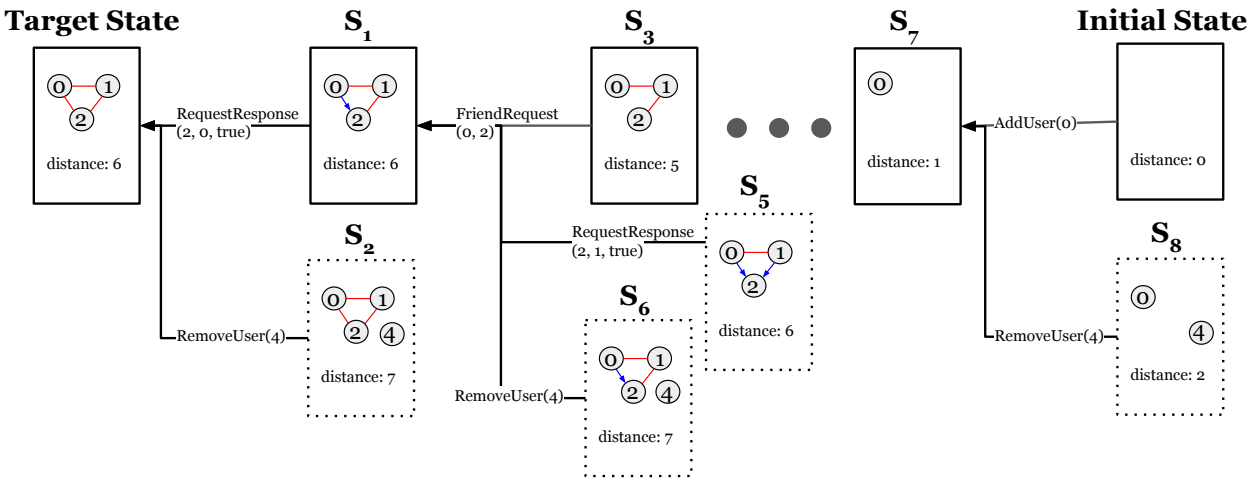


Figure 3.3: Object states Metamorph explores while solving the problem in Figure 3.2. Boxes corresponds to states and edges between states are annotated with method calls that transform one state into another. Solid boxes are states that lie along the path to the solution. Each state is illustrated with the corresponding social network graph, as well as with the piecewise distance from the initial state as computed by Metamorph.

implicitly puts three constraints of type $u \in$ users on the object, where u can be 0, 1, or 2. We say "implicitly" because the predicate uses a different syntax: users.Keys $= \{0, 1, 2\}$. Section 3.3.1 explains how we infer $0 \in$ users, etc. from users.Keys $= \{0, 1, 2\}$; for now, we ask the reader to take this as given. Prior to exploring a given synthesis problem, Metamorph analyzes the corresponding API and determines how many constraints of a given type each method call can affect. For example, Metamorph determines that only a call to AddUser can make a constraint $u \in$ users hold (where u is an arbitrary natural number), and that a single method call can only affect one such constraint. This follows from Dafny verifying the query in Figure 3.7b but failing to verify the query in Figure 3.7a as we discuss in more detail in Section 3.3.3. We refer to this phase of the algorithm as *pre-training*, since Metamorph only needs to perform it once for a given API and can reuse this information later for any other synthesis problem.

In the context of our running example, the information learned during pre-training allows Metamorph to conclude that there must be at least 3 calls to AddUser before one can construct a social network instance that satisfies the predicate in Figure 3.2a. Similarly, Metamorph learns that one call to RequestResponse can satisfy up to 2 constraints of the form $u1 \in$ users[u2].friends

by adding an undirected edge to the network. Since there are 6 such constraints implicit in the target state (undirected edges are recorded twice), we can conclude that at least 3 calls to RequestResponse are necessary. Combining all this information in the piecewise distance metric, Metamorph estimates that at least 6 calls are required to transform the initial state into the target state. We annotate each state in Figure 3.3 with the value of the piecewise distance in that state. Metamorph prioritizes the expansion of states that it estimates to be closer to the initial state in which a newly constructed object would be. This allows Metamorph to find the solution in Figure 3.2b quickly before any of the states outlined with a dashed line in Figure 3.3 are expanded. We provide a more detailed overview of the piecewise distance metric in Section 3.3.3.

3.2 Synthesis Algorithm

In this section, we present Metamorph's synthesis algorithm and then discuss the two subroutines that power the search via queries to the verifier.

3.2.1 Overview

Figure 3.4 gives pseudocode for the main synthesis algorithm. Metamorph's inputs include two state predicates, *target* and *initial*, a list of *methods* that are available to the synthesis algorithm, and (optionally) a list *cs* of constraints that are used by the piecewise distance metric (see Section 3.3.3). The search for a solution proceeds in a typical best-first search fashion where the algorithm maintains the set of *visited* states and the *fringe*. The fringe is a collection of all states that Metamorph knows how to reach but has not yet explored. We represent the fringe with a priority queue data structure, where states are ordered by the sum of (i) their distance to the *target* and (ii) the estimated distance to the *initial* state, which Metamorph computes using one of the distance metrics we discuss in Section 3.3. Initially, the fringe contains a single element, the target state (line 4). Metamorph also tracks all state transitions it uncovers. The set of such transitions, initially empty (line 5), allows Metamorph to reconstruct the path from the initial

```
1: procedure Synthesize(target: State, initial: State, methods: List⟨Method⟩, cs: List⟨Constraint⟩)
2:     visited ← new Set⟨State⟩()
3:     fringe ← new PriorityQueue⟨State, int⟩()
4:     fringe.EnQueue(target, 0)
5:     transition ← new Set⟨⟨precondition: State, command: Statement, postcondition: State⟩⟩
6:     while ¬fringe.Empty() do
7:         curr ← fringe.DeQueueMin()
8:         visited.Put(prev)
9:         for method ∈ methods do
10:            prev, args ← PreconditionFor(method, curr)
11:            if prev ≡ null then
12:                continue
13:            end if
14:            prev ← GeneralizePrecondition(prev, method(args), curr)
15:            if prev ∈ visited ∨ prev ∈ fringe then
16:                continue
17:            end if
18:            transitions.Add(prev, method(args), curr)
19:            if initial ⇒ prev then
20:                transitions.Add(initial, null, prev)
21:                break from while loop
22:            end if
23:            fringe.EnQueue(prev, k1 ∗ DistanceEstimate(initial, prev, cs) +
24:                                 k2 ∗ Distance(prev, target, transitions))
25:        end for
26:    end while
27:    return FindPath(initial, target, transitions)
28: end procedure
```

Figure 3.4: The synthesis algorithm used by Metamorph.

state to the target state at the end of the algorithm's execution.

To find new states, Metamorph first dequeues a previously encountered state *curr* from the fringe and marks it as visited (lines 7–8). Next, Metamorph iterates through all available methods. For each method, Metamorph calls PreconditionFor, which asks Dafny to identify a state predicate *prev* and method arguments *args* such that calling the method from state *prev* transforms the object into state *curr*. In other words, $\{\!\{prev\}\!\}\, method(args);\ \{\!\{curr\}\!\}$ must be a valid Hoare triple. We discuss PreconditionFor in more detail below.

For simplicity, we assume for now that all method arguments are immutable, though we do

allow method arguments to themselves be objects that must be synthesized recursively, as we discuss in Section 3.4. It is possible for *prev* to be *null*, which means that PreconditionFor could not find a state that matches the desired criteria. In particular, if Metamorph is given an unrealizable goal containing conflicting or unfounded predicates, PreconditionFor will return *null* for every query and cause Metamorph to terminate and report that no solution exists. If *prev* is *null* but there are other methods left to try or the fringe is not empty, we continue with the next iteration of the loop (lines 11–13). Otherwise, if *prev* is not *null*, we call GeneralizePrecondition to remove some of the terms in *prev* that might over-constrain the state in a way that inhibits the search process. We discuss GeneralizePrecondition more below.

Next, Metamorph checks whether it encountered the state *prev* before. If *prev* appears in either the set of visited states or the *fringe*, we continue to the next iteration of the loop (lines 15–17). Otherwise, *prev* is new, so we record an edge between *prev* and *curr* that we can later use to reconstruct the solution (line 18). Next, Metamorph checks if *prev* is implied by the *initial* state predicate that serves as a precondition for the synthesis query as a whole (line 19).[2] If *prev* is indeed implied by *initial*, then Metamorph has found a solution, in which case Metamorph breaks out of the outer loop and reconstructs the sequence of method calls from *transitions* using depth-first search (a call to FindPath subroutine on line 27). Otherwise, Metamorph adds *prev* to the fringe (lines 23–24) and continues with the next iteration of the loop.

When calculating the position of *prev* on the fringe, Metamorph computes the weighted sum of the distance between *prev* and *target*, inferred from the *transitions* set, and the estimated distance between *prev* and *initial*. Depending on the metric Metamorph uses to estimate the latter and the coefficients involved, the algorithm can function as a breadth-first search, greedy search, or A* search. We discuss this in greater detail in Section 3.3. The search ends when Metamorph finds a solution or when the fringe is empty, which means a solution could not be found.

In the following paragraphs, we describe the PreconditionFor and GeneralizePrecondition subroutines. The subroutines are wrappers around queries to the Dafny verifier, and we show examples of these queries in Figure 3.5.

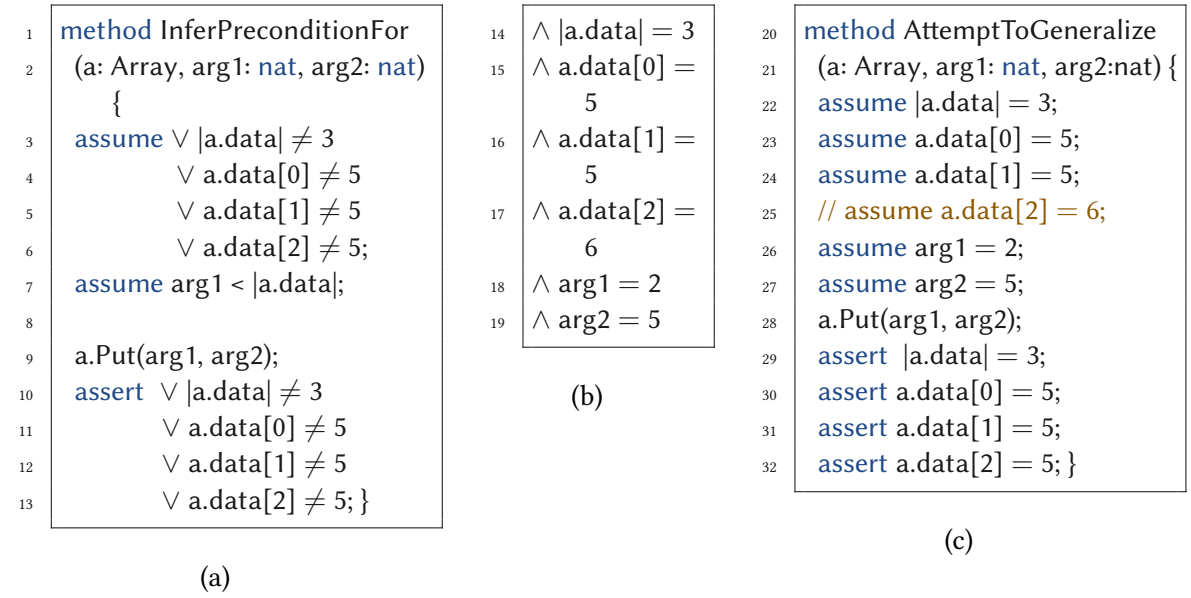


Figure 3.5: Example queries that Metamorph makes as part of the PreconditionFor (a) and GeneralizePrecondition (c) subroutines. Dafny verifies (c) but returns a counterexample (b) when asked to verify (a). GeneralizePrecondition proves that the constraint on line 17 is redundant because (c) verifies with line 25 commented out.

PreconditionFor. Given some method *method* and some state predicate *curr*, the PreconditionFor subroutine identifies the state *prev*≠*curr* and method arguments *args* such that calling method *method* with argument *args* in state *curr* must provably lead to state *prev*. Figure 3.5a shows an example of a query to the Dafny verifier that PreconditionFor makes. In this example, we attempt to synthesize a custom Array object of length 3 with all 3 elements set to 5, and we use method Put, which sets one element of an array to some value. PreconditionFor asks Dafny to prove that there exists no such state in which we could call Put and end up with the array that has the desired properties. The assumption on lines 3–6 ensures that state *prev* is different from *curr*. The assumption on line 7 ensures that the method argument satisfies the preconditions of the Put method, i.e., that the index is within array bounds. Line 9 contains the method call itself. Finally, the assertion on lines 10–13 ensures that after the method call, the object is not in the target state *curr*.

When PreconditionFor asks Dafny to verify the query in Figure 3.5a, Dafny returns a counterexample describing the state *prev* and method arguments *args* for which the assertion might

not hold. Figure 3.5b shows one such counterexample. PreconditionFor also sends a follow-up query to verify that the counterexample is correct, which we don't show but which is identical to that in Figure 3.5c if we uncomment line 25.[3] By only relying on counterexamples Dafny verifies to be valid, we ensure that our approach is sound: each transition between states is proven correct by Dafny, so the resulting solution is also provably correct.

GeneralizePrecondition. Prior to pushing new state predicates on the fringe, Metamorph attempts to simplify each predicate via a call to the GeneralizePrecondition subroutine. This is because these state predicates, when dequeued from the fringe, are then used as postconditions that must be satisfied, so we want them to be as general as possible. GeneralizePrecondition tries to simplify *prev*, the state preceding the method call, by dropping any constraints that are also not present in *curr*, the state following the method call. The intuition is that the complexity of the object is likely to increase following a given method call, so we expect *prev* to have fewer constraints than *curr*. GeneralizePrecondition sends a query to the verifier to check that the same method call still transforms the simplified state into *curr*. For example, the constraint on line 17 in Figure 3.5b is redundant because it constrains the value of the array at a position that is immediately affected by a call to Put. Therefore, GeneralizePrecondition sends a query in Figure 3.5c, where it removes the corresponding constraint from the state predicate *prev* (lines 22–24) and verifies that the method call still puts the object into the desired state *curr* (lines 29–32). This check passes, so we put the new simplified state on the fringe instead of *prev*. Otherwise, if the check were unsuccessful, we would have used *prev* as returned by PreconditionFor.

[3]The reason behind possibly spurious counterexamples is best explained by Dafny Reference Manual (dafny.org/dafny/DafnyRef/DafnyRef.html): "In practice and because of quantifiers, the SMT solver will usually return "unknown" instead of "sat", but will still provide a variable assignment that it couldn't prove that it does not make the formula true. Dafny reports it as a "counter-example" but it might not be a real counter-example, only provide hints about what Dafny knows." This issue is not specific to Dafny and is shared by other verification tools, e.g., Frama-C [94].

3.3 Guiding the Search

In our setting, as is typical, the space of possible candidate solutions is extremely large. To make synthesis tractable, we must guide the search by prioritizing states that are more likely to lead to a solution. In Figure 3.4, this priority is computed on lines 23–24, and it determines the position of a newly discovered state on the fringe. Metamorph computes the priority as a weighted sum of two distances—the known distance from the goal state, and the estimated distance to the initial state. Both distances must be in terms of the number of method calls necessary to transform one state into the other since this number determines the search depth. The nature of the search that Metamorph performs depends, therefore, on the estimate it employs and the coefficients by which it weights the two distances when computing the sum. In particular, if we disregard the estimated part entirely (which is equivalent to setting k_1 to zero in Figure 3.4), Metamorph defaults to breadth-first search. As we demonstrate in Section 4.4, breadth-first search is intractable in this setting, so in practice, we do need a way to estimate distances between states. If we take the estimate into account, then the search becomes an instance of best-first search or A* search depending on the distance metric that Metamorph uses. In this section, we first describe how Metamorph represents states and then discuss two alternative distance metrics and their benefits and limitations.

3.3.1 State Representation

The key to designing a distance metric is to take into account how object states are represented. Dafny extracts object states from counterexamples, which are satisfying assignments to boolean variables (sometimes called *models* or *satisfying interpretations*). As a result, Dafny represents each state as a conjunction of atomic constraints, where each constraint is a predicate applied to some literal values. For example, the constraints `1 in users[0].friends` and `2 in users[0].friends` in Figure 3.2a are two different applications of the same predicate `u1 in users[u2].friends`. We call such predicates *symbolic constraints* to avoid confusion with the meaning that the word “predicate" has

in Dafny. We say that two *atomic constraints* have the same type if they are applications of the same symbolic constraint.

In our running example from Section 3.1, the initial state corresponds to a social network with no users. The target state, in Figure 3.2a, describes a social network with three users forming a clique. Because the target state is provided to Metamorph by the user, it is not guaranteed to be a conjunction of atomic constraints in the same way that all other states are. For example, the constraint on line 3 in Figure 3.2a is not atomic because it encodes multiple constraints at once, namely that each of 0, 1, and 2 are in the set of users. To ensure that the target state representation is consistent with states extracted from counterexamples, Metamorph asks Dafny to prove `assert false` from the target state and then reconstructs the new state predicate from the counterexample. For instance, after we reconstruct the target state in Figure 3.2a from the counterexample, the state will contain three `u in users` constraints and six `u1 in users[u2].friends` constraints that identify three users and three undirected edges.

Since our goal is to estimate distances between a given and initial states, we need to keep track of the values that each constraint we encounter has in the initial state. Metamorph learns the value that each symbolic constraint has in the initial state by sending corresponding queries to Dafny where it asks for proof that a given constraint always holds or never holds in the initial state. If the value of a symbolic constraint is ambiguous, Metamorph sends queries for each associated atomic constraint. We should note that when Metamorph uses piecewise distance, it only takes into account constraints that it identified during the pre-training phase, giving it a minor performance boost. A constraint that is *true* in one state may be *false*, *true*, or *undefined* in another state. As an illustration, consider Figure 3.6a, which summarizes the differences between the initial and target states in our running example. Constraints of the type `u in users` are *false* in the initial state since there are initially no users in the network. Constraints of the type `u1 in users[u2].friends` are, on the other hand, *undefined* in the initial state because `u2` must be in the network for `u1 in users[u2].friends` to be a well-formed expression. In total, there are nine constraints whose value changes between the initial and target states.

3.3.2 Greedy Distance Metric and Best-First Search

An intuitive way to estimate distances between states is to directly compute the number of atomic constraints that hold in one state but not the other. As summarized by Figure 3.6a, this number is nine for the target and initial states in our running example. The advantage of comparing object states this way is that it comes with minimal computational overhead. Similar techniques have also been used in prior work [17]. However, this approach is inherently *greedy* in that it assesses the distance independently of the methods available to Metamorph for modifying the state. Adjusting one or multiple constraints can be achieved by a single method call or it can require multiple calls, meaning that the distance might appear shorter (or longer) than the actual sequence of method calls needed to reach the state. Consequently, Metamorph may be guided towards a path that minimizes constraint differences in the short term but fails to account for possible interactions between constraints and method calls. Because estimating distances between states in this way provides no optimality guarantee, Metamorph performs best-first search when using this approach, which takes less time to explore a state but might lead to a local minimum.

3.3.3 Piecewise Distance Metric and A* Search

To guarantee that Metamorph minimizes the number of method calls in the solution, we propose a novel metric, called *piecewise distance*. Piecewise distance is a provably *admissible heuristic*, meaning it puts a lower bound on the number of methods necessary to transform one state into another. Admissibility allows performing *A* search*, a graph search algorithm that is guaranteed to find the optimal solution. In particular, this means Metamorph performs A* search when it uses piecewise distance and weighs known and estimated distances equally ($k_1 = k_2$ in Figure 3.4).

The intuition behind piecewise distance is to treat each symbolic constraint as a "piece." For each symbolic constraint, Metamorph compares the number of atomic constraints of the same type whose values change in between states with the number of method calls required to enact these changes. Doing so results in a system of inequalities that bounds the number of calls to each of the methods in an API. Metamorph then uses integer programming to minimize the

Symbolic Constraint	Change	Count
u ∈ users	*false* ⟹ *true*	3
u1 ∈ users[u2].friends	*undefined* ⟹ *true*	6

(a) Difference between the initial and the target states.

Method	Symbolic Constraint	Change	Count
AddUser	u ∈ users	*false* ⟹ *true*	1
AddUser	u1 ∈ users[u2].friends	*undefined* ⟹ *false*	∞
RequestResponse	u1 ∈ users[u2].friends	*false* ⟹ *true*	2

(b) Possible effects of individual API method calls.

```
1: Miminize:
2: AddUser + RemoveUser + RequestResponse + FriendRequest
3:
4: Subject to:
5: AddUser * 1 ≥ 3
6: min(AddUser * ∞, RequestResponse * 2) ≥ 6
```

(c) The integer programming query.

Figure 3.6: The process of computing piecewise distance for the target state in Figure 3.2a.

total number of method calls subject to this system of inequalities. Figure 3.6 illustrates how Metamorph computes the piecewise distance between the initial and target states in our social network example from Section 3.1.

Pre-training and Effects of Method Calls. Figure 3.6b summarizes the effects that different methods can have on the values of symbolic constraints from Figure 3.6a. Metamorph learns this information by analyzing the API in a pre-training phase that is independent of any given synthesis problem. In particular, Metamorph learns that a call to AddUser can affect only one constraint of the type u ∈ users (first row on the figure), whereas a query to RequestResponse can change the values of two constraints of the type u1 in users[u2].friends (third row) because each undirected edge is stored twice in the graph. Finally, Metamorph learns that a single call to AddUser can flip arbitrarily many constraints of the type u1 in users[u2].friends from *undefined* to *false*, which we denote in Figure 3.6b with the ∞ symbol. Metamorph saves this information in a file, to be loaded later during synthesis proper.

```
method Pretrain1
 (s:SocialNetwork, u:nat, arg:nat)
{ assume u ∉ s.users;
 assume arg ∉ s.users; // method
     precondition
 s.AddUser(arg);
 assert u ∉ s.users; }
```

(a)

```
method Pretrain2
 (s:SocialNetwork, u1:nat, u2:nat, arg:nat)
{ assume u1 ∉ s.users ∧ u2 ∉ s.users;
 assume arg ∉ s.users; // method
     precondition
 s.AddUser(arg);
 assert u1 ∉ s.users ∨ u2 ∉ s.users;}
```

(b)

Figure 3.7: Queries Metamorph sends to Dafny when pre-training on the Social Network API in Figure 3.1. Dafny verifies (b) but cannot verify (a), so Metamorph learns that a call to AddUser can affect no more than one constraint of type u in users at a time.

At the start of the pre-training phase, Metamorph first collects all relevant symbolic constraints by asking Dafny to prove assert false after executing each method in the API. The counterexamples that Dafny returns include instances of any symbolic constraint that might be implied by any of the methods' postconditions. For each such symbolic constraint, Metamorph asks Dafny to prove that a single method call can never change the value of a constraint from *undefined* to *true* or from *false* to *true*. By doing this, Metamorph infers, for instance, that only a call to AddUser but not any other method can change the value of the constraint u ∈ users from *false* to *true*. Figure 3.7a shows the corresponding query that Metamorph sends to learn this fact. The query fails because Dafny cannot prove that the method does not affect the constraint, so Metamorph assumes that the method can affect it. Metamorph also attempts to put an upper bound on the number of atomic constraints that a method call can affect simultaneously. This is done again by asking Dafny to prove that a method call can never change the values of two, three, or more different atomic constraints of the same type at the same time. For example, Metamorph sends the query in Figure 3.7b to verify that a single call to AddUser cannot add two users to the database at the same time. Because queries to Dafny are costly, we assume that a method that can change the values of three constraints can affect arbitrarily many constraints at once.

Integer Solver Query. Metamorph uses an integer programming solver to lower-bound the total number of method calls necessary to transform one state into another. Here we give a

formal description of this process, followed by the discussion of the example in Figure 3.6c.

Let M be the set of all methods available to Metamorph, and let C be the set of all possible symbolic constraints. Furthermore, let $D(c, \textit{from}, \textit{to})$ be the number of atomic constraints of type $c \in C$ whose value changes from *from* to *to* between the two states under analysis, where $\textit{from}, \textit{to} \in \{\textit{true}, \textit{false}, \textit{undefined}\}$. Figure 3.6a summarizes the values that D takes in our running example. Lastly, let $A(m, c, \textit{from}, \textit{to})$ denote the maximum number of atomic constraints of type $c \in C$ that a single call to method $m \in M$ can change from *from* to *to*. Figure 3.6b summarizes the values that A takes in our running example.

To estimate the minimum number of method calls required for the transformation, we must minimize $\sum_{m \in M} N(m)$, where $N(m)$ denotes the number of calls to method m. To compute this lower bound, we assume that each method call affects as many constraints as it possibly can. The actual number of constraints whose value has changed must then not exceed this maximum. This reasoning provides us with a system of inequalities that $\sum_{m \in M} N(m)$ must be subject to, where there is an inequality for each constraint type $c \in C$ and each possible tuple of distinct values $\textit{from}, \textit{to} \in \{\textit{true}, \textit{false}, \textit{undefined}\}$:

$$\sum_{m \in M} N(m) \cdot A(m, c, \textit{from}, \textit{to}) + \min \left(\sum_{m \in M} N(m) \cdot A(m, c, \textit{from}, \textit{mid}), \sum_{m \in M} N(m) \cdot A(m, c, \textit{mid}, \textit{to}) \right) \geq D(c, \textit{from}, \textit{to})$$

The left-hand side of each inequality computes the maximum number of constraints of type $c \in C$ that could change from *from* to *to*. The change can either happen directly or indirectly via setting a constraint to *mid* first, where $mid \neq \textit{from}$ and $mid \neq \textit{to}$. The right-hand side computes the actual number of constraints whose value has changed.

To provide a concrete example, the first row of Figure 3.6a asserts that the values of three atomic constraints of type u in users must change from *false* to *true*. According to Figure 3.6c, the only method that can enact such a change is AddUser, and each method call can affect only one

constraint at a time. Hence, Metamorph requires that the integer programming solver satisfies the inequality on line 5, thereby ensuring there are at least as many calls to AddUser as there are users added to the database. The second row of Figure 3.6a states that the values of six atomic constraints of type u1 in users[u2].friends must switch from *undefined* to *true*. No single method can directly perform such an action—one must first add the corresponding users to the database before forming the connections. According to Figure 3.6b, the change can be performed in two steps: first, the value of the constraint can be changed from undefined to *false* by calling AddUser, and then the value can be changed again from *false* to *true* by calling RequestResponse. The total number of constraints affected in this way is the minimum between the number that AddUser can affect (arbitrarily many per method call) and the number RequestResponse can affect (two per method call). The second inequality in Figure 3.6c encodes this condition. Solving the query in Figure 3.6c yields the value of six, where the solver reasons that at least three calls to AddUser, and at least three calls to RequestResponse, are necessary to satisfy all inequalities.

Weighted A* search. In practice, because finding a solution is usually more important in synthesis than finding a provably optimal solution, we give more weight to the piecewise distance when choosing which state to expand next ($k_1 = 2k_2$ in Figure 3.4). In this case, piecewise distance still has an optimality guarantee. When $k_1 > k_2$, the algorithm becomes an example of weighted A* search, where the solution is guaranteed to have no more than k_1/k_2 times the minimal number of method calls. We found that increasing this ratio further does not affect Metamorph's performance while lowering it leads to longer running times on some benchmarks. We believe this happens because piecewise distance never underestimates the number of methods in the solution by more than a factor of two in our benchmarks. In our evaluation, we give a further empirical comparison of the piecewise distance metric and the greedy approach outlined above.

3.4 Implementation

We implement Metamorph in C#, which simplifies integration with Dafny because the Dafny verifier itself is also written in C#. The whole implementation fits in about 2500 lines of code. In our code, we rely as much as possible on the functionality that the Dafny API already provides, which ensures the relative ease with which Metamorph can be maintained as new versions of Dafny come out in the future. In this section, we highlight several technical aspects of Metamorph's implementation which we think are particularly informative about the way the tool functions.

3.4.1 Solver Timeouts

As part of the synthesis process, Metamorph makes multiple queries to the Dafny verifier, which, in turn, uses the Z3 SMT [88, 119] solver as a backend. Queries to SMT solvers are not guaranteed to terminate for undecidable theories unless provided with a hard time limit, and so we put an internal time limit on every query we sent to Dafny. Not all queries that Metamorph sends to Dafny are of equal importance, so we use two different time limits for different queries. In particular, if Dafny times out on a query we send during the state exploration process as part of the PreconditionFor and GeneralizePrecondition subroutines described in Section 3.2.1, Metamorph might never explore an object state that lies on the path to the solution, meaning that a single timeout of this kind might prevent Metamorph from solving the synthesis problem. Hence, we use a generous time limit of 100 seconds, which is five times the time limit that Dafny uses by default. By contrast, a timeout during one of the queries we send when computing the piecewise distance metric cannot have such a drastic effect. In the worst case, such a timeout can prevent Metamorph from finding a solution quickly. Consequently, we use a shorter time limit of 30 seconds for queries of this kind.

3.4.2 Recursive Object Synthesis

When we described the algorithm behind Metamorph in Section 3.2, we assumed that the methods Metamorph uses during synthesis only take immutable arguments as input. However, Metamorph can also synthesize programs where objects are passed as input to other method calls. To accomplish this, Metamorph incorporates a recursive step in the algorithm: it loops through all method arguments associated with a newly discovered transition and adds them to the fringe. The conditions under which Metamorph terminates with a solution also change—the tool must now check not only that it synthesized the method sequence that constructs the object specified by the user but also that it synthesized all of the mutable arguments for these methods. One of our benchmarks makes extensive use of this feature, where a binary tree must be constructed by adding two existing trees as subtrees of a new node.

3.5 Evaluation

We evaluate Metamorph in four complementary ways by targeting benchmarks that come from a variety of sources. In Section 3.5.4 we measure how Metamorph's performance scales to synthesizing large instances of widely used data structures, where a solution may require dozens of method calls at a time. We also analyze how different components of Metamorph contribute to its running time, and show that Metamorph outperforms a baseline synthesis approach. In Section 3.5.5, we explore the situations in which the piecewise distance metric has an advantage over the greedy approach. In Section 3.5.6, we demonstrate how Metamorph can be integrated with DTest, Dafny's test generation toolkit, to generate tests for a portion of the AWS Cryptographic Material Providers Library [4], which DTest alone has not been able to cover. Finally, in Section 3.5.7, we conduct a case study in which we use Metamorph to generate executable bytecode for a simple virtual machine, inspired by a real-world Dafny project, demonstrating that the techniques described in this chapter are more broadly applicable in the context of specification-guided synthesis.

In summary, we aim to answer the following research questions in our evaluation:

RQ1 How does Metamorph's performance change as we ask it to synthesize larger objects and thus increase the complexity of the synthesis problem?

RQ2 Does the piecewise distance offer a practical advantage?

RQ3 Can Metamorph increase the test coverage of Dafny's automated testing toolkit?

RQ4 Does Metamorph have applications beyond synthesis of data structure instances?

3.5.1 Benchmarks

We evaluate Metamorph on eight benchmarks, listed below, each of which comes with an API for manipulating a different Dafny object. The first six benchmarks each define a widely used data structure that we use to test Metamorph's ability to synthesize large objects in Sections 3.5.4 and 3.5.5. The last two benchmarks form the basis for the two case studies in Sections 3.5.6 and 3.5.7. We report the time it took to pre-train Metamorph on each benchmark in square brackets after the benchmark's name. This time depends largely on the number of methods in the API and the complexity of the specifications involved:

- *Social Network* [102s]. We describe this benchmark at length in Section 3.1.
- *Freezable Array* [17s]. This benchmark is a modified version of the freezable array data structure from an open-source implementation of the QUIC protocol layer in Dafny [34]. The array can be modified with calls to Put, Freeze, and Unfreeze, where freezing the array's content prevents further modifications until the next call to Unfreeze. The array is frozen and zeroed upon initialization.
- *Binary Tree* [2s]. This benchmark defines a binary tree data structure, where the only method available to Metamorph is the constructor, which takes as arguments an integer value to store in a new node and two potentially null subtrees. The synthesis of programs that construct binary trees presents a special challenge because Metamorph must recursively synthesize objects that are then passed as arguments to construct larger objects.
- *Doubly Linked List* [493s]. This benchmark defines a doubly linked list, with methods

PutFront and PutBack to add elements to the list and methods PopFront and PopBack to remove elements from the list. When specifying the API, we closely follow the methodology used in the paper on the Verus verification language [71].

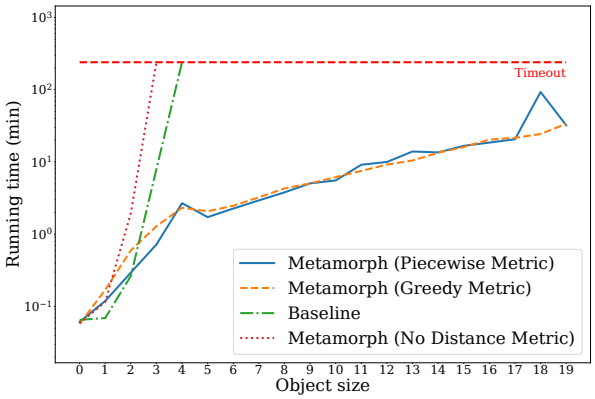
- *Firewall* [36s]. We adapt this benchmark from the paper on Cobalt, a tool for component-based synthesis from effectful libraries [84]. This benchmark describes a graph-like data structure, where one can add and remove devices, specify which devices can exchange data, and designate a central device. The edges of the graph are stored in a single set, making the specification substantially different from that of the graph in the *Social Network* benchmark. When we run Cobalt on the synthesis problems we define for this benchmark to test Metamorph, Cobalt either times out or terminates without a result. We believe this to be the consequence of the limit that Cobalt places on the number of methods in the solution.
- *Queue* [1622s] This benchmark defines a queue data structure, with methods to Enqueue and Dequeue to add and remove elements from the queue, Front and IsEmpty to check the status of the queue, and Rotate to rotate the queue.
- *HKDF* [15s] This benchmark is an unedited implementation of the HMAC-based key derivation function from the AWS Cryptographic Material Providers Library. We discuss this benchmark at length in Section 3.5.6, where we use it to demonstrate Metamorph's integration with the Dafny test generation toolkit.
- *Virtual Machine* [223s] Inspired by the Ethereum Virtual Machine project [21], this benchmark defines a simple virtual machine state, with a stack, memory store, and a program pointer. The state can be modified by calling one of 11 instance methods that each correspond to a bytecode instruction. We discuss this benchmark at greater length in Section 3.5.7.

3.5.2 Correctness

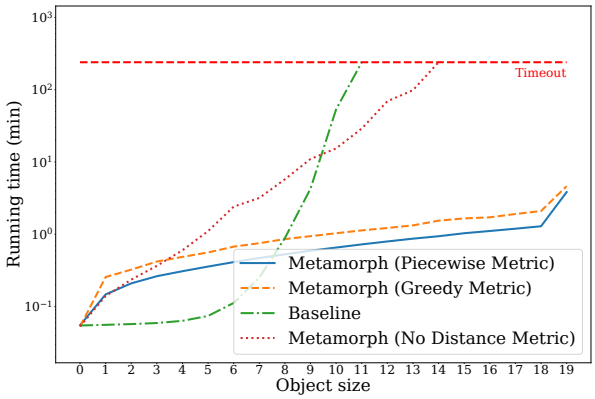
We verify the correctness of our system and experiments in three complementary ways. First, we provide a concrete implementation for every API method in our benchmarks and then use Dafny to verify that the implementation is consistent with the specification. This ensures that the specifications we provide to Metamorph as input are themselves internally consistent and correctly describe the APIs' behavior. Second, we use Dafny to verify programs that Metamorph generates against the synthesis goals. This ensures that every generated program is correct relative to the specification we provide for the corresponding API. Since verification is fully automated, the user of Metamorph can be confident in the correctness of the generated programs. To provide additional, if redundant, assurance, we can also use the compiler shipped with Dafny to translate each generated program to C# and produce an executable, where we test that the program satisfies the synthesis goal at runtime (see Section 3.5.6).

3.5.3 Baseline

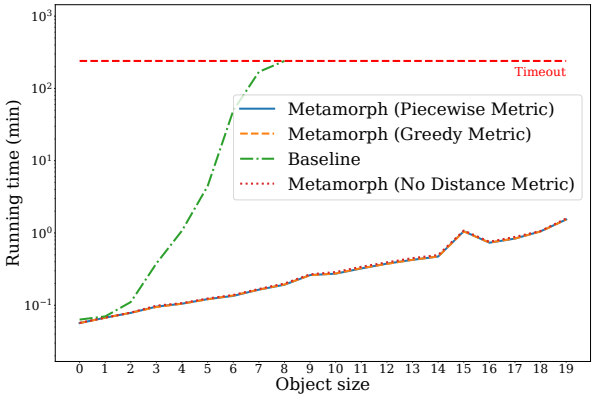
We cannot directly compare Metamorph to prior work on Dafny synthesis [114, 75], since existing solutions focus on the fundamentally different problem of synthesizing method bodies using built-in language constructs, and they cannot compose calls to user-defined methods. Therefore, we developed a naive program synthesis technique to serve as a baseline that relies on Dafny itself to synthesize solutions. For each benchmark, we define a domain-specific language of method call sequences and write an evaluator for this language in Dafny. For each synthesis problem, we then create a method that takes in a program in this domain-specific language, evaluates it, and returns true if and only if the synthesis goal is satisfied. Finally, we prompt the Dafny test generation toolkit [37] to produce a test for this method that leads to the true branch, i.e., a test whose input is the solution to the synthesis problem. Internally, this sends a single query to the SMT solver, which can then use its built-in heuristics to prune the search space, possibly without exploring all possible variable assignments. A further limitation of this baseline approach is that it requires specifying the maximum number of method calls in the program.



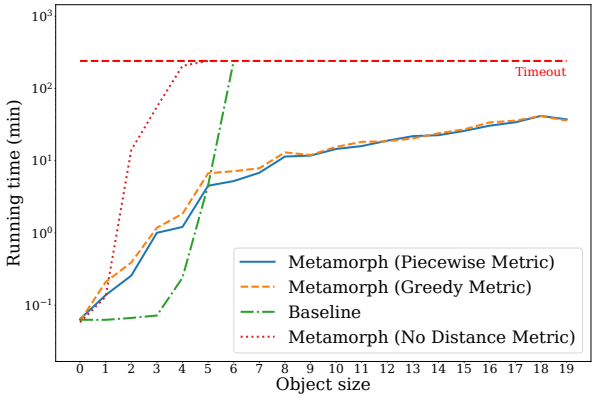
(a) Social Network.



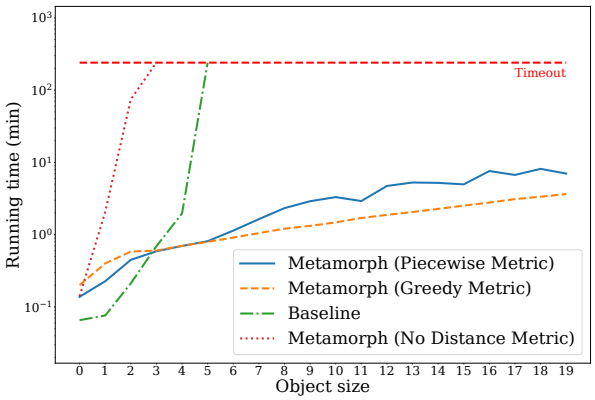
(b) Freezable Array.



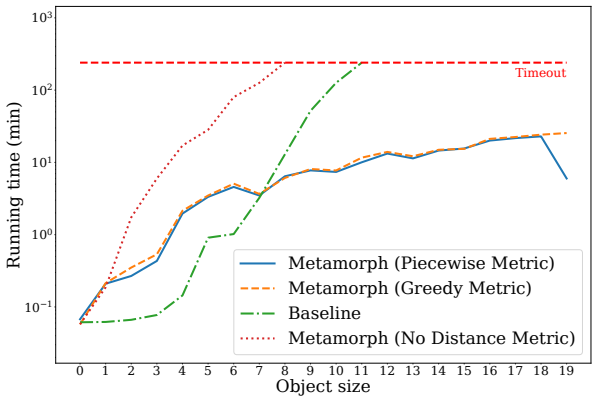
(c) Binary Tree.



(d) Doubly Linked List.



(e) Firewall.



(f) Queue.

Figure 3.8: Running time (log scale) of Metamorph, its modifications, and a baseline approach on six benchmarks.

3.5.4 Synthesizing Large Objects

To evaluate Metamorph's performance, we run it alongside the baseline approach we describe above on 20 synthesis problems of increasing complexity per each of the first six benchmarks.

Our goal is to show that Metamorph scales much better to complex synthesis queries than the baseline. We also conduct an ablation study by running Metamorph on the same set of problems while (i) replacing piecewise distance metric with the greedy approach outlined in Section 3.3, and (ii) disabling the distance estimate functionality entirely, which results in breadth-first search. We perform all experiments on a 384GiB RAM Ubuntu Machine with a 4-hour time limit on each synthesis problem.

For each benchmark, we gradually increase the complexity of the synthesis problems by requiring the creation of larger objects. The n^{th} problem in each benchmark requires the creation of, respectively: (i) a *Social Network* with n users connected to form a single cycle, (ii) a *Freezable Array* of size n that is not frozen and whose contents are integers from 0 to $n-1$, (iii) a balanced *Binary Tree* with n elements that must yield integers from 0 to $n-1$ on in-order traversal, (iv) a *Doubly Linked List* that holds elements from 0 to $n-1$, and, finally, (v) a *Firewall* graph with n devices, where the 0^{th} device acts as a central node and all devices are connected to form a directed cycle, (vi) a queue with n elements. Of all the synthesis problems we define, the one that requires the most method calls to solve is the 20^{th} problem for the *Social Network* benchmark. To solve this problem, Metamorph must create 57 method calls, of which 19 are necessary to add users to the network, 19 are required to send friend requests, and 19 more are needed to accept incoming requests.

Figure 3.8 summarizes the results of this experiment. Each subplot corresponds to one of the six benchmarks. On each subplot, the x-axis specifies the size of the synthesized object, and the y-axis indicates, on a logarithmic scale, the time it took each synthesis technique to produce a solution. The data in Figure 3.8 allows us to answer RQ1: the running time of Metamorph increases approximately linearly with the size of the object it is tasked with synthesizing. The linear running time or Metamorph is in contrast to the baseline solution, whose running time increases exponentially to the point where it quickly reaches our time limit of 4 hours. On the most simple problems, the baseline solution is faster than Metamorph, owing perhaps to the domain-agnostic heuristics built into the Z3 solver that Dafny uses. This suggests that in the

future, the two approaches could even be combined to run in parallel. In the end, however, the running time of Metamorph increases predictably with the size of the object, making it the more stable of the two approaches and more suitable to tackle complex synthesis problems.

Figure 3.8 clearly shows that Metamorph's performance suffers if no distance metric is used, in which case Metamorph's behavior defaults to that of breadth-first search. At the same time, the merits of the piecewise distance metric compared to the greedy metric are not clear. In particular, the greedy metric appears to yield better performance on the *Firewall* benchmark (Figure 3.8e) but worse on the *Freezable Array* benchmark (Figure 3.8b). We discuss the differences between the two metrics in more detail in Section 3.5.5.

Running Time Analysis Figure 3.9 breaks down the percentage of running time that each of Metamorph's components is responsible for. We present this data separately for each benchmark and also show the standard deviation (error bars) across all problems. We exclude the synthesis problems that require one or zero method calls from this analysis because not all components of Metamorph might be active when the solution is immediate. For each benchmark, we report the percentage of running time that Metamorph spent executing the PreconditionFor and GeneralizePrecondition subroutines. We also report the percentage of time Metamorph spent checking if a particular constraint holds in the initial state (see Section 3.3.1). Finally, we report the time Metamorph spent on all other tasks, including on integer programming queries. We do not include the time it took to pre-train Metamorph on each of the benchmarks in this analysis since pre-training is done once per benchmark.

In terms of the distribution of running times, the *BinaryTree*, *Freezable Array*, *Doubly Linked List*, and *Queue* benchmarks each stand out in some way. The *BinaryTree* benchmark only provides Metamorph with one API method to call, which explains why the search over object states (the PreconditionFor subroutine) dominates the running time. In contrast, when synthesizing instances of the *Freezable Array*, Metamorph spends a large portion of the running time analyzing the initial state (see Section 3.3.1). This is because all elements of the array already exist upon ini-

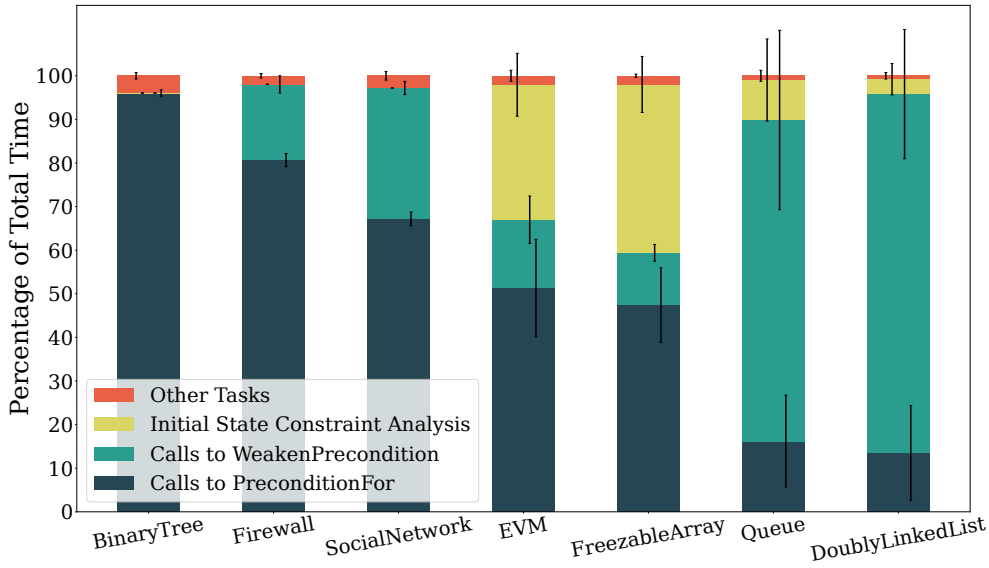


Figure 3.9: Breakdown of Metamorph's running time.

tialization, which makes this data structure different from those in the other benchmarks, where the size of a data structure can vary dynamically.

We also found that the *Freezable Array* benchmark benefits significantly from the GeneralizePrecondition subroutine. In particular, Metamorph fails to synthesize solutions unless this subroutine is enabled (as described in Section 3.2.1). The *Firewall* and *VirtualMachine* benchmarks experience a small slowdown from GeneralizePrecondition because the cost of the procedure outweighs its benefits. Finally, the running time of Metamorph on the *Doubly Linked List* and *Queue* benchmarks is dominated by calls to GeneralizePrecondition. This is mostly due to the complexity of the specifications on these data structures. Notably, these calls to GeneralizePrecondition are not necessary for Metamorph to succeed on these benchmarks. In other words, the GeneralizePrecondition subroutine expands the set of problems that Metamorph can tackle but in some cases, it may also slow down the synthesis.

3.5.5 Comparison of Distance Metrics

Figure 3.8 demonstrates that guiding the search with a distance metric is essential to making synthesis efficient. When we run Metamorph without a distance metric, Metamorph's running

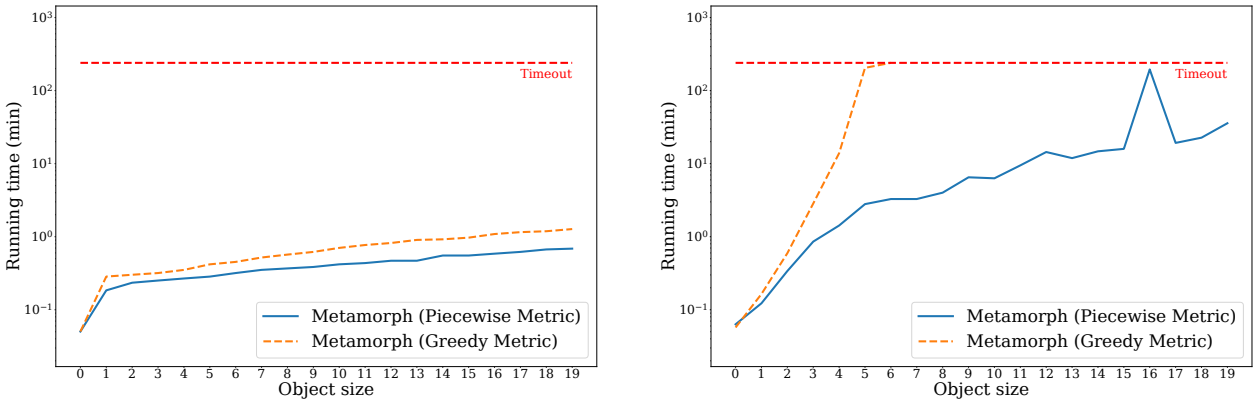
time increases exponentially with the size of the synthesis problem. At the same time, the comparison between the piecewise and greedy distance metrics is mixed. The greedy metric appears to perform better on the *Firewall* benchmark but worse on the *Freezable Array*, while the other benchmarks do not show a noticeable difference.

Thus, we conducted additional experiments to further analyze the differences between the two metrics. We hypothesized that the piecewise distance metric should perform better in the presence of local minima, where the synthesis must attempt a suboptimal path before reaching the solution. To test this hypothesis, we made the following modifications to the *Freezable Array* and *Social Network* benchmarks:

- We added a Fill method to the *Freezable Array* benchmark. The method sets all elements of the array to some user-specified value and then freezes the array.
- We relaxed the object invariant on the *Social Network* benchmark to allow incoming friend requests from non-existing users.

Otherwise, we kept the synthesis problems the same as described above. On these benchmarks, our hypothesis entails the piecewise distance metric allows Metamorph to use the Fill method in the *Freezable Array* benchmark instead of the more simple Put, even though a call to Fill adds additional constraints to the object state that must be undone by a call to *Unfreeze*. Similarly, our hypothesis entails that the relaxed specification of the *Social Network* API would show the benefits of the piecewise distance metric relative to the greedy approach.

Figure 3.10a shows the results of the experiment on the *Freezable Array* benchmark. As expected, Metamorph synthesizes the solution faster when using the piecewise distance metric. Moreover, the solution Metamorph produces when using the greedy distance metric does not make use of the new Fill method. Figure 3.11a shows this suboptimal solution, where the array is updated one element at a time using *Put*. In contrast, the piecewise distance metric guides Metamorph towards the optimal solution, shown in Figure 3.11b, which requires a single call to Fill followed by Unfreeze.



(a) Freezable Array with Fill method

(b) Social Network with Relaxed Object Invariant.

Figure 3.10: Running time (log scale) of Metamorph under different distance metrics.

Figure 3.10b shows the results of the experiment on the *Social Network* benchmark with the weakened object invariant. Regardless of the distance metric used, the running time increases significantly compared to the corresponding experiment in Figure 3.8a. This is because the relaxed object invariant no longer prevents Metamorph from exploring unreachable states, where Metamorph may assume two users are connected before one of them is added to the social network. Whether or not Metamorph explores such unreachable states depends, in part, on the SMT solver that infers possible method arguments during the synthesis process. In the absence of strong specifications, the solver may infer arguments that lead to an unreachable state even when there exists another set of arguments that would have avoided this. When using the piecewise distance metric, Metamorph only explores such “bad" states up to a certain depth before it tries an alternative path of com-

```
1  method Solution()
2    returns (arr:Array)
3    ensures Goal(arr)
4  { arr := new Array(7);
5    arr.Unfreeze();
6    arr.Put(0, 1);
7    arr.Put(1, 1);
8    arr.Put(2, 1);
9    arr.Put(3, 1);
10   arr.Put(4, 1);
11   arr.Put(5, 1);
12   arr.Put(6, 1); }
```

(a)

```
1  method Solution()
2    returns (arr:Array)
3    ensures Goal(arr)
4  { arr := new Array(7);
5    arr.Unfreeze();
6    arr.Fill(1);
7    arr.Unfreeze(); }
```

(b)

Figure 3.11: Two equivalent solutions synthesized when using the piecewise distance (a) and greedy distance (b) metrics.

parable depth, whereas the greedy metric does not provide such a recovery mechanism. For example, it is clear from the spike in the running time of Metamorph on one of the problems in Figure 3.10b that Metamorph spent time exploring such a “bad" state, yet it was still able to generate the correct solution in the end. Overall, the running time is significantly better when using the piecewise distance metric.

These experiments provide the answer to RQ2: while the greedy metric leads to faster running time in some cases, the piecewise distance metric provides improved performance when the API has an underspecified object invariant or the solution to the synthesis problem requires a call to a method that might temporarily add additional constraints to the object's state.

3.5.6 Integration with DTest

A key motivation of Metamorph is automated test generation, where Metamorph may allow synthesizing heap-based test inputs that have specific properties. To evaluate Metamorph in this context, we modified DTest to produce synthesis goals for Metamorph. Recall that DTest works by asking Dafny to prove certain statements unreachable and extracting concrete inputs from counterexamples when possible. We augmented this behavior to return synthesis goals for Metamorph when concrete inputs cannot be extracted. We then designed two experiments in which we ran DTest and Metamorph in tandem to evaluate their interoperability.

In the first experiment, we used DTest and Metamorph to generate tests for a portion of the AWS Cryptographic Material Providers Library [4], a Dafny library that together with AWS Encryption SDK (ESDK) [5] provides an interface between encryption backends and client applications. DTest was originally evaluated on the ESDK, which previously contained the code we discuss below, and DTest was unable to provide complete testing coverage of certain methods that take heap-based objects as input. In particular, DTest could not generate unit tests for the `Extract` and `Expand` methods that are part of the implementation of the Hashed Message Authentication Code (HMAC)–based key derivation function (see [69] for a technical specification). This is because HMAC is a class with instance methods `Init`, `Update`, and `GetResult` that are implemented

externally in C# and are only available to Dafny as specifications. The Extract and Expand methods take HMAC objects as input but DTest does not know how to set these objects to a state that would pass method preconditions.

Metamorph complements DTest by successfully synthesizing sequences of method calls that allocate new HMAC objects and set them to the state in which they can be passed as inputs to Extract and Expand. DTest takes 3 seconds to generate the tests for the HKDF (HMAC-based Key Derivation Function) module, and Metamorph takes 15 seconds to pre-train the piecewise distance function and 36 seconds to synthesize three different instances of the HMAC class that DTest needs.

For a second experiment, we designed a friend recommendation method that interacts with the *Social Network* API defined in Section 3.1. This method takes an arbitrary social network and a user ID as inputs and recommends as a new friend whoever the user shares the largest number of friends with. DTest successfully generates five tests that together provide full statement coverage of the method. Even for this fairly simple example, DTest requires that these instances have as many as four users each. Metamorph successfully synthesizes these instances but, as can be seen from experiments in Section 3.5.4, the baseline algorithm cannot synthesize instances of this size. DTest takes 3 seconds to generate the tests, and Metamorph takes 74 seconds to pre-train the piecewise distance function and 120 seconds to synthesize five different objects. These results allow us to positively answer RQ3: Metamorph enables DTest to generate tests for code it was previously not able to cover.

3.5.7 Synthesizing Virtual Machine Bytecode

In the experiments above, we primarily focused on synthesizing large instances of various data structures. However, the ideas developed in Metamorph apply to the synthesis of Dafny programs more broadly. To demonstrate this, we ran Metamorph on another benchmark that implements a simple virtual machine, inspired by Dafny Ethereum Virtual Machine (EVM) [21], one of the largest open-source Dafny projects.

While Dafny EVM itself is out of reach for Metamorph (or any other synthesis tool that we are aware of), our benchmark closely follows the virtual machine design outlined by the authors of Dafny EVM in a blog post [92]. The virtual machine supports 11 bytecode instructions: NOP (does nothing), PUSH (pushes a value onto the stack), POP (pops a value from the stack), STORE (stores a value from the stack in memory), LOAD (loads a value from memory onto the stack), ADD, SUB, MUL, DIV (each pops two values from the stack, performs the corresponding arithmetic operation, and pushes the result back on the stack), JUMP (increments the program counter by a given number), and JZ (increments the program counter if the first value on the stack is zero). In our implementation, each instruction corresponds to a method, where a method call either performs the associated operation or sets the exit code if the operation cannot be performed. e.g., if there are not enough values on the stack.

Table 3.1: Running time of Metamorph on problems from the Virtual Machine benchmark (median of 3 runs).

Problem	Running Time (seconds)	
	Piecewise	Greedy
Stack Underflow	15	22
Stack Overflow	223	274
Integer Overflow	48	61
Integer Underflow	46	61
Division by Zero	43	61
Invalid Jump	10	13
Invalid Memory Load	6	6
Invalid Memory Store	27	39

To test Metamorph on the Virtual Machine benchmark, we ask it to synthesize sequences of methods (program traces) that lead to one of 8 possible error exit codes. We should note that Metamorph cannot synthesize arbitrary programs for the Virtual Machine and that we chose this task specifically because it is well-suited for backward synthesis, which Metamorph is designed to perform. For example, to reach the DIV_BY_ZERO exit code, Metamorph must push two values on the stack, one of them zero, and then attempt to divide one by the other. Pre-training Metamorph takes 223 seconds (median of 3 runs), and Table 3.1 lists the running time of Metamorph on all 8 synthesis problems. While most exit codes can be reached with just a few instructions, stack overflow requires repeatedly pushing values onto the stack (we set the maximum stack size to 10), meaning that Metamorph can only solve this problem by using a distance metric to guide the search. In this particular case, both the piecewise distance metric and the greedy metric

are sufficient to find the solution, as is evident from Table 3.1. At the same time, the piecewise distance gives consistently better performance (omitting pre-training time), which is due to it only needing to analyze a subset of constraints when computing differences between states (see Section 3.3.1).

We believe this experiment shows that Metamorph could be the first step on the path to a more general-purpose synthesis tool for Dafny, which positively answers RQ4.

3.6 Discussion and Future Work

In this chapter, our focus is on synthesizing sequences of method calls that construct objects as specified by the user. Unlike some of the existing approaches to program synthesis [97, 84], our technique does not allow non-linear control flow and requires that the initial state of the object is concretely specified. By narrowing the scope of the problem in this way, we can synthesize programs that call out to dozens of external methods at a time, something that few synthesis tools are capable of. In addition, by reasoning about the effects of method calls one at a time, we tackle the scalability challenges associated with synthesizing programs in Dafny, an industrial scale verification-aware language that comes with a 1500-line long prelude of definitions and axioms. The key idea behind our approach is the use of counterexamples to guide the synthesis process. In this section, we focus on the implications of this idea, its applicability in other settings, and directions for future work.

Counterexample-guided Synthesis vs Weakest Precondition Reasoning. Counterexamples allow us to concretize the state of the object in between method calls. We represent each state in terms of constraints over the object's ghost fields and infer method arguments that allow us to transform one state into the next one. Contrast this with representing the state as a function of method calls that transform the object into some other state. The latter representation is more expressive because it allows performing weakest precondition reasoning, yet it grows in complexity with the number of method calls involved and requires quantification over the cor-

responding methods' arguments. As we show in Section 3.1, delaying the inference of method arguments until the end of the synthesis process may not be feasible, with excessive quantification being the most likely reason behind the prohibitively long running time. At the same time, the use of counterexamples presents its own challenges. Counterexamples that over-constrain the object state can lead to suboptimal or invalid synthesis paths. Even though our counterexample reduction technique partially mitigates this issue (see Section 3.2), the problem can still occur. For example, in the absence of an object invariant that allows Metamorph to rule out invalid states, Metamorph can start exploring states that could not be synthesized. In the future, we expect that we can combine the use of counterexamples with weakest precondition reasoning, perhaps by using the former as a fall-back technique for when the latter becomes too computationally expensive to tackle excessive quantification.

Piecewise Distance. To guide the synthesis, we rely on the fact that Dafny counterexamples are formulas in conjunctive normal form, which allows us to compute the piecewise distance between object states as we discuss in Sections 3.3 and 3.3.3. This approach need not be specific to Dafny. Regardless of the setting in which one uses an SMT solver, counterexamples that the solver returns are inherently satisfying assignments to boolean variables, which each correspond to an atomic constraint in Dafny. While a lot of post-processing is required to turn a counterexample that the solver returns into a counterexample that Dafny reports to the user [24], the same post-processing procedure can be followed in other settings. Therefore, we believe that the structure of counterexamples that we rely on in this work is not specific to Dafny, and, as a consequence, the piecewise distance metric could be useful in contexts where the state is similarly decomposed into multiple constraints (like in SUSHI [17] and Cypress [60]). We also leave as future work the systematic investigation of combining different distance metrics to guide Metamorph away from exploring invalid object states.

Specification Testing. Throughout this chapter, we assume that the specifications provided by the user exhaustively describe the API's behavior, as is common in verification setting. However,

Metamorph can also be used while such specifications are developed, e.g., to detect a specification weakness by synthesizing programs that lead to a bad state. For instance, by relaxing preconditions on the RemoveUser method and removing the object invariant from our running example, we can successfully prompt Metamorph to synthesize a social network in which edges point to non-existing users, thereby exposing a specification weakness. Therefore, while this would not be Metamorph's main application, we envision that in the future, it could be a part of a larger collection of tools for gradual verification or Dafny programs.

3.7 Related Work

Specification-guided synthesis Many approaches exist for synthesizing programs from logical specifications [60, 97, 66, 2]. From this line of research comes deductive synthesis, which involves the search over proofs of program correctness rather than programs, as exemplified by such tools as SusLik [97] and Cypress [60]. Cypress, in particular, is relevant to our work because it also performs best-first search (but not A* search), where the cost function is also computed by analyzing separate pieces of the specification (heaplets).

Formal specifications and symbolic execution have been used to guide the synthesis of method calls as well [120, 84]. Often such approaches operate on symbolic states or symbolic constraints and leave the inference of concrete method arguments as the last stage of the synthesis process. By abstracting away primitive arguments, one can reduce the search space's size but doing so may not always be practical, as we show in section 3.1.

Component-based synthesis. The ideas presented in this chapter may be characterized as falling under the umbrella of component-based synthesis, where components can be any pieces of code that are combined into executable programs. Component-based synthesis has been a popular direction in recent research [46, 120, 47, 39, 84, 49, 48]. Several approaches to component-based synthesis rely on type information to enumerate candidate solutions [39, 47, 96]. Types alone may not be sufficient to fully describe program behavior, so type-based techniques often

either produce a ranking of possible solutions or rely on executing candidate solutions to check their correctness. However, executing synthesis candidates directly can be prohibitively expensive in the context of web-based APIs due to execution quotas, data privacy concerns, and potentially irreversible side effects. Recent studies address these limitations, either through novel approaches to simulating execution [47] or by using richer type systems that can express more complex program behaviors [96]. Other approaches to component-based synthesis make use of large language models (LLMs) [90, 98, 62]. LLMs can be particularly useful for synthesizing code that interfaces with widely-used libraries, like Python's numpy, where there is a lot of training data and examples available online (see, e.g., [90]). LLMs have also been shown to successfully synthesize Dafny code [85, 106, 18], although not in the context of component-based synthesis.

Dafny Synthesis. Previous work on the synthesis and repair of Dafny programs is mainly focused on generating code that uses built-in language constructs rather than user-provided abstractions [81, 114, 75, 85, 106, 18]. Such techniques are most helpful for developers of Dafny libraries, whereas our work explores how one can automatically make use of existing library APIs. To our knowledge, no prior work has explored the synthesis of method call sequences in the context of Dafny. Of the existing synthesis tools that work with Dafny, Jennisys [75] is particularly relevant to our study, since it makes use of counterexamples to "execute the specification" and find partial solutions to a given synthesis problem. While we use counterexamples in a very different way, the unifying idea remains the same: counterexamples can help concretize a symbolic state in a way that simplifies reasoning about the code.

3.8 Conclusion

In this chapter, we presented a novel approach to specification-guided synthesis of objects that relies on counterexamples to reason about the effects of method calls. We also present two alternative distance metrics that allow us to guide the search over object states to speed up the synthesis. We implemented these ideas in a tool called Metamorph that can generate programs

in the Dafny language such that they construct objects via calls to available API methods. We evaluated the performance of Metamorph on several benchmarks and showed that it can efficiently synthesize large objects. Furthermore, we have successfully integrated Metamorph with DTest and have shown that, in tandem, the two tools can generate tests for code that DTest alone cannot cover.

Chapter 4

DafnyMPI: A Dafny Library for Verifying Message-Passing Concurrent Programs

Recall that one of the central motivations of this dissertation is adding support for concurrency to Dafny. Computational science relies heavily on concurrency to achieve high performance and enable large-scale simulation, modeling, and decision-making, so concurrency support is crucial for extending verification into these domains. Our aim, however, is not simply to add concurrency support, but to do so in a way that avoids major changes to Dafny, preserves its ease of use, and could potentially be replicated in other languages with automated verification support.

To achieve this, we propose DanfyMPI, a novel and scalable approach to verifying programs that use the Message Passing Interface (MPI)[83], a practical realization of process calculi such as CSP[55]. DafnyMPI avoids custom concurrency logics, making it more accessible to software engineers with no prior experience verifying parallel software. DafnyMPI is implemented as an Dafny library, where communication correctness, including deadlock freedom, is guaranteed if users satisfy the preconditions of DanfyMPI library method calls. A Dafny developer who wishes to verify concurrent code needs only to import the library without having to switch to a new toolchain or worrying about cross-compatibility. Combined with what Dafny already offers for verification of sequential programs, the library allows establishing full program correctness

including: absence of deadlocks, safe use of read and write buffers, termination, and functional equivalence with a sequential implementation. (Section 4.1 demonstrates this use of DanfyMPI to reason about a parallelized numerical implementation of the linear convection partial differential equation.)

Of all the properties guaranteed by DafnyMPI, deadlock freedom is the most intricate to establish, particularly because Dafny has no built-in understanding of parallel execution. To prove that the specification of DafnyMPI guarantees deadlock freedom, we introduce a core calculus that closely models the behavior of key MPI primitives: barriers and non-blocking send and receive operations. An ordering property on message tags ensures a process blocked on the message with the lowest tag is eventually unblocked by a matching command from its communication partner. The core calculus also enables a form of rely-guarantee reasoning for relating the data being sent by one process to what is received at the destination. We show that this approach can be extended to support various other MPI operations as well. (Section 4.2 formally defines the core calculus and its operational semantics and outlines the proof.)

To formally establish the remaining correctness properties (termination, safe buffer use, and program equivalence), we rely on tactics that are commonly used by Dafny engineers to verify sequential programs. For program equivalence, we write fully functional specifications and prove that both the sequential and parallel versions conform to them. To guarantee the safe use of buffers, we define custom array datatypes and use ghost state to model read and write locks as well as the contiguity of elements in multidimensional arrays. To optimize verification time, we treat most functions as opaque, allowing users of DafnyMPI to control which definitions are visible to the verifier. (Section 4.3 discusses the design of DafnyMPI and techniques for establishing full program correctness.)

To demonstrate the practical utility of DafnyMPI, we used it to verify numerical solutions for three partial differential equations: Linear Convection, Heat Diffusion, and Poisson. We discuss the proof engineering effort required to verify each benchmark and show that the increased proof complexity of the parallel versions is mirrored by a higher percentage of ghost (non-executable)

code. Finally, we compile the verified Dafny code to Python and demonstrate that the resulting programs exhibit the expected runtime behavior under parallel execution—specifically, that increasing the number of processes leads to faster computation. (Section 4.4 describes these experiments in more detail.)

In summary, the main contributions of this chapter are as follows:

- We introduce a technique for verifying concurrent MPI programs and establishing deadlock freedom, safe buffer use, termination, and equivalence with a sequential implementation.
- We implemented this technique as DafnyMPI, a library for the Dafny programming language.
- We verified three MPI programs to evaluate the practical benefits of the approach. We show that the parallel implementations do exhibit better runtime performance than their sequential counterparts.
- We release DafnyMPI alongside all the associated benchmark as an artifact, available at https://doi.org/10.5281/zenodo.17102521

By verifying MPI programs in a language designed for sequential reasoning, we aim to push the boundaries of scalable verification and support the development of reliable software.

4.1 Overview

To illustrate the use of DafnyMPI, we consider computing a solution to the one-dimensional linear convection equation, which models a wave moving at a constant speed. Later in Section 4.4, we show a figure that illustrates this phenomenon. More formally, the equation describes a quantity $u(x,t)$ that varies in space and time, and it states that the change over time ($\frac{\partial u}{\partial t}$) is proportional to the slope ($\frac{\partial u}{\partial x}$):

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{4.1}$$

To solve the equation, one must be able to find the value of u at a given time step t and point x given the initial state u_0 [105]. A numerical solution must approximate the partial derivatives in terms of small discrete steps in time (Δt) and space (Δx). One approach, known as the *upwind* scheme, is to use the forward difference to approximate the temporal derivative and the backward difference to approximate the spatial derivative:

$$\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} + c\frac{u(x, t) - u(x - \Delta x, t)}{\Delta x} = 0 \tag{4.2}$$

Solving Equation 4.2 for $u(x, t + \Delta t)$, we get:

$$u(x, t + \Delta t) = u(x, t) - c\frac{\Delta t}{\Delta x}(u(x, t) - u(x - \Delta x, t)) \tag{4.3}$$

In other words, the numerical solution iteratively computes $u(x, t + \Delta t)$ for all points in the spatial domain of u. Note that this computation is local — the value at a point x at time $t + \Delta t$ only depends on the values at points x and $x - \Delta x$ at time t. The locality of the computation means it can be easily parallelized across several processes, where each process is responsible for computing a portion of the solution. However, the processes must also exchange information at each time step to compute the boundary values: if point $x - \Delta x$ is outside its allotted portion of its spatial domain, the process responsible for computing $u(x, t + \Delta t)$ must receive the value $u(x - \Delta x, t)$ from its neighbor. The Message Passing Interface is ideal for facilitating such structured communication.

4.1.1 Solving PDEs with the Message Passing Interface

The Python code on the left side of Figure 4.1 implements Equation 4.3 using MPI. As is standard in MPI, the code is executed by launching it simultaneously across many CPU cores on a machine. Each process begins by obtaining its rank and the total number of running processes, also known as size (lines 1–3). Next, each process computes the portion of the initial state it is responsible for (lines 5–7). The initial conditions can vary; for simplicity, we don't further specify the cor-

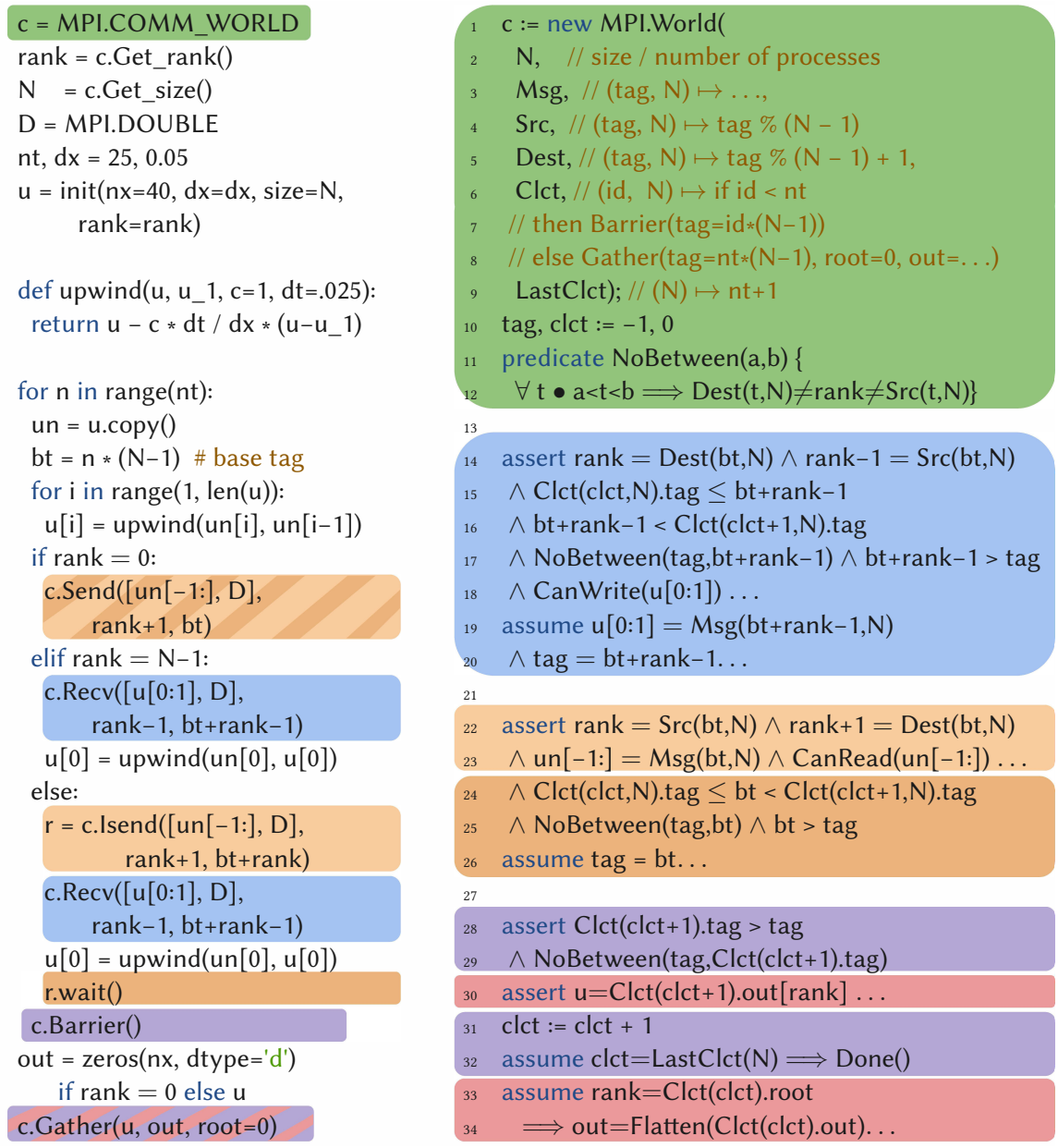


Figure 4.1: Example Python MPI program (left) and corresponding Dafny proof obligations (right, simplified)

responding function `init` here. The processes with adjacent ranks are responsible for adjacent portions of the spatial domain. We will say that one process is to the left of the other process if it is responsible for the portion of the spatial domain immediately to the left of the other process on the x-axis.

Each process then simulates the equation on `u` for `nt` time steps (lines 12–31). At each time

step, the process updates its portion of u according to Equation 4.3, which is implemented by function upwind in the code (lines 9–10). Each process performs the computation locally for all points it is responsible for (lines 15–16) except for the leftmost boundary value (u[0] in the code), which depends on the information from the neighboring process. The leftmost process 0 only needs to communicate the boundary value to the process on the right (lines 18–19), while the rightmost process $N - 1$ only needs to receive one from the process on the left. (lines 21–22). Every other process both receives a boundary value from the process on the left (lines 27–28) and sends one to the process on the right (lines 25–26). Note that for efficiency, the code uses the *immediate send* command (Isend), which allows the two operations to run concurrently — the immediate send operation is initialized before the receive but is not guaranteed to complete until after the wait call on line 30. Notice also that each message is tagged with a unique integer derived from the time step and rank (final argument in calls to Send, Recv, and Isend). We use these tags as a key part of the verification, as discussed below. The other arguments shared by all send and receive operations are the rank of the destination or source process, respectively, and the buffer—passed by reference—from which the message is read (for sends) or to which it is written (for receives).

A round of communication completes with each process calling Barrier (line 31), which syncs all processes before the next round of communication. After nt iterations, the Gather command collects all segments of u into a single array out managed by process 0 (lines 32–34). Process 0 may then handle the outputs as needed, e.g., by saving it to disk or displaying a plot.

4.1.2 DafnyMPI

The goal of this chapter is to develop a methodology for proving that MPI programs such as the one in Figure 4.1 are correct, by which we mean that they must terminate successfully with a result that is either identical to that of a matching sequential implementation or equivalent up to rounding differences caused by the non-associative nature of floating-point arithmetic and similar machine-level approximations. We consider sequential implementations to be the reference

implementations since they are typically easier to write and are therefore less likely to contain bugs. They are also simpler to debug. Full program correctness, then, is contingent on the following properties:

- *Deadlock Freedom.* Any process that is waiting for a send (receive) operation to complete will eventually be unblocked by a matching receive (send). Similarly, any process waiting on a barrier will eventually pass that barrier.

- *Safe use of read and write buffers.* An array segment used by an MPI operation must not be modified by any other operation or by the process itself until the operation is completed. In the case of receive operations, the buffer must remain entirely inaccessible, as its content is undefined until the message is received.

- *Termination.* Every process will eventually terminate. In other words, infinite loops or endless recursion are not possible.

- *Functional Equivalence.* The MPI program must compute the same result as a corresponding sequential implementation. In particular, each process must compute the portion of the final solution corresponding to its assigned region of the spatial domain. In our benchmarks, all such portions are then merged and returned by the root process (rank zero).

Dafny already supports reasoning about functional equivalence, termination, and the absence of runtime errors. Thus, given a Python MPI program, we manually translate it into an equivalent Dafny program, which we then verify. Following verification, this Dafny program can then be compiled back into Python for execution.

The key challenge with this approach is that Dafny is a single-threaded language, with no built-in notion of parallelism or concurrency. Thus, DafnyMPI is designed with the meta-property that, if the proof obligations of its MPI primitives are met, then the parallel execution of the program is guaranteed to satisfy the properties listed above.

The right side of Figure 4.1 shows a selection of the DafnyMPI proof obligations for our example, written as Dafny code. Here and in Section 4.2, we focus on deadlock freedom, as that is

the most intricate property to prove. Section 4.3 describes the proofs for the other properties. In the figure, the correspondence between source code and proof obligations is shown by color. In some cases, a program statement corresponds to multiple proof obligations, which is shown with appropriately colored stripes.

At the start of the program, corresponding to line 1 on the left, the code creates a new MPI.World instance that must include programmer-supplied specifications of the overall communication pattern of the program. Specifically, the programmer provides the *size* N, the total number of processes—which, as we do here, can be left unconstrained to show the proof holds for any number of processes greater than one—and a list of functions describing the program's use of point-to-point messages and collective operations. The first three functions, Msg, Src, and Dest, describe point-to-point messages, mapping message tags and the size to the contents of the message, the message source, and the message destination, respectively. For our example, we omit the message contents for conciseness, and the source and destination are computed from the tag and the size. The next two functions, Clct and LastClct, describe the program's collective operations, including barriers. LastClct specifies the total number of collective operations in the program, which, in the general case, is a function of the number of processes. In our example, there are always exactly nt barriers—one per loop iteration (lines 12–31)—and one Gather call on line 34, so LastClct is constant and is set to nt+1. The Clct function maps each collective operation ID to its description, including the operation type (e.g., Barrier or Gather) and the lowest tag—across all processes—of any point-to-point message that immediately follows it. This allows us to order collective operations, that otherwise cannot be assigned tags, relative to point-to-point messages. Some collective operations require additional fields, e.g., the description of Gather includes the receiving process (root) and the output value (out). Finally, the variable tag is initialized to -1 to track the tag of the last completed message, and clct is initialized to 0 to track the ID of the next upcoming collective operation. For convenience below, we also introduce helper predicate NoBetween(a,b), which holds if and only if there is no tag between a and b for which the current process is the sender or the receiver.

Next, the code corresponding to the two Recv calls on lines 21–22 and 27–28 asserts that, in order, the source and destination of the message match the tag, the tag falls between the previous and the next collective operations (in our case, barriers), no message was skipped, the current tag is greater than the one previously used, and the destination buffer (u[0:1]) can be written to. Then the snippet assumes (the rely part of rely–guarantee reasoning) that recv returns the message Msg(bt+rank−1,N) corresponding to the tag in the specification. Finally, the process's tag counter is updated to the message's tag.

The code corresponding to the Send call on lines 18–19 similarly begins by asserting that the source, destination, and contents of the message match the tag (the guarantee part of rely–guarantee reasoning), and that the source buffer is free to be read from. Next, the code enforces the same tag ordering properties as for the Recv call. After the send completes, the process's tag counter is incremented. Whenever a send is non-blocking, such as on lines (lines 25–30), these different conditions are enforced at different points along the program's execution. The ordering properties on the tag are enforced at the corresponding wait call (line 30) that completes the send, while the rest of the conditions are checked immediately when the point-to-point communication is initialized (lines 25–26). Additional pre- and postconditions, not shown in the figure, ensure that the process does not send a duplicate message and that the send buffer is not written to between the calls to ISend and wait.

To understand why the conditions listed above prevent deadlock, suppose, for a contradiction, that deadlock does occur and that, among all the blocked processes, process P is blocked on an MPI call with the lowest tag. The matching process for that message must not have initiated the communication with P on its own end yet, since otherwise P would eventually unblock. Therefore, the matching process must not have reached the point in the program at which it initiates communication with P. At the same time, there is a contradiction, because the matching process cannot be blocked at this point in the program, since then it would be the process blocked on the message with the lowest tag. Therefore, the matching process must be running and so there is no deadlock. (Section 4.2 formalizes this argument.)

Finally, the code corresponding to the Barrier call on line 31 asserts the barrier is called after any message that must precede it, before any message that must follow it, and without skipping any messages. If the ID of the next collective operation, which is incremented here, is equal to LastClct(N), Dafny can assume Done(), a special condition that signifies that the program finished execution. The user of the DafnyMPI library must have a postcondition on the main method that checks that Done() is true. In our example, this condition is only fulfilled after the call to Gather on line 34. This call has additional pre- and post-conditions attached to it that ensure the inputs conform to the specification provided by the user above. Among other things, the code asserts that u holds part of the output corresponding to the process's rank and guarantees that the zeroth process receives the output, flattened into a 1D array, into the variable out.

Together, these conditions guarantee deadlock freedom and allow the user to prove that the variable out will, for the zeroth process, contain the solution to the PDE after nt time steps. The next section formalizes this approach for a core calculus where we prove that these conditions are sufficient.

4.2 Proving Deadlock Freedom

In this section, we formalize DafnyMPI on a core calculus and prove that for a given program, if DafnyMPI's requirements are satisfied, then that program will be deadlock-free when run in parallel. Our core calculus adds several primitives from the MPI 1.0 standard [83] to a language of commands. As far as we are aware, later MPI standards maintain the semantics of these primitives, so our approach would still apply. Moreover, in our experience, most MPI programs are written in a way that does not require the more advanced features of later standards, such as MPI 2.0 or MPI 3.0, which introduce additional features such as one-sided communication and dynamic process management.

DafnyMPI directly encodes the following four MPI functions:

- `MPI_IRECV` (*immediate receive* or *non-blocking receive*) requests to receive a message with a

given tag from a given source. The operation returns immediately, and the message may then be received in the background. The operation returns an `MPI_REQUEST` object that can be waited on to ensure that the message has been received.

- `MPI_ISEND` (*immediate send* or *non-blocking send*) requests to send a message with a given tag to a given destination. The operation returns immediately with an `MPI_REQUEST` object that can be waited on to ensure the send has been completed. Note that completion is local in this context: it does not guarantee that the message is received at the destination, only that the sender can reuse the buffers associated with the operation.

- `MPI_WAIT` (*wait*), called on an `MPI_REQUEST` object, blocks a process until the corresponding communication has been completed.

- `MPI_BARRIER` (*barrier*) blocks until all processes also reach a barrier.

In Section 4.3, we discuss how we can extend the core calculus to support MPI functions that are available in DafnyMPI, using syntactic sugar or as simple variations of the initial four.

- `MPI_RECV` (*blocking receive*) can be encoded as an `MPI_IRECV` followed immediately by the relevant `MPI_WAIT`.

- `MPI_SEND` (*blocking send*) can be encoded similarly with `MPI_ISEND` and `MPI_WAIT`.

- `MPI_GATHER` must be called by all processes, after which the data associated with each process is transferred to the designated root process and stored in a single array. We encode this operation as a decorated barrier call, requiring that all processes are synced before exchanging information in this way.

- `MPI_ALLREDUCE` must be called by all processes, after which the data submitted by each process is folded using a binary operation such as summation or logical and. The result is then communicated back to all processes. This operation is encoded as a decorated barrier similarly to `MPI_GATHER`.

DafnyMPI does not currently support MPI *communicators*, which group processes together so that, e.g., only a group needs to pass a barrier, or *wildcards*, which allow one process to exchange a message with an arbitrary process associated with the same communicator. We leave support for these features to future work. Instead, DafnyMPI programs use the single default `MPI_COMM_WORLD` communicator on all operations.

4.2.1 The Core Calculus

Figure 4.2 shows the syntax of our core calculus, which is stratified into expressions and commands. As our proof focuses on interprocess communication, expressions e are limited to integers n, variables x, comparisons e = e, and basic arithmetic operations e + e, e * e, e **div** e, -e. Because there is no shared memory, each process has its own variable environment, and there is no way for one process to affect the evaluation of an expression by another process. However, an expression may not read a variable that is being written by a background receive call (see Section 4.2.3).

Commands c include four communication primitives corresponding to the MPI functions listed above: **irecv** e x, non-blocking receive of a message with tag e, where the message is written to x; **isend** e x, non-blocking send of a message with tag e, where the message is read from x; **wait** e, waiting for completion of the send or receive with tag e; and **barrier**, which blocks until all processes reach a barrier. In all cases, tags are integers. The remaining commands—**skip**, **if**, **while**, **set**, and sequencing—have the standard semantics, where zero is considered false and non-zero integers are considered true.

Our semantics models the state of a single process as a tuple $\langle c, \sigma, t, b \rangle$, where c is the command to be executed, σ is the variable environment mapping variables to integers, t is the largest tag previously used in a **wait** call, and b is the number of barriers the process has passed so far. Then the global program state S, encompassing all processes, is a tuple

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e = e \mid e + e \mid -e \\
 & \mid & e * e \mid e \ \mathbf{div}\ e \\
c & ::= & \mathbf{irecv}\ e\ x \mid \mathbf{isend}\ e\ x \\
 & \mid & \mathbf{wait}\ e \mid \mathbf{barrier} \\
 & \mid & \mathbf{skip} \mid \mathbf{if}\ e\ c\ c \\
 & \mid & \mathbf{while}\ e\ c \mid \mathbf{set}\ x\ e \mid c\ ;\ c
\end{array}
$$

Figure 4.2: Core calculus syntax.

$(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)$, where $\mathcal{P}$ is a set of process states, and $\mathcal{B}_r$, $\mathcal{B}_s$, and $\mathcal{B}_m$ are the global receive, send, and message buffers, respectively. Throughout this section, we will also use $N = |\mathcal{P}|$ to refer to the number of processes.

The receive and send buffers are maps from message tags to variable names, indicating the destination (for receives) or source (for sends) of the message within the process environment. The message buffer is a map from message tags to message payload, which is the value that would eventually be copied from the source variable in the sender's environment to the destination variable in the receiver's environment. This intermediate payload buffer is necessary because MPI allows a send operation to complete before the corresponding receive is called, in which case the message contents must be buffered internally, while the sender is allowed to reuse the source variable. It is sufficient to model the buffers globally because the programmer is required to provide functions that uniquely identify the sender and receiver for each message tag as part of the program's specification. Section 4.2.2 goes through a concrete example of how the buffers are used.

Our operational semantics is non-deterministic and may take a step in any process that is not blocked. Thus, we treat the running processes as a set and use the set notation for convenience. In particular, we write $\langle c, \sigma, t, b\rangle \cup \mathcal{P}$ (omitting curly braces around the singleton set) to select one process $\langle c, \sigma, t, b\rangle$ with the remaining processes as $\mathcal{P}$, and we write $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots$ to non-deterministically split the set of processes into subsets. We store the process rank by binding an integer to `rank` in σ. As a shorthand, we use a subscript on the process state to indicate rank, i.e., $\langle c, \sigma, t, b\rangle_i$ indicates a state where $\sigma(\texttt{rank}) = i$. We omit the subscript when the rank is irrelevant.

A state reduction is a sequence of states $S_0, S_1, \ldots, S_k$ such that each preceding state transitions to the next one according to the operational semantics (Section 4.2.2). As another shorthand, we sometimes refer to the state components of the ith process during the kth reduction step by subscripting with i, k, e.g., $\sigma_{2,0}$ refers to the variable environment of the process with rank 2 in the initial state S_0.

DafnyMPI programs are SIMD, with the same command c executed on all nodes. The initial

state of the execution of command c on N nodes is given by:

$$S_0 = (\bigcup_{0 \le i < N} \langle c;\ \textbf{barrier}, \{\texttt{rank} \mapsto i, \texttt{size} \mapsto N\}, -1, 0\rangle, \emptyset, \emptyset, \emptyset)$$

Notice the processes are initialized with variables `rank` and `size` to let the process know its rank and the total number N of processes running. Each process is also initialized with the exact same sequence of commands to execute, where the last command must be a **barrier**.

We verify a given program with respect to a user-provided specification, where, as in Section 4.1, the user must describe the overall communication pattern of the program ahead of time. Specifically, the user provides five functions, which we will refer to throughout this section and which together describe all MPI operations:

- Sender and Receiver are total functions that map a message tag and the number of processes to the rank of the process that sends or receives a message with that tag. In our running example in Figure 4.1, these correspond to functions Src and Dest, respectively.
- Message (Msg in Figure 4.1) is a total function mapping a message tag and the number of processes to the message payload, which in our core calculus semantics may only be an integer.
- BarrierCount (LastClct in Figure 4.1) maps the number of processes to the total number of barriers a process must pass before terminating.
- Barrier is a total function that maps the rank of a barrier and the number of processes to a message tag, indicating that any message with a smaller tag must be sent and received before the process reaches the barrier. This function roughly corresponds to function Clct in Figure 4.1, although DafnyMPI supports collective operations not explicitly modeled by the core calculus, as we describe in Section 4.3.

Next, we present the operational semantics of our core calculus and the guarantees that are provided by Dafny.

4.2.2 Operational Semantics

Figure 4.3 gives our small-step operational semantics, divided into two groups: (a) local rules that touch a single process, and (b) global rules that also manipulate shared buffers or several processes at once. Our semantics also includes big-step expression evaluation, $\langle e, \sigma \rangle \Downarrow v$, meaning evaluating expression e in variable environment σ yields value v. As expression evaluation is entirely standard, we omit these rules.

The local rules, shown in Figure 4.3a, define the single stepping relation $\rightsquigarrow$ on process states $\langle c, \sigma, t, b \rangle$ meaning taking one step in the process whose state is on the left yields the state on the right. IfTrue and IfFalse cover the two cases for conditionals, and While expands one step of the iteration. Finally, Set updates the variable environment appropriately, and SeqSkip removes a **skip** at the start of a sequence. A process whose command is a lone **skip** is considered to have terminated, and thus there is no rule to evaluate it further.

The global rules, shown in Figure 4.3b, define the single stepping relation $\longrightarrow$ on program states $(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)$. Importantly, these rules omit several premises that would typically be checked dynamically but instead are guaranteed to hold by the Dafny verifier. For example, the Send rule does not verify that the sender's rank matches the tag's designated sender, because this is checked as part of using DafnyMPI. We formalize these preconditions as *axioms* in Section 4.2.3, below.

The first two reduction rules are administrative: Proc lifts a single-process reduction to the reduction of the entire program state, and SeqStep takes a step in the first command of a sequence. Notice that both rules use set notation to pick a process non-deterministically.

Most of the rest of the rules model send operations, and the main challenge is send's non-deterministic behavior. Depending on the implementation and message size, the MPI runtime may copy the payload into an internal buffer, allowing the sender's subsequent **wait** to complete immediately. Alternatively, the process may block until a matching receive is called. Our operational semantics aims to mimic both of these behaviors.

We use the example in Figure 4.4 to illustrate the communication rules as we describe them.

$$\frac{\langle e, \sigma\rangle \Downarrow v \quad v \neq 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_1, \sigma, t, b\rangle}\ \textsc{IfTrue} \qquad \frac{\langle e, \sigma\rangle \Downarrow 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_2, \sigma, t, b\rangle}\ \textsc{IfFalse}$$

$$\frac{}{\langle \textbf{while } e\ c, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{if } e\ (c;\ \textbf{while } e\ c)\ \textbf{skip}, \sigma, t, b\rangle}\ \textsc{While}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle \textbf{set } x\ e, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{skip}, \sigma[x \mapsto v], t, b\rangle}\ \textsc{Set} \qquad \frac{}{\langle \textbf{skip};\ c, \sigma, t, b\rangle \rightsquigarrow \langle c, \sigma, t, b\rangle}\ \textsc{SeqSkip}$$

(a) Single-Process Operational Semantics Rules.

$$\frac{\langle c, \sigma, t, b\rangle \rightsquigarrow \langle c', \sigma', t', b'\rangle}{(\langle c, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Proc}$$

$$\frac{(\langle c_1, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}{(\langle c_1; c_2, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c'; c_2, \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}\ \textsc{SeqStep}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{isend } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s[t' \mapsto x], \mathcal{B}_m)}\ \textsc{Send}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{irecv } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r[t' \mapsto x], \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Recv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_r \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma[\mathcal{B}_r(t') \mapsto \mathcal{B}_m(t')], t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r \setminus \{t'\}, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{WaitRecv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s \setminus \{t'\}, \mathcal{B}_m)}\ \textsc{WaitSend}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad \textsc{Receiver}(t', N) = i \quad t' \in \mathcal{B}_s \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_{\textsc{Sender}(t', N)}\rangle \Downarrow v}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferOnWait}$$

$$\frac{t' \in \mathcal{B}_s \quad i = \textsc{Sender}(t', N) \quad c_i \neq \textbf{wait } e\textbf{; } c' \quad c_i \neq \textbf{barrier; } c' \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_i\rangle \Downarrow v}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferNoWait}$$

$$\frac{i = \textsc{Sender}(t', N) \quad j = \textsc{Receiver}(t', N) \quad t_i \geq t' \quad t_j \geq t' \quad t' \in \mathcal{B}_m}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m \setminus \{t'\})}\ \textsc{FreeBuffer}$$

$$\frac{}{(\bigcup_{0 \leq i < N} \langle \textbf{barrier; } c, \sigma, t, b\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\bigcup_{0 \leq i < N} \langle \textbf{skip; } c, \sigma, t, b+1\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Barrier}$$

(b) Multi-Process Operational Semantics Rules.

Figure 4.3: Core calculus reduction rules.

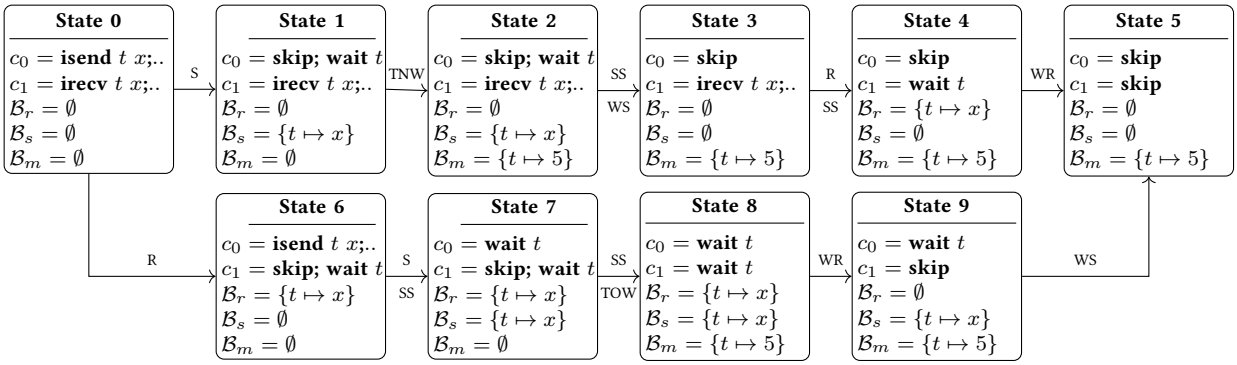


Figure 4.4: Two partial reductions for the following program, executed by 2 processes: **set** x 5**; if** `rank` $= 0$ **{ isend** t x **}{ irecv** t x **}; wait** t. Edges are labeled with abbreviations of reduction rules (SS sands for SeqSkip, etc.). The reduction rule above the line is taken first.

The figure shows two of the possible partial reduction sequences of a simple program. In the initial state (State 0 on the figure), the process with rank 0 is poised to initiate a non-blocking send for a message with tag t, and the process with rank 1 is poised to initiate a non-blocking receive for the same message. Either command may fire first, leading to State 1 or State 6, respectively.

Rule Send adds a request to send x as a message under the appropriate tag t'. The request is registered by placing x in the send buffer $\mathcal{B}_s$. Similarly, Recv adds a request to receive x with tag t'. As described above, these operations are non-blocking. In our example, we label the state transitions with S (send) and R (receive), and we see the appropriate updates to $\mathcal{B}_s$ and $\mathcal{B}_r$, respectively.

Continuing along the top of the figure, the transition from State 1 to State 2 illustrates TransferNoWait, which, given a sender process that is not waiting or at a barrier (recall c_i indicates the command of the ith process), copies the contents of the to-be-sent message from x to $\mathcal{B}_m$ with the appropriate tag t'. In the example, the message payload is 5, and thus $\mathcal{B}_m$ is updated to map t to 5. This behavior captures the case when the MPI implementation copies a message payload into a buffer, allowing the sender process to unblock before the matching receive is even initiated.

Once the message is copied into the message payload buffer, the process with rank 0 is now ready to complete the send, as captured by the transition from State 2 to State 3 via WaitSend.

The rule checks that the message with tag t' has been placed in $\mathcal{B}_m$ and then removes the tag from the set of pending sends. In our example, we see that this rule can fire because of the previous execution of TransferNoWait, and therefore tag t is removed from $\mathcal{B}_s$.

The transition from State 3 to State 4 uses Recv to add a request to receive the message with tag t, following which the WaitRecv can fire to move from State 4 to State 5. Analogously to WaitSend, WaitRecv can execute when a message with the given tag is in the message buffer, and it updates the variable to be written with the message payload, removing the tag from $\mathcal{B}_r$ in the process. Although it is not shown in the figure, x has been updated in c_1 to contain 5.

In contrast, the bottom row illustrates the case when an MPI send blocks until the matching receive is executed. Specifically, after the Send rule is used to transition from State 6 to State 7, the process with rank 0 cannot proceed any further. The WaitSend rule cannot fire because the message payload buffer $\mathcal{B}_m$ is empty and also the TransferNoWait rule cannot fire because the process is already blocked on a wait. The only rule that can unblock process with rank 0 is, therefore, TransferOnWait, which can fire whenever the receiving process with rank 1 reaches the corresponding wait call. This is illustrated with a transition from State 7 to State 8, where the process with rank 1 first gets to the **wait** call via the SeqStep rule and then initiates the transfer via the TransferOnWait rule. Subsequently, WaitRecv transitions the system from State 8 to State 9, and WaitSend completes the communication by advancing to State 5.

Note that in State 5, the final state in Figure 4.4, the message buffer $\mathcal{B}_m$ still contains the message tagged t. Once the communication associated with the tag has been completed, it may be removed from $\mathcal{B}_m$ at any point according to the FreeBuffer rule. The rule mimics the expected behavior of MPI runtime in that any memory used internally by the MPI should eventually be freed. The exact timing is implementation-dependent and irrelevant to our proof.

The last rule of our operational semantics, Barrier, specifies that all processes must execute a barrier simultaneously. Note that we adopt a slight abuse of notation by allowing the Barrier rule to apply to the final barrier, even though the rule formally presupposes the existence of a subsequent command.

4.2.3 Axioms

Our core calculus is designed to represent programs that a user of the DafnyMPI library could write. For Dafny to verify these programs, the user must prove a number of properties. These include safety properties Dafny enforces by default, such as loop termination, absence of division by zero, and absence of references to undeclared variables. Several other properties are MPI-specific and are expressed as preconditions on the relevant DafnyMPI methods. We refer to these properties as *axioms*, since, in developing our proof, we can assume they hold for any execution as they are discharged by the Dafny verifier. Each axiom other than AtStart is implicitly parameterized by a particular program state: for any state reachable from the initial state, if the axiom's premises hold, then so do its conclusions.

It is important to note that Dafny only models the behavior of a single (arbitrary) process and can only inspect the part of the program state that the process has access to. As a result, none of the axioms mentions the message buffer $\mathcal{B}_m$, whose content is internal to the MPI runtime. Similarly, the axioms only check whether a given tag is present in the send or receive buffers when the process to which the axiom applies is the intended sender or receiver. Additionally, because the Dafny verifier assumes all method calls terminate, it cannot itself account for the possibility of deadlock. DafnyMPI builds on top of this foundation to establish deadlock freedom explicitly. At the same time, Dafny's non-blocking assumption guarantees that the axioms hold in any state reachable under that assumption. This allows us to reason about a program state by considering what must hold in future states that are guaranteed to be reached. In particular, because Dafny proves that the final barrier count for any process is equal to BarrierCount(N), we can show that the barrier count never exceeds this value, which is one of the lemmas introduced in Section 4.2.5.

The axioms should also be understood in conjunction with the reduction rules presented above. For example, we say that the AtWait axiom in Figure 4.1 guarantees that tags increase monotonically but this is only true when we consider the axiom in conjunction with the WaitRecv and WaitSend reduction rules in Figure 4.3. Specifically, the two reduction rules always

Table 4.1: Axioms of the core calculus (implicitly parameterized by program state, except for AtStart).

Axiom	Premises	Conclusions
AtStart		$S_0 = (\bigcup_{0 \le i < N} \langle c;\ \textbf{barrier}, \{\texttt{rank} \mapsto i, \texttt{size} \mapsto N\}, -1, 0\rangle, \emptyset, \emptyset, \emptyset)$
AtSend	$c_{i,k} = \textbf{isend}\ e\ x\textbf{;}\ c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$i = \textsc{Sender}(v, N) \quad \langle x, \sigma_{i,k}\rangle \Downarrow \textsc{Message}(v) \quad v \notin \mathcal{B}_{s,k}$
AtRecv	$c_{i,k} = \textbf{irecv}\ e\ x\textbf{;}\ c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$i = \textsc{Receiver}(v, N) \quad v \notin \mathcal{B}_{r,k}$
AtWait	$c_{i,k} = \textbf{wait}\ e\textbf{;}\ c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$v > t_{i,k}$ $\forall v'.\ t_{i,k} < v' < v \Rightarrow (\textsc{Sender}(v', N) \neq i \wedge \textsc{Receiver}(v', N) \neq i)$ $\textsc{Barrier}(b_{i,k}, N) \le v < \textsc{Barrier}(b_{i,k}+1, N)$ $(i = \textsc{Sender}(v, N) \wedge v \in \mathcal{B}_{s,k}) \vee (i = \textsc{Receiver}(v, N) \wedge v \in \mathcal{B}_{r,k})$
AtBarrier	$c_{i,k} = \textbf{barrier;}\ c$ $v = \textsc{Barrier}(b_{i,k}+1, N)$	$v > t_{i,k}$ $\forall v'.\ t_{i,k} < v' < v \Rightarrow (\textsc{Sender}(v', N) \neq i \wedge \textsc{Receiver}(v', N) \neq i)$ $\forall v.\ \textsc{Sender}(v, N) = i \Rightarrow v \notin \mathcal{B}_{s,k}$ $\forall v.\ \textsc{Receiver}(v, N) = i \Rightarrow v \notin \mathcal{B}_{r,k}$
AtEnd	$c_{i,k} = \textbf{skip}$	$b_{i,k} = \textsc{BarrierCount}(N)$
AtSet	$c_{i,k} = \textbf{set}\ x\ e\textbf{;}\ c$	$\forall v.\ \textsc{Sender}(v, N) = i \Rightarrow (v \notin \mathcal{B}_{s,k} \vee x \neq \mathcal{B}_{s,k}(v))$ $\forall v.\ \textsc{Receiver}(v, N) = i \Rightarrow (v \notin \mathcal{B}_{r,k} \vee x \neq \mathcal{B}_{r,k}(v))$ $x \neq \texttt{rank}$
AtRead	$\textsc{Receiver}(v, N) = i$ $\mathcal{B}_{r,k}(v) = x$	$x \notin$ direct subexpressions of $c_{i,k}$

set the new tag $t_{i,k+1}$ to match the argument of the **wait** call, which, in turn, is guaranteed to be greater than the previous tag $t_{i,k}$ by the AtWait axiom.

Figure 4.1 presents the axioms of our language. The AtStart axiom describes the initial state of the program as outlined above in Section 4.2.1. AtSend and AtRead state that a message with a given tag can only be sent or received by its designated Sender and Receiver. Both axioms also require that the corresponding buffers are empty to disallow repeatedly sending or receiving the same message before calling **wait**. Additionally, AtSend guarantees that the message payload is as specified by the Message function.

The AtWait axiom enforces the tag ordering property. First, it guarantees that message tags used with **wait** increase monotonically for each process. Second, it enforces that no **wait** call is skipped—if the process is the designated receiver or sender of a message, it must receive or send this message before moving on to messages with higher tags. Third, the axiom ensures that no barriers are skipped between successive **wait** calls. Finally, it ensures that **wait** is only called

Table 4.2: Definition of deadlock an related terms.

Term	Conditions
$\textsc{OnSend}((\langle \textbf{wait } e\textbf{; } c', \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\langle e, \sigma\rangle \Downarrow t' \quad i = \textsc{Sender}(t', N) \quad t' \notin \mathcal{B}_m$
$\textsc{OnRecv}((\langle \textbf{wait } e\textbf{; } c', \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\langle e, \sigma\rangle \Downarrow t' \quad i = \textsc{Receiver}(t', N) \quad t' \notin \mathcal{B}_m \quad t' \notin \mathcal{B}_s$
$\textsc{OnBarrier}(\langle \textbf{barrier; } c', \sigma, t, b\rangle)$	
$\textsc{Terminated}(\langle \textbf{skip}, \sigma, t, b\rangle)$	
$\textsc{Deadlock}((\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\lvert\mathcal{P}_3\rvert \neq N \quad \lvert\mathcal{P}_4\rvert \neq N$ $\textsc{OnRecv}(p, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m), \ \forall p \in \mathcal{P}_1$ $\textsc{OnSend}(p, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m), \ \forall p \in \mathcal{P}_2$ $\textsc{OnBarrier}(p), \ \forall p \in \mathcal{P}_3$ $\textsc{Terminated}(p), \ \forall p \in \mathcal{P}_4$

after the corresponding **isend** or **irecv**.

AtBarrier requires that processes call barriers in order and wait on all send and receive operations that should precede them. Similar to the Barrier rule above, the AtBarrier axiom should be understood to apply to the final barrier as well. The axiom also guarantees that the send and receive buffers are empty at the barrier, i.e., there may be no ongoing send or receive calls. Because the last command executed by every process is a barrier, this last requirement guarantees that every **irecv** and **isend** is eventually followed by a matching **wait**. At the end of each process's execution, the AtEnd axiom states that the process passed exactly $\textsc{BarrierCount}(N)$ barriers.

Finally, AtSet and AtRead prevent non-deterministic behavior by disallowing reads or writes to variables that are used in interprocess communication—that is, those referenced in send or receive buffers. We explain in Section 4.3 how these two axioms are enforced by Dafny in practice when discussing the safe use of read and write buffers. AtSet also prevents changes to variable `rank`, which means we can rely on it always being equal to the process' actual rank in the operational semantics.

4.2.4 Deadlock

Table 4.2 introduces the terminology used to formally define and reason about deadlock. We begin by listing the conditions under which we consider a process to be blocked:

- A process is blocked on a send if it is about to execute a **wait** command for a message where

it is the designated sender, but the message contents have not yet been transferred to the message payload buffer. In this case, the sender remains blocked until the receiver reaches the corresponding **wait** command. (This condition is defined by the OnSend predicate in the table.)

- A process is blocked on a receive if it is about to execute a **wait** command for a message where it is the designated receiver, but neither the message payload buffer nor the send buffer contains the relevant tag. This means the sender has not yet initiated the communication. (This condition is defined by the OnRecv predicate in the table.)

- A process may blocked on a barrier if it is about to execute the **barrier** command, as defined by the OnBarrier predicate in the table. As long as not all other processes have reached the barrier, the process will remain blocked.

With these definitions in place, we can now formally define deadlock. A deadlock is any state, other than the one in which all processes have terminated, where no reduction rule applies. Note that we exclude the FreeBuffer rule from this condition because any state where only FreeBuffer applies eventually leads to a deadlock. One can see that this definition of deadlock is equivalent to every process being blocked or terminated, except when all processes have terminated. This is captured by the definition of Deadlock in Table 4.2. All processes in a deadlocked state must belong to one of four groups: those blocked while waiting for a receive or send operation to complete ($\mathcal{P}_1$ and $\mathcal{P}_2$ in the table), those blocked on a **barrier** ($\mathcal{P}_3$), and those that have terminated ($\mathcal{P}_4$). Additionally, it is not possible for all processes to belong to $\mathcal{P}_4$ (because then the program would have terminated) or $\mathcal{P}_3$ (because then the processes would pass the barrier). Note that the OnRecv and OnSend predicates specifically rule out states where WaitSend, WaitRecv, TransferOnWait, or TransferNoWait could be applied. With deadlock formally defined, we now proceed to prove deadlock freedom.

4.2.5 The Deadlock Freedom Proof

In this section, we provide the complete proof that our core calculus does not admit deadlocks. The lemmas and the theorem that follow apply to any program in the core calculus that also respects the axioms presented above.

The key ideas behind the proof can be seen by focusing on Theorem 1. The intermediate lemmas serve to formalize trivial properties of message tags and buffers.

Lemma 1 (NoTwoTagsTheSame)**.** *If two processes are about to execute* **wait** *commands in a deadlocked state, the corresponding message tags must be different.*

Let $i \neq j$ be distinct process ranks, and let k be a state ID. Suppose $\textsc{Deadlock}(S_k)$. Suppose $c_{i,k} = \textbf{wait}\ e_i\textbf{;}\ c'$ and $c_{j,k} = \textbf{wait}\ e_j\textbf{;}\ c''$, and $\langle e_i, \sigma_{i,k}\rangle \Downarrow v_i$, $\langle e_j, \sigma_{j,k}\rangle \Downarrow v_j$. Then $v_i \neq v_j$.

Proof. Assume, for the sake of contradiction, that $v_i = v_j = v$. By the AtWait axiom applied to processes i and j, and without loss of generality, we know $\textsc{Receiver}(v, N) = i \neq j = \textsc{Sender}(v, N)$. By the AtWait applied to process j, we also know that $v_j = v \in \mathcal{B}_s$. Because S_k is a deadlocked state and because $c_{i,k} = \textbf{wait}\ e_i\textbf{;}\ c'$ and $\textsc{Receiver}(v, N) = i$, the OnRecv predicate must hold for process i in state S_k. Therefore, $v_i = v \notin \mathcal{B}_s$. This contradicts our earlier assertion that $v_j = v \in \mathcal{B}_s$. Therefore, $v_i \neq v_j$. □

Lemma 2 (NonEmptyBuffer)**.** *If a process is responsible for sending or receiving a message with tag $v \in \mathbb{N}$, and that process's max used tag value eventually reaches v, then at some point during execution, the message buffer must have contained the payload tagged v.*

Let i be a process rank and v be a tag such that $\textsc{Sender}(v, N) = i \vee \textsc{Receiver}(v, N) = i$. Suppose there exists a reduction $S_0 \longrightarrow \cdots \longrightarrow S_k$. Suppose further that $t_{i,k} \geq v$.

Then, there exists state S_j that belongs to the reduction sequence $S_0 \longrightarrow \cdots \longrightarrow S_k$ such that $v \in \mathcal{B}_{m,j}$.

Proof. By the AtStart axiom applied to process with rank i, $t_{i,0}$ = -1, so $t_{i,0} < v \leq t_{i,k}$.

We now prove that there must exists some state ID l, such that $0 < l \leq k$ and $t_{i,l} = v$. Suppose, for the sake of contradiction, that no such l exists. Then let m be the largest state ID such that $0 \leq m < k$ and $t_{i,m} < v$. By construction, $t_{i,m+1} > v$. Therefore, the reduction from S_m to S_{m+1} must change the value of t_i. The only applicable reduction rules are WaitSend and WaitRecv. According to the premises of both rules, $c_{i,m} = \textbf{wait}\ e$ for some e, and $\langle e, \sigma_{i,m} \rangle \Downarrow t_{i,m+1}$. This allows us to invoke the AtWait axiom, which guarantees that $\forall v'. t_{i,m} < v' < t_{i,m+1} \Rightarrow (\textsc{Sender}(v', N) \neq i \wedge \textsc{Receiver}(v', N) \neq i)$. However, we know this quantified statement to be false, in particular because one of $\textsc{Sender}(v, N) = i$ or $\textsc{Receiver}(v, N) = i$ is true according to the premises of the lemma. Therefore, our assumption must be wrong and $t_{i,l} = v$ for some state ID l.

Let j be the smallest state ID value such that $0 < j \leq k$ and $t_{i,j} = v$. We know such j exists because of the result established in the previous paragraph. By construction, $t_{i,j-1} \neq v$. Therefore, the reduction from S_{j-1} to S_j must change the value of t_i. The only applicable reduction rules are WaitSend and WaitRecv, which both requires that $t_{i,j} = v \in \mathcal{B}_{m,j}$. □

Lemma 3 (BufferStaysNonEmpty)**.** *If the process is the intended receiver or sender of a message in the payload buffer but the message's tag is higher than that of any message the process has received or sent so far, then the message must remain in the buffer.*

Let i be a process rank and v be a tag such that $\textsc{Receiver}(v, N) = i \vee \textsc{Sender}(v, N)$. Let l and k be state IDs, where $0 \leq l \leq k$, and $S_0 \longrightarrow \cdots \longrightarrow S_l \longrightarrow \cdots \longrightarrow S_k$ is a state reduction sequence. Assume that $v \in \mathcal{B}_{m,l}$ and $t_{i,k} < v$. Then $v \in \mathcal{B}_{m,k}$.

Proof. We will first show that t_i can never decrease. Note that the only reduction rules that change the value of t_i are WaitRecv and WaitSend. Furthermore, whenever a reduction is performed according to these rules, the AtWait axiom applies to the preceding state. This axiom guarantees that the new value t_i following the reduction must be greater than the one before it. It follows that t_i may never decrease.

We now return to our original proof goal. Suppose, for the sake of contradiction, that $v \notin \mathcal{B}_{m,k}$. Then, v must be removed from the buffer at some point during the reduction. The only

reduction rule that may remove elements from the message payload buffer is FreeBuffer, which requires that $t_{\text{Receiver}(v,N)} \geq v$ and $t_{\text{Sender}(v,N)} \geq v$ in the state in which the element is removed. However, this would not hold for any state with ID less than or equal to k because i must be either the sender or receiver of the message, we know that $t_{i,k} < v$, and t_i cannot decrease. Therefore, the FreeBuffer rule would never apply for tag v in any state prior to or including k and, therefore, $v \in \mathcal{B}_{m,k}$. □

Lemma 4 (NumberOfBarriers)**.** *If the process has a barrier at the end of its command sequence, its internal barrier counter is strictly less than BarrierCount(N).*

Let i be a process rank and let k be a state ID. Suppose that $c_{i,k} = c$**; barrier** for some command c. Then $b_{i,k} < \text{BarrierCount}(N)$.

Proof. Note that the only reduction rule that can change b_i is the Barrier rule and that this rule increments b_i for each barrier passed. In other words, b_i can never decrease during program execution for any process. Note that if the process with rank i ever terminates in some state with ID m, the process would have to have passed the barrier at the end of its command sequence. Therefore, its barrier counter in that state is $b_{i,m} \geq b_{i,k} + 1 > b_{i,k}$.

Consider that under the non-blocking Dafny semantics, the process will eventually terminate by reaching state with rank $m > k$ such that $c_{i,m} =$ **skip**. By the AtEnd axiom applied to this state, $\text{BarrierCount}(N) = b_{i,m}$. Combined with the result of the previous paragraph, this gives us $\text{BarrierCount}(N) > b_{i,k}$. □

Theorem 1 (Deadlock Freedom)**.** There exists no reduction from the initial state S_0 to a deadlocked state.

Proof. Suppose, for the sake of contradiction, that there exists a reduction $S_0 \longrightarrow \cdots \longrightarrow S_k$ and $\text{Deadlock}(S_k)$. By the definition of Deadlock, there must then exist some $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{B}_s$, $\mathcal{B}_r$, and $\mathcal{B}_m$ such that $S_k = (\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4, \mathcal{B}_s, \mathcal{B}_r, \mathcal{B}_m)$ and $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, $\mathcal{P}_4$ have the properties described in Table 4.2. There are two cases, either $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$ or $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$.

- **Case 1**. Suppose $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$. This means that all processes that are still running are waiting on some barrier. By the definition of DEADLOCK, $|\mathcal{P}_3| \neq N$. In other words, there must be at least one process that has terminated. Let that process be p_m with rank m. Because this process has terminated, the ATEND axiom applies, meaning that $b_{m,k} = \text{BARRIERCOUNT}(N)$. Note that all processes must always share the same b, since all processes pass barriers simultaneously and all processes start with $b = 0$ according to the ATSTART axiom. However, we know that at least one process, let us call it $p_i \in \mathcal{P}_3$ has not terminated yet. By the NUMBEROFBARRIERS Lemma we know that $b_{i,k} < \text{BARRIERCOUNT}(N)$. However, we also know that $b_{i,k} = b_{m,k} = \text{BARRIERCOUNT}(N)$. Hence, we reached a contradiction. Therefore, $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$

- **Case 2**. Suppose $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$. This means that at least one process must be blocked on a **wait** call for which it is the receiver or sender. Let m be the rank of the process that is waiting on a message with the smallest tag, namely that $c_{m,k} = \textbf{wait}\ e_m\textbf{;}\ c'$ and $\langle e_m, \sigma_{m,k} \rangle \Downarrow v_m$, where v_m is smaller than the respective value for any other process in $\mathcal{P}_1 \cup \mathcal{P}_2$. Such a minimum exists because, according to NOTWOTAGSTHESAME lemma, no two processes may wait on the same tag. We will now assume that $p_m \in \mathcal{P}_1$ (the reasoning is symmetric when $p_m \in \mathcal{P}_2$.) Because $p_m \in \mathcal{P}_1$, we know that $m = \text{RECEIVER}(v_m, N)$.

  Now consider the process with rank $i = \text{SENDER}(v_m, N)$. We will now show that $t_{i,k} \geq v_m$. We know that $p_{i,k} \in \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4$. So there are three subcases:

  - Subcase 1: $p_i \in \mathcal{P}_1 \cup \mathcal{P}_2$. Then $c_{i,k} = \textbf{wait}\ e_i\textbf{;}\ c'$ and $\langle e_i, \sigma_{i,k} \rangle \Downarrow v_i$. By construction, we know that $v_i > v_m$. By ATWAIT axiom, we also know that $v_i > t_{i,k}$. By the same axiom, it must be the case that $t_{i,k} \geq v_m$ since there must be no tag between v_i and $t_{i,k}$ for which i is the sender.
  - Subcase 2: $p_i \in \mathcal{P}_3$. By the ATBARRIER axiom applied to process i in state k, it follows that $\text{BARRIER}(b_{i,k} + 1, N) > t_{i,k}$. By the ATWAIT axiom applied to process m in state k, $v_m < \text{BARRIER}(b_{m,k} + 1, N)$. As discussed above, all processes have the same value

of b in any given state. Hence, we have $b_{i,k} = b_{m,k}$ and $v_m < \textsc{Barrier}(b_{i,k} + 1, N)$. By the AtBarrier axiom applied to process i in state k, there must be no tag between $t_{i,k}$ and $\textsc{Barrier}(b_{i,k} + 1, N)$ for which i is the sender, so therefore $t_{i,k} \geq v_m$.

- Subcase 3: $p_i \in \mathcal{P}_4$ meaning that p_i has terminated. Note that by the AtStart axiom, p_m has a **barrier** command at the end of its command sequence. Since p_m has not passed this barrier yet, the NumberOfBarriers lemma applies and $b_{m,k} < \textsc{BarrierCount}(N)$. At the same time, the AtEnd axiom applies to process with rank i, so $b_{i,k} = \textsc{BarrierCount}(N)$. However, this leads to a contradiction because, as discussed above, all processes have the same value of b in any given state, so $b_{i,k} = b_{m,k}$. Therefore, Subcase 3 is not possible.

We have now established that $t_{i,k} \geq v_m$. Consider also that by the AtWait axiom, $v_m > t_{m,k}$. Since $t_{i,k} \geq v_m > t_{m,k}$, we can invoke the NonEmptyBuffer lemma for process i and the BufferStaysNonEmpty lemma for process m. This gives us that $v_m \in B_{m,k}$. However, this is a contradiction because the Deadlock predicate requires that $v_m \notin B_{m,k}$. Therefore, $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$.

We have now shown that both cases lead to a contradiction. Therefore, our initial assumption must be wrong and Deadlock is not reachable. □

The proof above connects the behavior described by the MPI standard—formalized as the operational semantics of the core calculus—with the specifications we write in Dafny—formalized as the axioms of the core calculus—to establish that the specifications prevent deadlock in programs that use DafnyMPI. The only other instance of interprocess reasoning necessary to establish full program correctness concerns the contents of the point-to-point messages, which we discuss next.

4.2.6 Rely-Guarantee Reasoning for Message Contents

For Dafny to verify properties about the values computed by each process, it must know the contents of the messages those processes receive. It is clear that the contents will always be exactly as specified by the Message function since the AtSend axiom guarantees that any message that is ever sent abides by it. More formally, this is an example of the rely-guarantee reasoning. Dafny can rely on each message received by a given process to abide by the user-provided specification because it guarantees that every message ever sent also abides by the specification. In the Dafny code, this is captured by a postcondition on each **wait** call, which describes the message being received. With this postcondition in place, no further interprocess reasoning is necessary to support the MPI primitives in our core calculus: all other properties needed for full program correctness can be established directly in Dafny. In the section that follows, we discuss the Dafny-specific aspects of our approach in more detail.

4.3 Implementation

DafnyMPI consists of two parts. The first part is the specifications of MPI primitives, which are instance methods of the `MPI.World` class. These specifications, along with a few supplementary lemmas and functions, comprise 379 lines of Dafny code and form the trust base of the library. This file does not get compiled to Python directly; instead, each specified MPI primitive is implemented as a Python wrapper function (about 60 lines of code total), which we provide alongside the library.

The second part of DafnyMPI defines two classes for manipulating 1D and 2D arrays of real numbers, with the added ability to track which elements are currently in use by MPI (see below). For maximum portability, all array operations such as rolling, slicing, scalar and per-element multiplication and addition, etc. are implemented in Dafny and rely only on calls to the constructor and three primitive Get, Set, and Size methods. This part of the library totals 1772 lines of Dafny code, which can be directly compiled to Python (or any other target language supported by Dafny)

with the exception of the constructor and the three primitive methods mentioned above. The latter are all implemented in Python using the Numpy library, and the code is provided alongside the DafnyMPI library. In practice, all array operations could be delegated to Numpy at runtime to optimize for performance but we opted to minimize the trust base instead.

Next, we discuss some of the implementation challenges that we could ignore in Section 4.2 but must address in DafnyMPI.

Dealing with Tag Overflow. The proof of deadlock freedom in Section 4.2.5 assumes that MPI tags are unbounded. In reality, different MPI implementations impose different limits on tag size, e.g., the Intel MPI library uses 20 bits to represent tags [58]. The only requirement imposed by the MPI Standard itself is that the upper bound on the tag must be no less than 32767 [83].

To ensure tags are always valid, DafnyMPI adds a proof obligation to require that the Barrier function we introduce in Section 4.2.1 never increases by more than 32767. This ensures that no more than 32767 tags are used between any two barriers. Therefore, even if tag overflow occurs, every tag between any two barrier calls remains unique, allowing processes to deterministically identify messages.

Supporting Blocking Send and Receive. Our core calculus models only non-blocking send and receive operations. As mentioned earlier, we can easily encode the blocking variants as a **isend** or **irecv** followed immediately with a **wait**. The DafnyMPI library supports blocking send and receive by expressing them in this way for verification purposes while using the corresponding `MPI_SEND` and `MPI_RECV` commands at runtime. Support for blocking sends and receives simplifies the code by removing the need to manage `MPI_REQUEST` objects when non-blocking communication provides no additional benefit.

Supporting Collective Operations The MPI standard describes a number of one-to-many, many-to-one, and many-to-many collective operations such as `MPI_SCATTER`, `MPI_GATHER`, and `MPI_ALLREDUCE`. The core calculus doesn't include these features but DafnyMPI does support the

latter two operations by treating them as decorated barriers. Specifically, DafnyMPI imposes a restriction on collective operations requiring that they are used only when there are no pending send or receive operations running in the background. At runtime, DafnyMPI explicitly puts a barrier before any such collective operation. Therefore, the addition of these new collective operations cannot introduce deadlocks. Any additional information that is necessary to reason about the given operation's result is provided by the user when the MPI is initialized, similar to how the one-to-one messages are specified.

For example, to describe the Gather operation on line 34 of our running example, the user must specify that the final ntth barrier is a Gather operation and provide the exact functional specification of the information that will be collected (top Dafny snippet in Figure 4.1). At the call to Gather, in addition to any conditions that are required for a regular barrier, the user must also prove that the data sent by each process matches the respective portion of the specified result and that the relevant buffers can be read from and written to. The specification of Gather then guarantees that the data received by the root process will be exactly as specified above.

Safe Use of Read and Write Buffers Our core calculus does not include aliasing, and the AtSet and AtRead axioms we introduce above only state that variables given as input to **irecv** or **isend** must not be used until the operation is completed. In practice, the situation is more complex. A typical MPI operation takes as input a reference to a buffer, the type of elements it contains, and the total number of elements. The MPI implementation may then read from or write to the buffer at any point until the operation completes. The buffer must be contiguous in memory so that the position of a given element can be identified using a simple offset. Send commands can and often do use overlapping buffers, but any receive command must have its buffer to itself. Additionally, a process cannot write to a buffer currently used by any MPI operation or read from any buffer that is being written to by an MPI call. Implementing Dafny checks to ensure these rules are followed is challenging for two reasons. First, Dafny does not support pointer arithmetic, so we have to add additional specifications to explicitly model the layout of the array

in memory. Second, we cannot simply mark any object used by the MPI as locked because many MPI programs work by manipulating large arrays of data, where multiple MPI commands may simultaneously read from and write to different parts of the array.

To support buffers, we introduce two custom classes for 1D and 2D arrays of floating point numbers. Both classes implement a shared interface (a.k.a. *trait* in Dafny) for modeling memory contiguity and per-element read and write locks as part of the ghost state. Write locks are flags, where if a portion of an array is locked for a write, that portion of the array cannot be further read or written to. Read locks, on the other hand, are counters, where arbitrarily many MPI operations may be reading the same elements at the same time, and writes are disallowed so long as at least one MPI operation is reading the relevant elements. On any receive or send command, the user provides as input the reference to the array that will serve as the buffer, alongside the index from which the data will be read from or written to and the size of the message. The preconditions on the corresponding MPI call require that the relevant elements of the array are available to be read or written to. If the MPI call is non-blocking, the postconditions then mark the relevant elements of the array as locked until the operation completes. For 2D arrays, we model C-style contiguity (as guaranteed at runtime by the Numpy constructor), meaning that a row or series of rows is contiguous in memory but columns are not, unless the array only has one column.

Functional Equivalence with Sequential Specification To ensure functional correctness and measure performance gain from parallelization, we also developed a sequential implementation for each MPI program we verified and proved equivalence between the two. We did this by developing a functional specification of the computation and then proving that both the sequential and the parallel versions return the same values as the specification. To reason about the parallel version of the code, the equivalence proof must account for the contents of each received message, which is made possible as described in Section 4.2.6.

Termination Of all the correctness properties we list in Section 4.1.2, proving termination requires the least amount of work. Once deadlock freedom is established and the verifier success-

fully rules out runtime errors such as out-of-bounds array access, loops and recursions remain as the only obstacles to termination. We establish that loops and recursion terminate as is standard in Dafny, by specifying some positive quantity that decreases on each iteration and is bound below by zero. For the outer for loop in our running example in Figure 4.1, this quantity is `nt-n`, the number of iterations that remain.

Proof engineering considerations A common issue that arises in automated verification is *brittleness*: the more information one gives to the verifier, the more likely it is that the verification query times out. To address this problem, we follow the approach previously introduced by Dafny engineers [82, 63] in making most functions—particularly those involving quantifiers—*opaque*, hiding their definitions from the verifier unless explicitly revealed. We also specify the behavior of some such functions using specialized lemmas instead of using post-conditions, which similarly gives the user more control over which facts the verifier has access to at any given point. We recommend that the clients of DafnyMPI follow the same approach to avoid brittleness in their proofs.

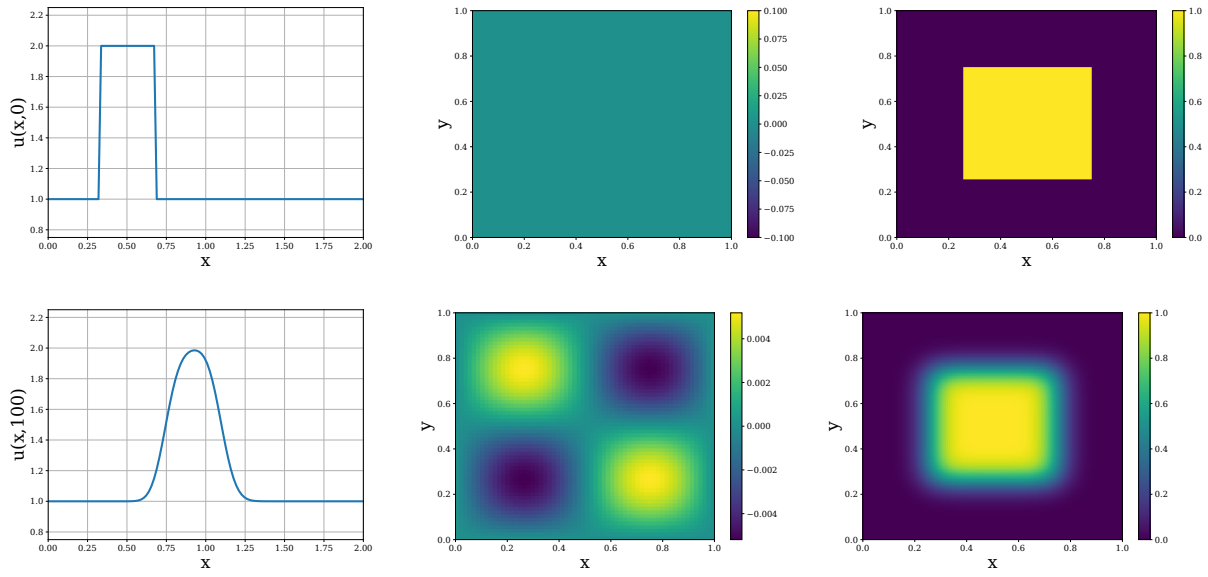


Figure 4.5: Benchmark PDE simulations. Top row: initial conditions; bottom row: solutions after 100 time steps as computed by DafnyMPI. From left to right: Linear Convection, Poisson, Heat Diffusion

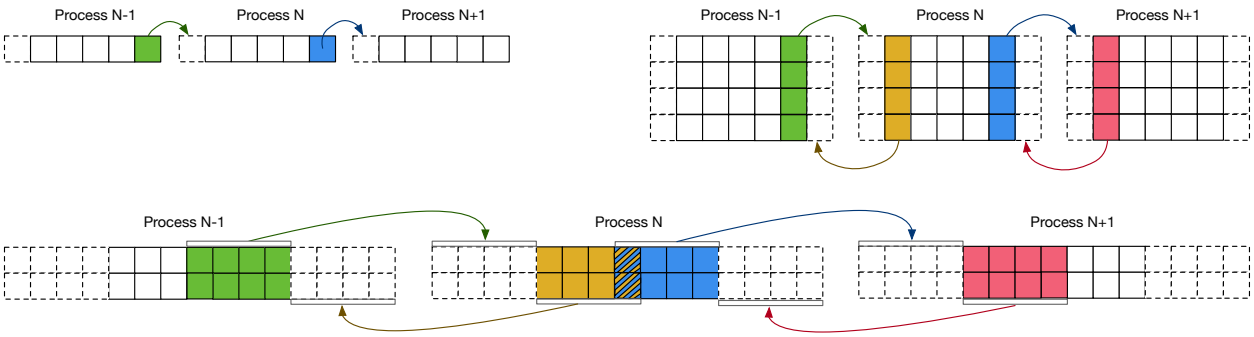


Figure 4.6: Structure of MPI Communication for the Linear Convection (top left), Poisson (top right, transposed), and Heat Diffusion (bottom, transposed) benchmarks.

4.4 Evaluation

To determine whether DafnyMPI meets its design goals, we conducted a case study by implementing and verifying three common numerical PDE solvers. We evaluated the proof engineering effort, in terms of auxiliary proof-related code that must be added to the program being verified, and runtime performance, in terms of performance of the parallel implementation. Our results suggest that DafnyMPI can offer significant runtime improvements, with speedups of up to five times, while keeping the added proof burden manageable, even as the amount of proof-related code increases compared to sequential implementations. We conducted all experiments on a MacOS machine with an M3 processor and 48 GiB of memory, and the Dafny solver was configured to use an unlimited number of virtual cores for verification.

4.4.1 Benchmarks

We used DafnyMPI to implement numerical solvers for three different PDEs, each of which uses a different communication pattern. Figure 4.5 illustrates the solver results after simulating each PDE for 100 time steps using DafnyMPI. Figure 4.6 sketches the MPI communication pattern for each solver. The benchmarks are as follows:

- **Linear Convection.** As discussed in Section 4.1, we implemented the upwind scheme for solving the 1D linear convection PDE. The equation models a wave propagating at constant speed, as shown in the left column of Figure 4.5. Notice that while the analytical solution to

linear convection preserves the wave shape exactly, the numerical solution introduces diffusion and cannot fully maintain sharp edges in the initial condition. The top left schematic in Figure 4.6 shows that, in the MPI version, each process must send the rightmost value in its domain to its neighbor at each iteration.

- **Poisson.** We implemented Jacobi iteration for solving the 2D Poisson equation with Dirichlet boundary conditions. The Jacobi method updates the value at a point based on all four of its neighbors on each iteration. For the MPI implementation, we partition the domain into equal horizontal stripes. Each process must send the top and bottom rows of its stripe to its neighbors and receive the two boundary rows in return. The top right schematic in Figure 4.6 illustrates this communication pattern (we use columns instead of rows on the Figure due to space constraints.) The solution to the Poisson equation is a function whose second spatial derivative (Laplacian) is equal to another function, provided as input. Over the course of multiple iterations, the solution can be expected to converge to an equilibrium, so the processes must communicate on each iteration to check whether convergence has been reached. The latter is handled via a collective call to `MPI_ALLREDUCE`. The middle column of Figure 4.5 shows the solution to the Poisson equation after 100 iterations, where we set the initial conditions to a matrix of zeroes and the right-hand side of the equation to a function $f(x, y) = \sin(2\pi x)\sin(2\pi y)$, whose values on the discretized domain are precomputed using externally implemented trigonometric functions.

- **Heat Diffusion.** We implemented the fourth-order Runge-Kutta method (RK4) and applied it to the 2D heat diffusion equation. The right column of Figure 4.5 illustrates the behavior described by the equation using a heatmap. The heat that is originally at the center of the plot gradually dissipates as the simulation proceeds. The fourth-order Runge-Kutta method computes the value at a point based on a region around it that includes any points within the Manhattan distance of 4. For the MPI implementation, we again partition the domain in equal horizontal stripes. Unlike in Poisson, however, the processes must com-

municate 4 rows of data at a time, which introduces an additional layer of complexity to the problem. The bottom schematic in Figure 4.6 illustrates this process (as before, we use columns instead of rows.) Note that the data the process sends to its two neighbors can be overlapping, and the sends can happen concurrently. The receives, however, must use separate buffers.

4.4.2 Proof Engineering Effort

To provide insight into the engineering effort involved, Table 4.3 reports the number of lines of code (LOC) for each of the three benchmarks described above. While it is only an indirect measure, LOC roughly corresponds to the amount of effort that went into verifying each respective part of the code. We separately list the number of lines in the specification (Spec), sequential implementation (Seq), and parallel MPI implementation (Par) for each benchmark. For the Heat Diffusion and Poisson benchmarks, we also list the size of the codebase that is shared between the two implementations (Shared), which corresponds to a number of helper methods for implementing the Jacobi and RK4 techniques, respectively. As can be expected, due to the increased complexity, the parallel version requires more lines of code to implement.

Ghost vs Executable Code. Table 4.3 also reports the percentage of lines in each group of files that are *ghost*. Ghost code is code that is not executable and whose only purpose is to support the proof. As can be seen from the figure, the percentage of ghost code increases significantly for the parallel version, which reflects the increased complexity of the proof. At the same time, the percentage of ghost code in the parallel implementations is comparable to that in some of the more complex subroutines of the sequential version. For example, the 4th-order Runge-Kutta method (the Shared subcategory of the Heat Diffusion benchmark in Table 4.3) requires a significant amount of auxiliary proof code to relate the functional specification to the rolling array operations used in the implementation. As a result, the ghost code makes up 81% of this shared component, which is close to the 89% observed in the parallel implementation of the same benchmark.

Table 4.3: Number of lines of code (LOC) and verification time as measures of proof complexity.

	Linear Convection			Poisson				Heat Diffusion				Total
	Spec	Seq	Par	Spec	Shared	Seq	Par	Spec	Shared	Seq	Par	
LOC	209	106	379	384	126	153	1626	328	406	137	1516	5370
% Ghost	100	45	65	100	72	55	91	100	81	57	89	87
Time to verify (s)	3	3	25	4	6	4	148	3	11	4	154	371

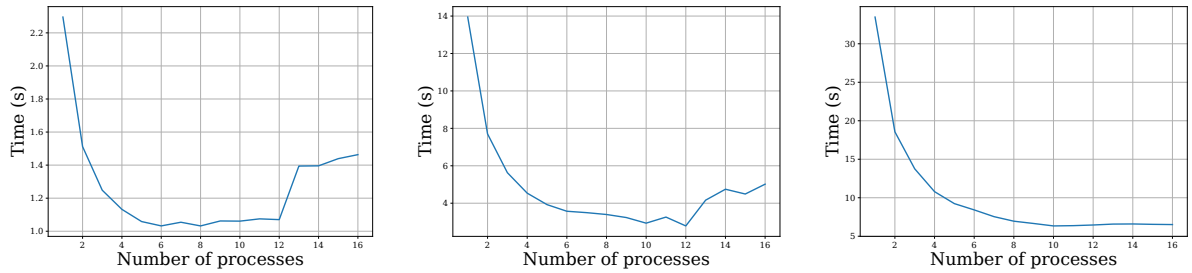


Figure 4.7: Running time as function of the process count. From left to right: Linear Convection, Poisson, Heat Diffusion.

Verification and Compilation Speed It takes about 6 minutes to verify all of the benchmarks and under 20 seconds to compile the code to Python using Danfy's built-in compiler. Heat Diffusion is the most expensive benchmark when it comes to verification time, as can be seen in Table 4.3. This is primarily due to the need to map the message payload between 2D and 1D representations multiple times during verification to ensure that the data being sent is contiguous in memory.

4.4.3 Runtime Performance

Given the considerable effort that goes into proving an MPI program correct, we want to be certain that the programs we write provide the desired performance boost when compared with corresponding sequential implementations. Thus, we transpiled the parallel implementations of all three benchmarks to Python using the Dafny compiler and ran them with different numbers of processes. [1] Figure 4.7 shows the results of this experiment. All time measurements are the

[1]For simplicity of reasoning about the program's result, our current proofs assume that the domain size is divisible by the number of processes. This assumption does not affect the communication pattern or the proof of deadlock

median of three runs. We use the sequential implementation for the one-process run. As expected, increasing the number of processes reduces the running time, achieving up to a fivefold speedup over the sequential version in the Heat Diffusion benchmark. However, as is typical, increasing the number of processes past a certain threshold leads to diminishing returns, and, for Linear Convection and Poisson, the running time even goes up once we assign 12 or more processes to the MPI program. We hypothesize that the running time increases because we ran the experiments on a machine with 16 cores, so giving 12 cores to the program might have interfered with the other processes running in the background, thereby increasing the total running time. This effect is most noticeable in the Linear Convection benchmark because this is the most lightweight of the benchmarks in terms of running time.

4.5 Discussion and Future Work

Comparison with a Proof Assistants An initial prototype of the DafnyMPI system was partially formalized in the Rocq proof assistant. To achieve this, we developed a domain-specific language similar to the core calculus in Section 4.2.1, alongside specifications for an interpreter for that language. While we still believe that this approach is feasible, during the verification of the linear convection benchmark in Rocq it became apparent that the proof engineering burden could be lessened considerably using the constructs built-in to Dafny, which leadi to the work on adding concurrency support for Dafny. The difference between the two approaches is most apparent when comparing the length of the partial proof of deadlock freedom for the Linear Convection example in Rocq (about 1000 lines, even with many admitted proof goals), which exceeded that of the entire program correctness proof in DafnyMPI (694 lines). It could be both feasible and useful to implement reasoning similar to DafnyMPI with other verification frameworks, particularly those that, like Dafny, delegate lower-level proof details to the verifier. We leave this task to future work.

freedom. When the domain size is not divisible by the number of processes, we round it up to the nearest divisible value before measuring running time. In our benchmarks, all domain sizes are divisible by 16, so experiments with 1, 2, 4, 8, and 16 processes are unaffected by this adjustment.

Floating-point Arithmetic Throughout our codebase, we use Dafny's real type to model floating point numbers. While common in verification, this approach does not provide a precise model of IEEE-754 or any other floating-point standard because it does not account for floating-point underflow, and does not prevent division by zero errors that may result from it. Several studies exist on automating the verification of programs that use floating-point arithmetic [1, 43], and it may be a possible direction for future work on DafnyMPI.

Running Time Because we model most array operations in Dafny, the runtime performance suffers compared to what could be achieved with native libraries such as Python's Numpy. This is, in part, by design: we chose to minimize the trust base by only relying on Numpy for single-element set and get operations and implementing all common array iterations on top of that foundation. In principle, it should be possible to replace our implementations of such operations as element-wise addition, slicing, and rolling, with corresponding Numpy operations at runtime, which should result in a significant performance boost.

4.6 Related work

There are several threads of related work.

MPI-specific verification tools Researchers have developed a number of verification systems for MPI. ISP [110], its successor DamPI [113], MOPPER [41], and MUST [53] are dynamic analysis tools that reason about MPI programs. In contrast to DafnyMPI, they do not prove ahead of time that a program will be correct for all runs. TASS [100] and MPI-SV [25] use symbolic execution and model checking. These approaches do not require the same amount of proof engineering as DafnyMPI, but they are also subject to potential state explosion as they have to explore many program paths and interleavings.

Model checking for Concurrency Model checking has been used successfully across industry and academia for verification of concurrent programs, e.g., using TLA+ [70] to reason about operating systems used on space missions [111] and cache-coherence systems [11]. As previously discussed, a major challenge of model checking is the state explosion problem. Moreover, scientific software is typically designed to run with an arbitrary input size and an arbitrary number of processes, thus it has an unbounded state space. With an unbounded state space, model checking approaches can detect bugs but cannot prove their absence.

Some researchers have explored addressing this challenge by pruning the state space for MPI programs [109]. Other researchers have explored providing auxiliary information, such as the number of iterations in a loop, to allow MPI programs with a larger state space to be verified using model checking [56]. In contrast to DafnyMPI, both approaches can only provide guarantees for the portion of the state space that was explored. This is because even after pruning, MPI programs of sufficient complexity can still have a prohibitively large state space.

Other Systems for Reasoning about Concurrency A variety of other approaches have been proposed for reasoning about concurrent and parallel programs. Concurrent separation logic (CSL), as exemplified by Iris [64] and Steelcore [107], extends Hoare logic with shared-memory semantics such as resources (heap), resource ownership, and the separating implication for heap predicates [19]. While CSL models the heap directly, DafnyMPI instead hides those details from the library user, managing its internal state during calls to the DafnyMPI functions. This reduces some of the proof engineering burden but is obviously restricted to programs using MPI. Additionally, CSL does not provide tools for reasoning about liveness or deadlock-freedom. For formal verification of those properties, one must rely on a supplemental logic such as that provided by LinearActris [61].

While Dafny lacks native support for concurrency, the DafnyInfoFlow project[101] has added support for concurrent reasoning, also using Rely-Guarantee. However, the intended correctness properties DafnyInfoFlow and DafnyMPI aim to verify are different. DafnyInfoFlow tracks se-

curity classifications for variables in concurrent programs, so that security violations, such as leaking information contained in a private or high-security variable to the user, can be avoided. In contrast, DafnyMPI focuses on deadlock freedom and termination guarantees and has no reasoning abilities for security-related concepts.

Several researchers have developed type systems that aim to prevent common concurrency bugs such as data races and deadlock. Examples include TyPiCal [68], based on session types; CLL [44], based on dependent types; and Asynchronous Liquid Separation Types [67], based on refinement types. In contrast to DanfyMPI, these type-based approaches do not reason about program equivalence.

Finally, researchers have developed a variety of other formal verification frameworks for concurrent programs, such as VCC[32], VerCors[15], and CIVL[79]. A major difference between DafnyMPI and these frameworks is their core design. VCC requires annotating C code with invariants and pre- or post-conditions; Vercors verifies Java code with separation logic annotations; and CIVL requires writing code in the Boogie modeling language directly.

4.7 Conclusion

We have introduced DafnyMPI, a novel verification library that brings formal reasoning about message-passing interface (MPI) parallel programs into Dafny, a language designed for sequential code. Our approach enables verification of core MPI properties including deadlock freedom, safe use of buffers, and functional equivalence with sequential specifications, all without requiring custom concurrency logics or deep expertise in process calculi. By leveraging Dafny's built-in verification capabilities and layering MPI-specific invariants on top, we believe our approach achieves a balance of rigor, accessibility, and scalability.

Our formal model demonstrates that a small set of preconditions on message tags, barriers, and buffer usage suffices to guarantee deadlock freedom. These preconditions are enforced through a core calculus and a set of Dafny assertions. Experimental results on PDE solvers demon-

strate the viability of our method: although proof engineering remains nontrivial, the verified parallel implementations exhibit significant speedups over their sequential counterparts.

Looking forward, we envision extensions of DafnyMPI to support more expressive MPI features (e.g., communicators, wildcards), to integrate more seamlessly with scientific programming ecosystems, and to further reduce proof engineering effort. We hope that DafnyMPI will serve as a foundation for future work on verified high-performance computing and encourage the adoption of formal methods in domains where correctness is critical but traditional techniques are impractical.

Chapter 5

Conclusion and Future Work

In this dissertation, I presented several ideas for reusing Dafny's automated sequential program verifier beyond its initial intended purpose, with the aim of broadening Dafny's applicability and appeal.

In Chapter 2, we presented DTest, a tool for automated test generation for Dafny code. DTest directly alters the verification pipeline by making changes to how the Dafny code is transpiled to Boogie, an intermediate verification language. These changes allow querying the verifier for inputs that trigger the execution of a desired branch or statement in the program. Test suites generated by DTest achieve up to 79% statement and 84% branch coverage on our benchmarks, thereby increasing confidence in the code's correctness.

In Chapter 3, we presented Metamorph, a tool for synthesizing large heap-based objects in Dafny. Unlike DTest, Metamorph does not alter Dafny behavior directly and instead works by repeatedly running the Dafny verifier on small examples and interpreting the output to fill in holes in the program sketch. Metamorph is partly inspired by a limitation of DTest: it cannot synthesize heap-based objects and must rely on creating runtime mocks. By contrast, Metamorph synthesizes provably correct Dafny code that creates objects according to user-provided specifications. We demonstrate that Metamorph works on a variety of benchmarks and can synthesize programs containing up to 57 method calls.

In Chapter 4, we presented DafnyMPI, a library for verifying message passing concurrent programs. Unlike DTest and Metamorph, whose novelty lies primarily in interpreting the output of the verifier in new ways, DafnyMPI fundamentally expands the range of problems that can be verified with Dafny. Specifically, DafnyMPI redefines a number of MPI primitives as Dafny methods such that the specifications of these methods collectively guarantee the absence of deadlocks and allow establishing functional equivalence with a sequential version. We demonstrate the applicability of DafnyMPI by implementing a number of parallel PDE solvers in Dafny and showing that these parallel implementations are up to five times faster than their sequential counterparts.

Combined, DTest, Metamorph, and DafnyMPI achieve the previously stated goal of (i) offering additional benefits to the developer that would justify the transition to a verification framework, and (ii) supporting concurrent computation when necessary, while not complicating the verification of sequential programs. This dissertation demonstrates that automated verification should be viewed not as an end in and of itself, but as a basis for developing new techniques.

5.1 Broader Applicability

Most of this dissertation focuses on Dafny, which raises the question of how broadly these ideas apply to other verification-aware languages and tools. To answer this question, I now discuss two concrete examples of such tools and how the differences between them and Dafny affect the overall argument.

Perhaps the most direct competitor of Dafny is Viper [89]. Like Dafny, Viper uses Boogie [8] as its verification backend, meaning that the ideas behind DTest and Metamorph could be applied directly to Viper as well, barring the inevitable software engineering challenges associated with lifting counterexamples from Boogie to the syntax of Viper and its frontends. Unlike Dafny, Viper is built with concurrency in mind and directly supports permissions, among other features. Another difference between the two languages is their intended place in the verification pipeline. After they are verified, Dafny programs can be compiled to C#, Java, Python, Go, and

JavaScript. Viper, on the other hand, works in the opposite direction: it provides a common verification backend that accepts programs written in different frontend languages, including Chalice, Python (via Nagini [35]), Rust (via Prusti [3]), and Go (via Gobra [116]). Notwithstanding these differences, there is a place for a library like DafnyMPI in the Viper ecosystem. In fact, the Viper permission system may simplify modelling buffer locks, which DafnyMPI defines explicitly (as discussed in detail in Section 4.3). Were such a library available in Viper, a frontend like Nagini could then directly translate calls to Python's mpi4py library to corresponding methods in Viper. Furthermore, this functionality would be complementary to the lower-level concurrency that Viper already supports. Indeed, while it may be possible to redefine MPI functionality in terms of lower-level primitives already available in Viper, doing so would likely be very challenging: modelling MPI methods like `MPI_ALLREDUCE`, which is supported by DafnyMPI on the specification level, may require building a binary tree of processes, for example. The approach we discuss in Chapter 4 allows us to hide such details from DafnyMPI clients, improving the overall usability. I hypothesize that the ideas discussed in this dissertation should therefore be applicable to Viper as well.

A more distant point of comparison to Dafny is Frama-C [30], a framework for verifying C programs. Frama-C has a WP plugin [9] that, like in Dafny, uses weakest precondition calculus to discharge proof obligations to SMT solvers to verify that the program satisfies its specification. Unlike Dafny and because it targets C, Frama-C supports pointer arithmetic, which is also why the WP plugin comes with a selection of memory models that describe the heap at different levels of abstraction. The choice of the memory model may therefore affect the precision of counterexamples Frama-C generates, and, consequently, the applicability of approaches similar to DTest and Metamorph. In principle, however, generation of tests and/or inputs using symbolic constraints should be possible as evidenced by the existence of the PathCrawler plugin [115], which implements concolic execution within Frama-C. There also exists an early version of a plugin for Frama-C, called Conc2Seq, that facilitates verification of concurrent programs [14]. Conc2Seq automatically translates a concurrent C program with a fixed number of threads into a sequen-

tial program that, at each step, selects a thread at random and performs an atomic operation for that thread. While potentially promising, existing work focuses primarily on the correctness of such a transformation itself, and it is not clear how feasible verification of programs generated by Conc2Seq is. There is some surface level similarity with DafnyMPI, however, in that the number of thread/processes does not change during program execution. A more well-known system for verifying concurrent C code is VCC [32, 28]. VCC targets low-level concurrent C code, including data structures and primitives used for thread synchronization over shared memory. In theory, this should allow verification of the MPI primitives and even of the MPI runtime itself. However, verification of programs that use these MPI primitives presents a challenge at a different level of abstraction, and I believe that the ideas behind DafnyMPI are, therefore, complementary to VCC.

Finally, DTest is cited by subsequent work on Ada/SPARK test generation [45] as a similar approach. This latest research cites over-reliance of DTest on mocking as its main limitation, but this is also something that Metamorph addresses. Overall, I believe that this dissertation describes a series of broadly applicable steps developers of verification-aware languages could take to facilitate wider adoption of the technology.

5.2 Future Work

This dissertation opens up several directions for future research.

5.2.1 DTest and Metamorph

DTest and Metamorph offer substantial new capabilities that allow generating high-coverage test suites for Dafny code. At the same time, there is a limit to the size of the Dafny programs that these tools can handle, particularly when generating system-level tests or in the presence of large APIs. We hypothesize that concolic execution may present a more scalable alternative to DTest, as it provides the solver only with the information relevant to a specific execution path. At the same time a hypothetical concolic execution tool would, like Delfy [103, 26], depend on

the compilation backend and hence likely be more difficult to integrate with all Dafny features and types. Alternatively, if an interpreter for Dafny were developed, concolic execution could then be implemented on top of this functionality. In the end, an ensemble approach of different methods may be optimal, allowing the user to choose the approach that best suits their needs. In this way, the Dafny ecosystem may be extended to support multiple complementary testing plugins, similar to how plugins are treated in frameworks like Frama-C.

5.2.2 DafnyMPI

At the moment, despite its relative ease of use, DafnyMPI still incurs a significant overhead when it comes to proof engineering effort—up to 87% of our code is ghost code, meaning that its only purpose is to support the proofs. At the same time, it became apparent during the development of benchmarks for DafnyMPI that many concurrent programs share similar communication topologies, particularly when it comes to PDE solvers. Consequently, we hypothesize that it should be possible to prove correct not just individual applications but rather entire communication patterns. Such communication patterns could then be parametrized as desired to create new provably correct concurrent programs with minimal proof effort, making verification of concurrent programs even more accessible. In future work, we may adopt this more general verification strategy while at the same time extending DafnyMPI to support a larger subset of MPI primitives, particularly when it comes to collective communication operations. This work can further allow us to integrate DafnyMPI with existing scientific computing software such Finch [52], where the user may specify the parameters of a given problem and the parallelization is handled automatically.

5.2.3 Counterexample-Centric Verification

As a more speculative stretch goal for future research one could envision the development of a new verification-aware programming language that incorporates all of the ideas presented in this dissertation. Such a language could come with an equivalent of DafnyMPI built into it, thus incorporating light-weight concurrency support. To guard against proof brittleness, this new lan-

guage may enforce opaqueness of functions, as has been proposed for Dafny [82, 63]. At the same time, this hypothetical language could treat verifier counterexamples as first-class objects, where the counterexamples could be stored in variables and inspected programmatically, mimicking the role that reflection plays in other languages. This would allow the development of testing and synthesis tools such as DTest and Metamorph as applications written directly using the language rather than as standalone plugins. Simultaneously, the ability to inspect counterexamples and debug proof failures could improve the performance of LLM-driven synthesis tools compared to existing approaches for verification languages such as [85]. A natural framework for realizing the proposed language and the associated tooling may be Rosette [108]. Rosette is intended as a basis for designing other solver-aided languages, and it already treats solver counterexamples as first-class objects. Building on Rosette could make it possible to define a higher-level verification language that combines concurrency support in the style of DafnyMPI with test generation and synthesis capabilities similar to DTest and Metamorph.

This dissertation demonstrates that languages that support automated verification of sequential programs have the potential to expand this functionality to testing, synthesis, and concurrency. I hope that these results encourage broader adoption of verification as a key component of the software development process.

Bibliography

[1] Rosa Abbasi et al. “Deductive Verification of Floating-Point Java Programs in KeY”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2021, pp. 242–261. DOI: 10.1007/978-3-030-72013-1_13.

[2] Rajeev Alur et al. “Syntax-Guided Synthesis”. In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[3] Vytautas Astrauskas et al. “The Prusti Project: Formal Verification for Rust”. In: *NASA Formal Methods Symposium*. Springer. 2022, pp. 88–108. DOI: 10.1007/978-3-031-06773-0_5.

[4] AWS Cryptographic Material Providers Library. Accessed: 2025-02-27. 2025. URL: https://github.com/aws/aws-cryptographic-material-providers-library.

[5] AWS Encryption SDK. Accessed: 2025-02-27. 2025. URL: https://github.com/aws/aws-encryption-sdk-dafny.

[6] Diego de Azevedo Oliveira et al. “BTestBox: a Tool for Testing B Translators and Coverage of B Models”. In: *Tests and Proofs*. 2019, pp. 83–92. DOI: 10.1007/978-3-030-31157-5_6.

[7] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Computing Surveys* 51.3 (2018). DOI: 10.1145/3182657.

[8] Mike Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *International Symposium on Formal Methods for Components and Objects*. 2005, pp. 364–387. DOI: 10.1007/11804192_17.

[9] Patrick Baudin et al. *WP Plug-in Manual*. URL: https://frama-c.com/download/frama-c-wp-manual.pdf.

[10] Benedikt F. H. Becker, Cláudio Belo Lourenço, and Claude Marché. "Explaining Counterexamples with Giant-Step Assertion Checking". In: *Workshop on Formal Integrated Development Environment*. 2021, pp. 82–88. DOI: 10.4204/EPTCS.338.10.

[11] Robert Beers. "Pre-RTL Formal Verification: An Intel Experience". In: *Proceedings of the 45th Annual Design Automation Conference*. Anaheim, California, 2008, pp. 806–811. DOI: 10.1145/1391469.1391675. URL: https://doi.org/10.1145/1391469.1391675.

[12] D. Beyer et al. "Generating Tests from Counterexamples". In: *International Conference on Software Engineering*. 2004, pp. 326–335. DOI: 10.1109/ICSE.2004.1317455.

[13] Dirk Beyer and Marie-Christine Jakobs. "CoVeriTest: Cooperative Verifier-Based Testing." In: *Fundamental Approaches to Software Engineering*. 2019, pp. 389–408. DOI: 10.1007/978-3-030-16722-6_23.

[14] Allan Blanchard et al. "Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs". In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2016, pp. 67–72. DOI: 10.1109/SCAM.2016.18.

[15] Stefan Blom and Marieke Huisman. "The VerCors Tool for Verification of Concurrent Programs". In: *International Symposium on Formal Methods*. 2014, pp. 127–131. DOI: 10.1007/978-3-319-06410-9_9.

[16] Boogie. Accessed: 2025-02-27. 2025. URL: https://github.com/boogie-org/boogie.

[17] Pietro Braione et al. "Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs". In: *International Symposium on Software Testing and Analysis*. 2017, pp. 90–101. DOI: 10.1145/3092703.3092715.

[18] David Brandfonbrener et al. *Verified Multi-Step Synthesis using Large Language Models and Monte Carlo Tree Search*. 2024. arXiv: 2402.08147.

[19] Stephen Brookes and Peter W. O'Hearn. "Concurrent Separation Logic". In: *ACM SIGLOG News* 3 (2016), pp. 47–65. DOI: 10.1145/2984450.2984457. URL: https://doi.org/10.1145/2984450.2984457.

[20] Franck Cassez. "Verification of the Incremental Merkle Tree Algorithm with Dafny". In: *Formal Methods*. 2021, pp. 445–462. DOI: 10.1007/978-3-030-90870-6_24.

[21] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. "Deductive Verification of Smart Contracts with Dafny". In: *Formal Methods for Industrial Critical Systems*. 2022, pp. 50–66. DOI: 10.1007/978-3-031-15008-1_5.

[22] Franck Cassez et al. "Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny". In: *Formal Methods*. 2023, pp. 571–583. DOI: 10.1007/978-3-031-27481-7_32.

[23] Aleks Chakarov et al. "Formally Verified Cloud-Scale Authorization". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 2508–2521. DOI: 10.1109/ICSE55347.2025.00166.

[24] Aleksandar Chakarov et al. "Better Counterexamples for Dafny". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2022, pp. 404–411. DOI: 10.1007/978-3-030-99524-9_23.

[25] Zhenbang Chen et al. "MPI-SV: A Symbolic Verifier for MPI Programs". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. Seoul, South Korea, 2020, pp. 93–96. DOI: 10.1145/3377812.3382144. URL: https://doi.org/10.1145/3377812.3382144.

[26] Maria Christakis et al. "Integrated Environment for Diagnosing Verification Errors". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2016, pp. 424–441. DOI: 10.1007/978-3-662-49674-9_25.

[27] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *International Conference on Functional Programming*. 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[28] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2.

[29] *Coverlet*. URL: https://github.com/coverlet-coverage/coverlet.

[30] Pascal Cuoq et al. “Frama-C: A Software Analysis Perspective”. In: *International conference on software engineering and formal methods*. Springer. 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16.

[31] Dafny. Accessed: 2025-02-27. 2025. URL: https://github.com/dafny-lang/dafny.

[32] Markus Dahlweid et al. “VCC: Contract-Based Modular Verification of Concurrent C”. In: *2009 31st International Conference on Software Engineering-Companion Volume*. 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.

[33] Richard A De Millo, Richard J Lipton, and Alan J Perlis. “Social Processes and Proofs of Theorems and Programs”. In: *Communications of the ACM* 22.5 (1979), pp. 271–280. DOI: 10.1145/359104.359106.

[34] Antoine Delignat-Lavaud et al. “A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1162–1178. DOI: 10.1109/SP40001.2021.00039.

[35] Marco Eilers and Peter Müller. “Nagini: a Static Verifier for Python”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33.

[36] Aleksandr Fedchin, Alexander Y Bai, and Jeffrey S Foster. “Metamorph: Synthesizing Large Objects from Dafny Specifications”. In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1 (2025), pp. 759–785. DOI: 10.1145/3720448.

[37] Aleksandr Fedchin et al. “A Toolkit for Automated Testing of Dafny”. In: *NASA Formal Methods*. 2023, pp. 397–413. DOI: 10.1007/978-3-031-33170-1_24.

[38] Aleksandr Fedchin et al. “DafnyMPI: A Dafny Library for Verifying Message-Passing Concurrent Programs”. In: *Proceedings of the ACM on Programming Languages* 10.POPL (2026). DOI: 10.1145/3776705.

[39] Yu Feng et al. “Component-Based Synthesis for Complex APIs”. In: *Symposium on Principles of Programming Languages*. 2017, pp. 599–612. DOI: 10.1145/3009837.3009851.

[40] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3—–Where Programs Meet Provers”. In: *European Symposium on Programming*. 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[41] Vojtundefinedch Forejt et al. “Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs”. In: *ACM Trans. Program. Lang. Syst.* 39 (2017). DOI: 10.1145/3095075. URL: https://doi.org/10.1145/3095075.

[42] Stefan Forstenlechner et al. “A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming”. In: *Genetic Programming*. 2017, pp. 262–277. DOI: 10.1007/978-3-319-55696-3_17.

[43] Clément Fumex, Claude Marché, and Yannick Moy. “Automating the Verification of Floating-Point Programs”. In: *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9*. 2017, pp. 102–119. DOI: 10.1007/978-3-319-72308-2_7.

[44] Deepak Garg and Frank Pfenning. “Type-Directed Concurrency”. In: *International Conference on Concurrency Theory*. 2005, pp. 6–20. DOI: 10.1007/11539452_5.

[45] Samuel Jimenez Gil, Manuel I Capel, and Gabriel Olea Olea. “Automatic Test Cases Generation from Formal Contracts”. In: *Information and Software Technology* 172 (2024), p. 107467. DOI: 10.1016/j.infsof.2024.107467.

[46] Sumit Gulwani et al. “Synthesis of Loop-Free Programs”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011, pp. 62–73. DOI: 10.1145/1993498.1993506.

[47] Zheng Guo et al. “Type-Directed Program Synthesis for RESTful APIs”. In: *International Conference on Programming Language Design and Implementation*. 2022, pp. 122–136. DOI: 10.1145/3519939.3523450.

[48] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. “Absynthe: Abstract Interpretation-Guided Synthesis”. In: *Proceedings of the ACM on Programming Languages*. Vol. 7. PLDI. 2023, pp. 1584–1607. DOI: 10.1145/3591285. URL: https://doi.org/10.1145/3591285.

[49] Tihomir Gvero et al. “Complete Completion Using Types and Weights”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2013, pp. 27–38. DOI: 10.1145/2491956.2462192.

[50] Dan Hao et al. “On-Demand Test Suite Reduction”. In: *International Conference on Software Engineering*. 2012, pp. 738–748. DOI: 10.1109/ICSE.2012.6227144.

[51] Chris Hawblitzel et al. “Ironclad Apps: End-to-End Security via Automated Full-System Verification”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 165–181. DOI: 10.5555/2685048.2685062.

[52] Eric Heisler, Aadesh Deshmukh, and Hari Sundar. “Finch: Domain Specific Language and Code Generation for Finite Element and Finite Volume in Julia”. In: *Computational Science – ICCS 2022*. Ed. by Derek Groen et al. Cham: Springer International Publishing, 2022, pp. 118–132. ISBN: 978-3-031-08751-6. DOI: 10.1007/978-3-031-08751-6_9.

[53] Tobias Hilbrich et al. “MUST: A Scalable Approach to Runtime Error Detection in MPI Programs”. In: *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. 2010, pp. 53–66. DOI: 10.1007/978-3-642-11261-45.

[54] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. “Actris: Session-Type Based Reasoning in Separation Logic”. In: *POPL* 4 (2019), pp. 1–30. DOI: 10.1145/3373096.

[55] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 21 (1978), pp. 666–677. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[56] Kohei Honda et al. “Verification of MPI Programs Using Session Types”. In: *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*. Vienna, Austria, 2012, pp. 291–293. DOI: 10.1007/978-3-642-33518-1_37. URL: https://doi.org/10.1007/978-3-642-33518-1_37.

[57] Hwa-You Hsu and Alessandro Orso. “MINTS: A General Framework and Tool for Supporting Test-suite Minimization”. In: *International Conference on Software Engineering*. 2009, pp. 419–429. DOI: 10.1109/ICSE.2009.5070541.

[58] Large MPI Tags with the Intel MPI Library. Accessed: 2025-04-12. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/large-mpi-tags-with-the-intel-mpi.html.

[59] Ahmed Irfan et al. “Testing Dafny (Experience Paper)”. In: *International Symposium on Software Testing and Analysis*. 2022, pp. 556–567. DOI: 10.1145/3533767.3534382.

[60] Shachar Itzhaky et al. “Cyclic Program Synthesis”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 944–959. DOI: 10.1145/3453483.3454087.

[61] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. “Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing”. In: *POPL* 8 (2024). DOI: 10.1145/3632889. URL: https://doi.org/10.1145/3632889.

[62] Naman Jain et al. “Jigsaw: Large Language Models Meet Program Synthesis”. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1219–1231. DOI: 10.1145/3510003.3510203.

[63] Emina Torlak Joseph W. Cutler Michael Hicks. “Improving the Stability of Type Safety Proofs in Dafny”. In: *Dafny 2024 - POPL 2024*. 2024. URL: https://popl24.sigplan.org/details/dafny-2024-papers/3/Improving-the-Stability-of-Type-Safety-Proofs-in-Dafny.

[64] Ralf Jung et al. “Iris From the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic”. In: *Journal of Functional Programming* 28 (2018), pp. 1–73. DOI: 10.1017/S0956796818000151.

[65] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2003, pp. 553–568. DOI: 10.1007/3-540-36577-X_40.

[66] Jinwoo Kim et al. “Semantics-Guided Synthesis”. In: *Proceedings of the ACM on Programming Languages*. Vol. 5. POPL. 2021. DOI: 10.1145/3434311.

[67] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. “Asynchronous Liquid Separation Types”. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 2015, pp. 396–420. DOI: 10.4230/LIPIcs.ECOOP.2015.396.

[68] Naoki Kobayashi. “A New Type System for Deadlock-Free Processes”. In: *CONCUR 2006–Concurrency Theory: 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006. Proceedings 17*. 2006, pp. 233–247. DOI: 10.1007/11817949_16.

[69] Hugo Krawczyk and Pasi Eronen. *RFC 5869: HMAC-based Extract-and-Expand KDF (HKDF)*. https://datatracker.ietf.org/doc/html/rfc5869. Accessed: 2024-10-15. 2010.

[70] Leslie Lamport. “The Temporal Logic of Actions”. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), pp. 872–923. DOI: 10.1145/177492.177726. URL: https://doi.org/10.1145/177492.177726.

[71] Andrea Lattuada et al. “Verus: Verifying Rust Programs Using Linear Ghost Types”. In: *Proceedings of the ACM on Programming Languages*. Vol. 7. OOPSLA1. ACM New York, NY, USA, 2023, pp. 286–315. DOI: 10.1145/3586037.

[72] K Rustan M Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[73] K Rustan M Leino. “Accessible Software Verification with Dafny”. In: *IEEE Software* 34.6 (2017), pp. 94–97. DOI: 10.1109/MS.2017.4121212.

[74] K Rustan M Leino, Peter Müller, and Jan Smans. “Verification of Concurrent Programs with Chalice”. In: *International School on Foundations of Security Analysis and Design*. Springer, 2007, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7.

[75] K. Rustan M. Leino and Aleksandar Milicevic. “Program Extrapolation with Jennisys”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 2012, pp. 411–430. DOI: 10.1145/2384616.2384646.

[76] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[77] *Dafny utilities library*. URL: https://github.com/dafny-lang/libraries.

[78] Daniel Liew, Cristian Cadar, and Alastair F Donaldson. “Symbooglix: A Symbolic Execution Engine for Boogie Programs”. In: *International Conference on Software Testing, Verification and Validation*. 2016, pp. 45–56. DOI: 10.1109/ICST.2016.11.

[79] Ziqing Luo, Manchun Zheng, and Stephen F Siegel. “Verification of MPI programs using CIVL”. In: *Proceedings of the 24th European MPI Users’ Group Meeting*. 2017, pp. 1–11. DOI: 10.1145/3127024.3127032.

[80] Valentin JM Manès et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[81] Hugo Rafael Fecha Martins. “Automated Program Repair of Arithmetic Programs in Dafny: Repairing Simple Arithmetic Programs”. MA thesis. Instituto Superior Técnico, 2022.

[82] Sean McLaughlin et al. “Enhancing Proof Stability”. In: *Dafny 2024 - POPL 2024*. 2024. URL: https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability.

[83] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.0*. Accessed: 2025-05-04. 1994. URL: https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps.

[84] Ashish Mishra and Suresh Jagannathan. “Specification-Guided Component-Based Synthesis from Effectful Libraries”. In: *Proceedings of the ACM on Programming Languages*. Vol. 6. OOPSLA2. ACM New York, NY, USA, 2022, pp. 616–645. DOI: 10.1145/3563310.

[85] Md Rakib Hossain Misu et al. “Towards AI-Assisted Synthesis of Verified Dafny Methods”. In: *Proceedings of the ACM on Software Engineering*. Vol. 1. FSE. 2024, pp. 812–835. DOI: 10.1145/3643763. URL: https://doi.org/10.1145/3643763.

[86] *Mockito*. URL: https://github.com/mockito/mockito.

[87] *Moq*. URL: https://github.com/moq/moq.

[88] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[89] Peter Müller, Malte Schwerhoff, and Alexander J Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *International conference on verification, model checking, and abstract interpretation*. Springer. 2015, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[90] Daye Nam et al. “Predictive Synthesis of API-Centric Code”. In: *International Symposium on Machine Programming*. 2022, pp. 40–49. DOI: 10.1145/3520312.3534866.

[91] Artifact for Paper "Metamorph: Synthesizing Large Objects from Dafny Specifications". Accessed: 2025-02-27. 2025. DOI: 10.5281/zenodo.14925936.

[92] David J. Pearce. *Formalising a Simple Virtual Machine*. Accessed: 2024-10-15. 2022. URL: https://whileydave.com/2022/06/28/formalising-a-simple-virtual-machine/.

[93] Salman Pervez et al. "Practical Model-Checking Method for Verifying Correctness of MPI Programs". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting*. 2007, pp. 344–353. DOI: 10.1007/978-3-540-75416-9_46.

[94] Guillaume Petiot et al. "How Testing Helps to Diagnose Proof Failures". In: *Formal Aspects of Computing* 30.6 (2018), pp. 629–657. DOI: 10.1007/s00165-018-0456-4.

[95] Nadia Polikarpova, Carlo A Furia, and Scott West. "To Run What No One Has Run Before: Executing an Intermediate Verification Language". In: *International Conference on Runtime Verification*. 2013, pp. 251–268. DOI: 10.1007/978-3-642-40787-1_15.

[96] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program Synthesis from Polymorphic Refinement Types". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 522–538. DOI: 10.1145/2908080.2908093.

[97] Nadia Polikarpova and Ilya Sergey. "Structuring the Synthesis of Heap-manipulating Programs". In: *Proceedings of the ACM on Programming Languages*. Vol. 3. POPL. ACM New York, NY, USA, 2019, pp. 1–30.

[98] Kia Rahmani et al. "Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis". In: *Proceedings of the ACM on Programming Languages*. Vol. 5. OOPSLA. 2021, pp. 1–29. DOI: 10.1145/3485535.

[99] Alastair Reid et al. "Towards Making Formal Methods Normal: Meeting Developers Where They Are". In: *arXiv preprint arXiv:2010.16345* (2020). DOI: 10.48550/arXiv.2010.16345.

[100] Stephen F Siegel and Timothy K Zirkel. "Automatic Formal Verification of MPI-based Parallel Programs". In: *ACM Sigplan Notices* 46 (2011), pp. 309–310. DOI: 10.1145/1941553.1941603.

[101] Graeme Smith. "A Dafny-Based Approach to Thread-Local Information Flow Analysis". In: *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 2023, pp. 86–96. DOI: 10.1109/FormaliSE58978.2023.00017.

[102] Armando Solar-Lezama et al. "Combinatorial Sketching for Finite Programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2006, pp. 404–415. DOI: 10.1145/1168857.1168907.

[103] Patrick Spettel. "Delfy: Dynamic Test Generation for Dafny". MA thesis. Eidgenössische Technische Hochschule Zürich, 2013. DOI: 10.3929/ethz-a-010056933.

[104] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. "From Program Verification to Program Synthesis". In: *Proceedings of the ACM on Programming Languages*. POPL. 2010, pp. 313–326. DOI: 10.1145/1706299.1706337.

[105] Gilbert Strang. *Differential Equations and Linear Algebra*. Wellesley-Cambridge Press, 2014. DOI: 10.1137/1.9780980232790. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780980232790.

[106] Chuyue Sun et al. "Clover: Closed-Loop Verifiable Code Generation". In: *AI Verification*. 2024, pp. 134–155. DOI: 10.1007/978-3-031-65112-0_7.

[107] Nikhil Swamy et al. "SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs". In: *POPL* 4 (2020). DOI: 10.1145/3409003. URL: https://doi.org/10.1145/3409003.

[108] Emina Torlak and Rastislav Bodik. "Growing Solver-aided Languages with Rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 135–152. DOI: 10.1145/2509578.2509586.

[109] Sarvani Vakkalanka et al. “Implementing Efficient Dynamic Formal Verification Methods for MPI Programs”. In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. 2008, pp. 248–256. DOI: 10.1007/978-3-540-87475-1_34.

[110] Sarvani S. Vakkalanka et al. “ISP: A Tool for Model Checking MPI Programs”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT, USA, 2008, pp. 285–286. DOI: 10.1145/1345206.1345258. URL: https://doi.org/10.1145/1345206.1345258.

[111] Eric Verhulst et al. “Formal Development of a Network-Centric RTOS”. In: Springer, 2011. DOI: 10.1007/978-1-4419-9736-4. URL: https://api.semanticscholar.org/CorpusID:57903213.

[112] Anh Vo et al. “Formal Verification of Practical MPI Programs”. In: *ACM Sigplan Notices* 44 (2009), pp. 261–270. DOI: 10.1145/1594835.1504214.

[113] Anh Vo et al. “A Scalable and Distributed Dynamic Formal Verifier for MPI Programs”. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–10. DOI: 10.1109/SC.2010.7. URL: https://doi.org/10.1109/SC.2010.7.

[114] Changjing Wang et al. “A Method to Deduce and Synthesize the Dafny Programs”. In: *Wuhan Univ. J. Nat. Sci.* 26.6 (2021), pp. 481–488. DOI: 10.1051/wujns/2021266481.

[115] Nicky Williams et al. “PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis”. In: *European Dependable Computing Conference*. Springer. 2005, pp. 281–292. DOI: 10.1007/11408901_21.

[116] Felix A Wolf et al. “Gobra: Modular Specification and Verification of Go Programs”. In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17.

[117] *XUnit*. URL: https://github.com/xunit/xunit.

[118] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[119] Z3. Accessed: 2025-02-27. 2025. URL: https://github.com/Z3Prover/z3.

[120] Yunfan Zhang et al. “Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2023, pp. 1–12. DOI: 10.1145/3551349.3556951.