DAFINITE: Finitization of Dafny Protocols

Justin T. Beemer
University of Michigan
jubeemer@umich.edu

Zachary C. Carey
University of Michigan
zccarey@umich.edu

Benjamin D. Manley
University of Michigan
bdmanley@umich.edu

Abstract

Formal verification is a promising field of open research which has potential to change the way we build distributed systems. It does not eliminate bugs from system design, but rather reduces the scope of potentially buggy code from the entire implementation to a small declarative description of the protocol. Debugging techniques are thus still required to catch bugs in protocol descriptions. Model checking is one approach to verifying that protocols guarantee certain properties, but is limited by state-space explosion for protocols with infinite domains.

As a step toward overcoming this limitation, we have implemented Dafinite: a software prototype which generates a finitized state transition system from a protocol description written in a subset of the Dafny programming language. We have tested Dafinite by implementing buggy descriptions for two distributed protocols and using a model checker to generate counterexamples that demonstrate violations of the systems' safety properties.

1 Introduction

The design and implementation of distributed systems is notoriously difficult. Even with the most experienced developers and testers, subtle bugs can go undetected and affect systems in production. This has led to an increased interest in the *formal verification* of distributed protocols and their implementations. With formal verification, developers write a *protocol description* for their system in a language like Dafny [7], and automated tools and theorem provers determine whether or not a corresponding *implementation* of the system satisfies the protocol [5]. With a formally verified program, a developer may be confident that their system implements their protocol correctly even before the code is run for the first time.

Despite its benefits, formal verification is not a magic bullet that eliminates bugs from the system development process. Rather, it significantly reduces the amount of code which a developer must check for bugs. Instead of needing to inspect and test thousands of lines of systems code, the developer must only look for errors in the protocol specification, which is often multiple orders of magnitude smaller than the corresponding implementation.

The question remains, then, how to debug protocols which do not actually describe a system where the desired properties hold. A powerful candidate for this process is model checking. Model checkers can take as input a description of a state transition system and a set of properties to check. Using powerful tools such as Satisfiability Modulo Theories (SMT) solvers, the model checker can either prove or disprove that the desired properties hold in the system. In the event of a disproof, they often output a sequence of state transitions which demonstrate a violation of a given property, called a *counterexample*.

A fundamental limitation of this approach is the exponential growth of the state space as a system scales. It is not computationally feasible for a model checker to prove properties on systems with infinite domains, which many distributed protocols have. Consider a relatively simple lock server distributed system, where each server has a single semaphore and may establish a "link" with exactly one client at a time (the *safety property*). The protocol description permits an arbitrary (even infinite) set of client and server nodes, making it impossible for a model checker to verify the system's safety property in the general case.

However, consider a finite instance of the lock server system described above with a single server and two clients. This is the smallest instance of the system which could meaningfully violate the safety property, and is sufficiently small for a model checker to verify. If there was a bug in the Dafny code and the model checker detected a resulting violation of the safety property in the finite instance, the counterexample would be extremely useful for understanding the problem and debugging the protocol.

As a first step in the direction of enabling the use of model checkers for debugging distributed protocol descriptions written in Dafny, we have implemented Dafinite: a prototype of a finitization module within the Dafny compiler which gener-

ates a finitized version of the state transition system in VMT format [3]. The module currently supports a limited subset of Dafny language features, including the membership operation for sets of user-defined datatypes. To demonstrate the value of such a tool, we wrote descriptions of two simple distributed protocols in Dafny (the lock server described above and two-phase commit), inserted various bugs, and used the Averroes model checker [6] to generate counterexamples which demonstrated violations of the safety properties caused by the added bugs.

2 Related Work

IronFleet [5] represents the current state-of-the-art methodology for formal verification of large-scale distributed system implementations. In the IronFleet approach, the implementation and proof of a system is divided into layers: the *specification layer*, a high-level state machine which describes all possible allowable behaviors of the system; the *protocol layer*, which introduces the concept of independent hosts communicating via exchange of messages; and the *implementation layer*, an imperative program which implements the protocol. Verification proceeds by proving that the protocol correctly refines the specification and the implementation correctly refines the protocol. Dafinite generates a finite instance of a protocol layer state machine.

I4 [8] makes the use of formal verification more accessible by automating the derivation of an *inductive invariant* for a system. The inductive invariant is a property that must be true under all states of the system and is closed under a transition relation. It is a critical part of the verification step of IronFleet. Finding this invariant is a challenging task which typically requires expertise and a deep understanding of the protocol. I4's key observation is that as the size of a system grows, its inductive invariant grows in size but not generally in complexity. As such, it generates a finite instance of a protocol description written in IVy [9], uses a model checker to find an inductive invariant for the finite instance, and performs a series of iterative steps to attempt to generalize the invariant to an infinite system. I4's translation script for finitizing IVy protocols served as an important starting point for our understanding and approach.

The VMT format [3] is an extension of SMT-LIBv2 [1], a library for standardizing theories and input/output languages for SMT solvers. It is designed to support representation of symbolic state transition systems. State variables have two representations: their "current" and "next" values. Both are created via a `declare-fun` and linked with a special annotation `:next`. Their initial values may be constrained in a special predicate function describing the allowable initial states of the system, annotated with `:init`. The state transition function is specified by annotating a predicate with `:trans`. Safety properties for the system, which can be verified by model checkers as described above, are special predicates annotated with `:invar-property`. Support for Linear Temporal Logic (LTL) liveness properties also exists in VMT, though our prototype only permits users to specify safety properties in their Dafny protocols.

3 Design

Dafinite's design closely resembles the IVy finitizer in I4 [8] and reuses logic from it where applicable.

Dafny is a Turing complete language, capable of representing arbitrary real-world distributed protocol implementations. In its current state, Dafinite imposes strict semantic and stylistic constraints on the Dafny files it can finitize. The enforced style follows a similar style presented for the specification and protocol layers of the IronFleet methodology [5]. Use of predetermined style and naming conventions simplified iteration over the program's Abstract Syntax Tree (AST) and enabled us to design and implement a prototype that works end-to-end for a subset of Dafny's language features.

3.1 Input Constraints

In the protocol being finitized, each object which has a potentially infinite domain (e.g., clients and servers in the lock server protocol) must be represented as a Dafny `datatype`. The state of the system consists of the members of these datatypes (e.g. `Server.semaphore`, of type `bool`), as well as a special "`id`" field used to distinguish instances of each object. A special, all-encompassing datatype called `DafnyState` must be defined, and its members must be the arbitrarily-sized sets of the datatypes to be finitized (e.g., `DafnyState.clients`, of type `set<Client>`).

The possible state transitions must be described by Dafny predicates (single-expression boolean functions) whose names have the prefix "Action" and take two `DafnyState` arguments corresponding to the system state before and after the transition. If and only if the predicate is true, then it is possible for the state machine to transition from the previous state to the next state via that action. Listing 1 provides some example pseudocode for a connect "event" in the lock server example.

A few special Dafny predicates must also be defined. `Init` must specify the restrictions on the initial state of the system. `Next` must be a disjunction of all of the Action predicates, thereby forming a single predicate representing all possible state transitions. `Safety` must specify the system's desired safety properties, whose unconditional truth will become the goal of the formal verification.

Lastly, while support for arbitrary Dafny expressions and types (including sequences and maps) are the goal for this tool, we could only implement a subset of language features due to time limitations. As we demonstrate in our evaluation, this subset is sufficient for implementing some basic protocols, but more flexible support and fewer restrictions on Dafny

developers are desirable. A more extended discussion of the current features and limitations of Dafinite are included in Section 6 and our style guide.

```
// Example instance declaration
datatype Server=(semaphore)
datatype Client=(connectedServers)
datatype DafnyState=(clients, servers)

// Example transition for system
ActionConnect(prev_state, next_state)
{
    exists client c and server s such that
        in prev_state:
            s.semaphore is true
        and
        in next_state:
            s.semaphore is false
            and
            s in c.connectedServers
}
```

Listing 1: Pseudocode for the lock server, including type declarations and an example transition.

3.2 Finitization

In the invocation of Dafinite, just as in the invocation of I4's IVy finitization, the user must provide a finite instance count for each of the defined datatypes in the Dafny protocol (e.g. "`Client=2,Server=1`"). These finitized-domain sizes are matched with the corresponding datatypes parsed out of the Dafny file to instantiate the appropriate number of each datatype. Dafinite then parses the `Init`, `Next`, `Safety`, and each of the action predicates, recursively breaking down complex expressions and outputting the equivalent logic in VMT.

Universal quantifiers (`forall`) are replaced with an appropriate conjunction over each instance of the finitized domain, and existential quantifiers (`exists`) with an appropriate disjunction. Taking the lock server with two clients and one server as an example, consider the logical statement "there exists some client c and server s such that c holds the semaphore of s." The presence of the quantifier introduces uncertainty that may make the problem undecidable for a solver. However, in the finitized version of the protocol, this logic can be expressed in decidable first-order logic as $s_0 \in c_0.connectedServers \vee s_0 \in c_1.connectedServers$.

To implement finite sets of user-defined datatypes, we instantiate a boolean state variable to represent each object's membership in its set. For example, in a finite instance of the lock server protocol with n servers, each client would have n state booleans indicating membership of each server in its set of connected servers. This design leads to some difficulty supporting more complex set operations like subset, superset, union, intersection, and difference, and our prototype stops short of this. Our initial goal was to use the array theory supported by VMT and Averroes' underlying SMT solver (MathSAT5 [2]) to represent sets, using finitized datatypes as array indices and booleans as values to represent membership, but we could not determine how to properly constrain the initial values of state arrays in VMT. We thus opted for our current solution as a means to support set membership, as we found it to be a critical language feature for implementing basic protocol descriptions. We encourage future work to revisit this decision and investigate methods for constraining the initial values of arrays, as we believe array theory would greatly facilitate support for additional set operations, as well as other datatypes like sequences and maps.

4 Implementation

Dafinite is implemented with around 700 lines of C# code integrated into the Dafny compilation and verification codebase. This allows it to reside in the same C# namespace as the Dafny parser, giving direct access to a program's AST. Invoking Dafinite is done by simply adding a few arguments to the existing command line call used to compile and verify a Dafny file. For example, compiling and verifying a lock server protocol in lock_server.dfy is done by calling

```
dafny lock_server.dfy
```

and creating a finitized version with one server and two clients and printing to lock_server.vmt is done with

```
dafny lock_server.dfy /vprint lock_server.vmt
/finitize:Server=1,Client=2
```

5 Evaluation

We implemented two distributed system protocols in Dafny: the simple lock server described above and two-phase commit. These protocols were implemented in the format described previously, which we found to work well. For each of the tests, we used Dafinite to finitize the protocol, then used the model checker Averroes [4, 6] to determine if the finite instance of the protocol violates the system's desired safety properties. The following sections describe each of the protocols, as well as bugs we injected to test the model checker's utility as a protocol debugger, in more detail.

5.1 Lock Server

The lock server system consists of clients and servers. Servers maintain a boolean representing the state of a semaphore (lock), and clients maintain the set of servers whose semaphores they "hold". The safety property for this system is "no two clients can have a link to (i.e., hold the lock of) the same server at the same time" [8]. We implemented a description of this protocol in 80 lines of Dafny. Listing 1 includes some pseudocode for the lock server protocol.

We used Averroes to confirm that the safety property held for small instances of the protocol. After verifying the correctness of our protocol description, we added bugs to the Dafny code and checked that (i) Averroes correctly determined that the safety property could be violated and (ii) the counterexample it produced could be used to understand why the protocol was buggy. For the lock server, we performed this test with two separate bugs:

1. We removed the requirement for a `Connect` action that the semaphore of the server being connected to is not held in the starting state. Removal of this requirement allows a client to connect to a server while its semaphore is held by another client, which violates the safety property.

2. We removed the requirement for a `Disconnect` action that the involved client and server are connected in the starting state. Removal of this requirement allows clients to release a semaphore they did not hold. This can lead to a violation of the safety property if a client incorrectly "releases" a server's semaphore while it is held by another client, and another client subsequently connects to that server.

Averroes was able to detect that both of these changes could lead to a violation of the safety property, and in each case provided a sequence of state transitions that demonstrated how the safety property could be violated. Listing 2 provides an example of such an execution when bug 1 is present. (Note that the state variables `ClientX_connectedServers_ServerY` correspond to the booleans representing set membership as explained in Section 3).

```
initial state:
    Server0_semaphore=true
    Client0_connectedServers_Server0=false
    Client1_connectedServers_Server0=false

    Inputs transformed:
        action -> connect
state:
    Server0_semaphore=false
    Client0_connectedServers_Server0=false
    Client1_connectedServers_Server0=true

    Inputs transformed:
        action -> connect
state:
    Server0_semaphore=false
    Client0_connectedServers_Server0=true
    Client1_connectedServers_Server0=true

safety property violated
```

Listing 2: Model checker simulation of a lock server system with two clients and one server after inserting bug 1 into the protocol description.

5.2 Two-Phase Commit

Two-phase commit is an atomic commit protocol where a collection of processes must come to a unanimous decision to commit or abort. The system consists of a *coordinator* process (omniscient in our protocol) and a collection of voting processes (*participants*). The participants each send a vote to either commit or abort to the coordinator, and the coordinator decides based on the votes and sends its decision to the participants. The safety properties for this system are as follows: (i) all processes reach the same decision, (ii) the decision cannot be reversed, (iii) the decision is to commit only if all processes vote yes, and (iv) the decision must be to commit if there are no failures and everyone votes to commit.

We implemented a description of this protocol in 250 lines of Dafny. As with the lock server, we used the model checker to confirm that the safety properties held for a finitized version of the protocol before injecting bugs and ensuring Averroes could produce a counterexample which would help debug the protocol. For two-phase commit, we added the following bugs to the Dafny:

1. We allowed processes to commit before receiving a decision from the coordinator. This can violate correctness because a voting process can prematurely commit before everyone has voted yes, meaning another process could have voted no and aborted.

2. We allowed processes to choose to either commit or abort after receiving a commit decision from the coordinator. This can violate correctness because after receiving a commit decision, nodes would have the right to choose the decision, meaning some nodes may commit and some may abort.

As before, Averroes successfully detected that these changes could lead to a violation of the safety properties and provided a sequence of state transitions that demonstrated such a case. We omit an example of the model checker output for this protocol due to space constraints.

5.3 Debugging with a Model Checker

Over the course of the project, we found utilizing the model checker as a protocol debugger to be an extremely valuable tool. The detailed simulations outputted by Averroes helped us to see exactly how safety properties may be violated in the presence of certain bugs. In fact, the model checker revealed to us that our initially "correct" Dafny description of the lock server protocol actually contained bug 2 described above! We were able to utilize the counterexample to quickly locate the bug and fix our protocol, demonstrating the utility of model checking as a debugging technique.

6 Future Work

While we have demonstrated the feasibility of finitizing Dafny protocols, work remains to develop Dafinite into a robust and flexible tool for supporting formal verification of distributed systems. Currently, Dafinite is restrictive in the required formatting of state machine descriptions. In particular, it requires a single, high-level, "omniscient" view of the system state (i.e., the `DafnyState` type) where the `Next` predicate is a disjunction of the action predicates. Other approaches to writing protocol descriptions are more common and provide real benefits to the underlying solvers and theorem provers, so increasing the flexibility of our tool would be a useful step in removing restrictions from developers who want to use it.

The most critical avenue of future work that should be prioritized is the expansion of Dafny language features supported by Dafinite. Currently, the tool is limited to relatively simple boolean logic and set membership for sets of user-defined objects finitized by the tool. As discussed in Section 3, we believe a move from enumerated boolean state variables to arrays would greatly facilitate the addition of support for additional set operations, sequences, and maps. The primary problem to solve in order to achieve this is properly constraining the initial values of state represented by arrays in the underlying VMT format. We believe support for some other important features, including linear integer arithmetic, would require substantially less work. We chose to omit it from our initial prototype solely due to time constraints and its absence from the protocols we used to evaluate our prototype. Dafinite also cannot currently parse non-predicate functions. A complete list of Dafinite's current capabilities and limitations is contained in the style guide in our repository.

Additionally, Dafinite does not currently include support for specifying liveness properties. We prioritized safety properties for our prototype due to time constraints, but VMT does support annotation of LTL properties (see Section 2) and we believe it may be feasible to use model checkers to verify them in a similar manner.

Finally, reducing redundant logic in the VMT written by the tool would greatly simplify the output file and improve readability. While this is not critical for the technical correctness of the tool, it may improve scalability and performance when generating larger protocol instances. Additionally, while VMT is not designed with human readability in mind, manually parsing the file can be a necessary evil when debugging changes to Dafinite.

7 Conclusion

In this paper we introduced Dafinite, a tool for generating finite instances of distributed protocols written in the Dafny programming language. These instances are of immense value as they mitigate state-space explosion and thus permit the use of model checkers to debug protocol descriptions. We implemented descriptions of two simple distributed protocols in Dafny and found that Dafinite, in combination with a model checker, was extremely valuable for identifying and fixing errors. Future expansion of Dafinite and support for additional Dafny language features would increase its value and make it an effective tool in the development of formally verified distributed systems.

Acknowledgments

We'd like to thank the following people for their help throughout the project:

- Professor Manos Kapritsos, for sharing his advice and knowledge of the state-of-the-art in formal verification of distributed systems
- Tony Zhang, for guiding us toward this project idea
- Haojun Ma, for helping us get started and for his previous work with finitization of IVy protocols
- Armin Vakil, for sharing his knowledge of the Dafny grammar and AST with us

Availability

Our code is publicly available and may be viewed at https://github.com/zccarey/dafinite

Division of Work

The bulk of the work conducted for this project was performed synchronously by the authors in a group setting, which allowed for even distribution of work. Justin researched the VMT format for representation of state transition systems, Zachary specialized in the recursive expansion and finitization of expressions in the Dafny AST, and Benjamin focused on tuning the Dafinite input constraints and parsing Dafny datatypes and special predicates. All contributed to writing the test Dafny protocols and this report.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[2] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[3] Alessandro Cimatti, Alberto Griggio, and Stefano Tonetta. The vmt-lib language and tools. *arXiv preprint arXiv:2109.12821*, 2021.

[4] Sakallah K. Goel, A. Averroes 2.0, 2021[Online]. Available at `http://www.github.com/aman-goel/avr` .

[5] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[6] Suho Lee and Karem A. Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 849–865, Cham, 2014. Springer International Publishing.

[7] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[8] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.

[9] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.