# Paxos in Dafny

Joakim Hagen

May 15, 2015

# Abstract

A stronger alternative to software unit testing is full verification. Verification of functional correctness for software often requires a formal specification written in a separate non-executable language describing the desired behavior of the code. We take a look at Dafny, a language with verification as a built in feature to executable code. In this language we will attempt to verify the consensus algorithm Paxos, a method of reaching agreements in distributed systems. An application demonstrating Paxos in action will be presented and features of the language and verification in general will be explored.

# Acknowledgements

I want to thank my adviser Hein Meling for presenting the interesting and explorative task of implementing Paxos in Dafny, initially geared towards a master thesis and for continuous support and guidance with good help from Tormod Erevik Lea.

My appreciation extends to the C# community for already having posed and answered all the questions that came up and to the user lexicalscope from CodePlex for confirming there is also some activity in the Dafny community.

# Contents

# 1

# Introduction

Verification of correct functionality is traditionally done by writing unit test code that accesses the code's methods and classes, creates dummy objects, throws it some data and then checks whatever is returned against static values. This will uncover most bugs and thus save a lot of time debugging, but more often than not, the product is released in a state in need of maintenance for a while regardless of prior effort.

Since unit testing procedures are written in tandem with the code to some degree it hands the developer's valuable mental resources another ball to juggle. Any minor change in code architecture will require similar changes to the test code. Unit testing is a time consuming task following the law of diminishing returns. At one point one will spend more time writing tests than what is saved of debugging time. If you desire to verify the behavior of all aspects of your code, expect spending more time writing tests than spent on the code being tested.

In an effort to automate this verification process Microsoft Research developed a new programming language called *Dafny*[1] and a corresponding program verifier for functional correctness. If used correctly, it provides a guarantee of internal consistency and bug free execution provided the operating system and hardware follows suit.

A category of algorithms called *consensus algorithms* are notorious for the difficulty of implementing them correctly. *Paxos* (Lamport, 2001)[2] is the name of the one we will explore. Consensus algorithms provide a protocol for agreeing on a common value amongst a group of agents, these agents often being a collection of servers working concurrently in a method called *state machine replication* and these distributed systems require some synchronization. This is done primarily in order to ensure redundancy in case of failure of a machine for critically important services.

While the servers or the network upon which they rely may fail, the consensus algorithms themselves generally don't have any failure tolerance. Any loss of consensus might be irreversible unless specifically accounted for. Especially so if the system is allowed to run out of order as if nothing had happened. So the proof of its correctness is at least as important as the implementation itself. No amount of testing can provide that degree of confidence. Herein lies the motivation for this project. We will attempt to implement Paxos in Dafny with the hope of verifying its correctness.

Although Paxos has previously been verified to be correct, this proof has always been second hand to the program implementation and written in a mathematical language with no intent of execution. An implementation of Paxos written in *Event ML*[3] has been verified by extracting a programmability lemma in *Nuprl*[4] terms, a logic based support language. But in Dafny this is the first attempt at a proof through executable code.

# 2

# Paxos

In this chapter an outline of how the Paxos algorithm works will be presented. As it is known as a difficult to implement algorithm, much is likely due to the difficulty in understanding and explaining it. If not for the "hows" then definitely for the "whys". Consensus algorithms deal with multiple agents, and in Paxos' case these will additionally play one or more completely different roles. What we are looking for is emergent properties of a distributed system where no single node requires access to all the data, yet may still infer and conclude that other agents will agree with its results.

Concepts will be introduced as they are required. This is to hopefully reduce the timespan a relevant concept requires attention. A system where a holistic perspective is crucial to its understanding may lose important meaning if unnecessarily modularized. Also, to stay somewhat aligned with Dafny's verification syntax which will be explored later, we will explain with the help of annotations presenting restrictions (in orange) and conclusions (in blue).

In the Paxos algorithm, an agent performs one or more of these three roles;

**Proposer:**
- Responsible for initiating an instance where consensus is sought.

**Acceptor:**
- Decides which proposals to accept

**Learner:**
- Detects consensus when a majority of acceptors agree.

Paxos can handle a simultaneous failure of any $\lceil N/2 \rceil - 1$ agents, where $N$ is the total number of agents, if all agents perform all three roles.

In communication amongst each agent, two variables are exchanged; *Round* and *value*. Round is a unique integer that can be considered an identification of a specific proposal and value is any type or object that we would like to achieve consensus about. Our first important restriction becomes…

> *All proposers starts with unique rounds*

Because a unique round means that we know its corresponding value comes from the same proposer, we can guarantee that…

> *Where we find equal rounds we also have equal values*

## Phase 1

Starting an instance of Paxos, a proposer picks a round and the value it intends to issue. The round might be the agents unique ID or a previously agreed upon starting number. The value might be a reference to a task to perform from a queue. Then, with that data attached…

**Proposer broadcasts a "Prepare" message to all Acceptors**

Acceptors are completely independent of each other. The receiving acceptor's job is now to inform the proposer of a previously accepted round and value, if any, and promise to ignore any future messages with round less than that provided by the proposer. Let's phrase that as a restriction.

> *An acceptor will ignore a message where provided round is less than promised round*

Do note that a promise does not imply acceptance, and they should not be confused with one another. We will be asked to accept a round and value at a later point. If no earlier promise of a higher round has been made…

**Acceptor answers Proposer with a "Promise" message**

If, however, an acceptor has promised a higher round, we will never hear from it. We can on the basis of our above restriction thus say for the proposer…

> *All encountered accepted rounds, if any, will be less than our issued round*

They cannot be equal since we are using a unique number. At this point we need to introduce another restriction.

> *A proposer will only ask to accept if a majority of acceptors have promised/are prepared*

Any acceptor having accepted a round then implies that the majority of all acceptors were prepared for that round and the proposer issuing that round received those promises. Assuming our proposer have received promises from a majority of acceptors, we know that we would have at least one of those acceptors in common with any previous majority groups. That is a mathematical inevitability. We also know that our proposed round is greater than any previously accepted rounds. If not, we would never hear from the majority and thus never achieve one of our own. We can now, for all acceptors, promised or not, guarantee the following statement is true.

> *No acceptor has accepted any round equal to or larger than our proposed round*

Paxos is often thought of as divided into two phases. With this, phase 1 is finished. A majority of acceptors are receptive to our proposal, even if some acceptors never get a message or ignores it. We have blocked the possibility of a lesser round being accepted while we are proposing ours.

**Optional optimization:** For reduced bandwidth use, several instances of Paxos may share this first phase's message exchange. The proposer may send a single prepare message with a list of values. To allow for multiple instances, one needs to include a unique and static identifier for each instance. This is called a *slot*. Rounds need only be unique within one slot, so the list may share a single round. The receiving acceptors will now treat each slot independently yet answer with a single promise message. This is called *Multi-Paxos*, but what we will be using is also called *Single-Paxos* as a clarification on this matter.

## Phase 2

**Proposer broadcasts an "Accept" message to all Acceptors**

If the acceptors have not made a new promise with, and/or accepted a higher round to another proposer since our request, they will accept our round and value. Any answers to subsequent requests will contain that accepted information. Upon accepting a new round…

**Acceptor broadcasts a "Learn" message to all Learners**

The learner will keep track of which acceptors have accepted the value of the highest round. Note that none other than the highest accepted round will ever reach a consensus. Once a majority is in agreement, the value is marked as learned and a consensus has been attained. There is still a flaw that must be addressed with another restriction however.

> *A proposer must re-issue the highest encountered accepted round's corresponding value, if any.*

We need our proposer to remember the response of the acceptor with the highest accepted round. Consider that a consensus might be reached through a lower round than ours before we receive a majority of promises. Then that accepted round and value will surely be provided by at least one of the acceptors we have in common. If we were to issue our own value at this point, the higher round would override the previous accepted value and the consensus would be lost. We can, considering this restriction, then assert that…

> *Once a consensus is reached, the learned value cannot change.*

## Notes

Paxos does not handle byzantine failures or corrupted messages and does not guarantee liveness without modification. It can, and is supposed to, handle loss or duplication of messages and loss of some replica servers. The active proposer is called a *leader* and often acts without interference from other proposals. Although Paxos will not fail to learn a value only when consensus is reached, stagnation may still occur between several proposers believing to be the leader. This may happen if multiple proposers alternate finishing phase 1, effectively blocking the completion of phase 2. We will not attempt to prove liveness in this project.

# 3

# Dafny

It is not the intention of this chapter to teach the reader enough about the language to use it, but enough to understand some of the code presented and what the language is capable of. The tutorials and documentation[5] provided by Microsoft is, despite being the best source, incomplete. We will not assume the reader has read any, but it is recommended and we do not intend to replicate it. Some of the basics will be covered, just the bare minimum to understand the material that's not found in the official documentation.

Dafny builds on *Boogie*[6], an intermediate verification language, intended as a layer on which to build program verifiers for other languages. The Boogie language is used as a step in the compilation process before it is sent to the tool carrying the same name in order to generate verification conditions. These conditions are then sent to a *satisfiability modulo theory* (SMT) solver called *Z3*[7], all of which are also products by Microsoft Research.

The *Visual Studio*[11] plugin that is provided with the Dafny compiler provides syntax-highlighting and verification while typing. The verifier is on by default, but may be turned off. This is useful when your projects get large enough that verification takes 30 seconds or more to complete. Given that the call to the verifier blocks the thread responsible for Visual Studio's coding interface, delay between every minor change makes for a less than pristine development environment.

One of the main goals of Dafny is to eliminate the need for unit testing and debugging by providing a guarantee that the code functions according to the guidelines of the developer.

Dafny is an object oriented language, but with some limitations.

Dafny does -
- support inductive and co-inductive datatypes.
- support generics.
- support traits.

Dafny does **not** -
- have any standard libraries.
- interact with the OS in other ways than print.
- accept arguments to Main.
- support concurrent programming.
- support inheritance / subclassing or nested classes.

Since no arguments can be sent to Main, Dafny code is compiled as a library and used by a 'client application'. This application is not verified together with the library and is responsible for following the restrictions required.

## Methods

The general form of a method in Dafny has three parts. Firstly the method identification, which includes the name, comma delimited arguments and an optional `returns` part specifying pre-allocated returned values. The second is a set of optional (to some degree) annotations specifying statements that must be true for the program to verify correctly and the scope of operation.

```
method name(arg_name: type/class)
    returns (ret_name: type/class)
  requires  boolean_expression;
  modifies  set_of_objects;
  decreases integer_expression;
  ensures   boolean_expression;
{
  // conventional code
  assert boolean_expression;
}
```

Methods have 5 possible annotations: `assert`, `requires`, `ensures`, `modifies` and `decreases`. The `assert`, `requires` and `ensures` clauses are Boolean expressions, while `modifies` takes a set of objects and `decreases` takes a numerical expression useful to guarantee bounds within loop constructs. It is not used in this project and you may refer to rise4fun for further explanation. No annotations have any functional effect on the compiled product of your code.

The `requires` checks the state of its expression (precondition) where the method is called. If you attempt to call this method without satisfying the requirements, the verification fails. The `ensures` checks at the point of returning. If this (postcondition) fails, it is either because the expression is incorrect, the code is incorrect or you need a more descriptive precondition in the failing method or in one that's called. The `modifies` set of objects, is the limited scope of what is allowed to be written/changed. All prior collected verification knowledge about these objects and all their types are discarded when the method is called.

An `assert` is put inside the code block as a statement and is very handy for locating problems in your code when the problem is a failed postcondition. Put one or more in between your lines and see where the first assert fails. This annotation is not necessary for any form of verification.

Sometimes some of the annotations may not be necessary. If the method only reads and returns then `modifies` is not necessary. If no objects are referenced, there might not be any initial `requires`, and `ensures` is mostly for the sake of the code where the method is used.

The same annotations can be used multiple times and all expressions must be satisfied.

```
requires boolean_expression_1;
requires boolean_expression_2;
```

The above two annotations are equivalent to the one below.

```
requires boolean_expression_1
  && boolean_expression_2;
```

## Predicates

When expressions become big and redundant Dafny provides a refactoring solution in the form of a *predicate*.

```
predicate name(arg_name: type/class)
  requires  boolean_expression;
  reads     set_of_objects;
  ensures   boolean_expression;
{
  boolean_expression
}
```

Predicates looks like methods but their body is a single Boolean expression without semicolon termination. They have no `decreases` annotation use a `reads` instead of `modifies`. A predicate cannot modify, and needs its scope of readable objects to be specified explicitly. Predicates can be called by other annotations, which allows for refactoring of similar code.

## Ghosts

Sometimes dummy objects or other form of testing data is required in order to successfully verify some code. To this end, *ghost variables* serve its purpose. Ghost variables never hold any real data but are treated by the verifier as if they could, and thus some potentially more difficult inferences can be made. They are only used for verification purposes and do not affect the end product in any way. Ghost variables cannot be passed as arguments to non-ghost code, so there is a limited scope within an object or method where it can be used.

*Ghost methods* are in similar fashion methods that do not influence the function of the code but allows for more advanced proofs where the theorem prover does not succeed in making the correct inferences by itself. This is not used in this project.

## Functions

Single statement methods can be reduced to what is a function in Dafny. Functions are by default ghosts and cannot be used outside of verification code. To make a function callable from non-ghost code the keyword method has to be appended after function. Instead of named return variables a single type after a colon represents the returned value of the body statement.

```
function name(arg_name: type/class): type/class
  requires  boolean_expression;
  reads     set_of_objects;
  decreases integer_expression;
  ensures   boolean_expression;
{
  // single statement
}
// alternatively non-ghost
function method name(arg… etc
```

Although functions are more limited than methods, they do not require annotations like them. Functions can, like predicates, be called directly from verification statements.

# 4

# Dafny's Shortcomings

A quote from rise4fun[8] says "Dafny lifts the burden of writing bug-free code into that of writing bug-free annotations. This is often easier than writing the code, because annotations are shorter and more direct". In fact, having spent about a week's time in total on the actual code and several months on writing annotations, this tends not to align with the claim. Rather, it stands in strong contrast to it. It does not appear to be easier. Nor is it shorter, considering the annotations also takes more space than the code, even after all reusable expressions are refactored out.

## Independent Annotations

```
requires myobject != null;
reads myobject.member;
```

The top code fails because the reads clause cannot conclude that myobject is not null. `member` is considered unreachable. This implies that the clauses are completely independent of each other, and unfortunately introduces redundant code. By experience, verbosity reduces clarity and overview.

```
requires myobject != null;
reads if myobject != null
  then myobject.member
  else {};
```

The solution makes the code significantly less readable.

## State Anchoring

The following code illustrates the discarded information about `this` and subsequently `member` when called by a method mentioning it in its `modifies` clause.

```
method test()
  modifies this;
{
  //do stuff not involving member
}
method somewhere_else()
{
  this.member := null;
  this.test();
  assert this.member == null;
}
```

The assertion on the last line fails, but it is not entirely obvious why that is. Our intention in this code is to prove that `member` is unchanged after passing through `test()`. This should be a trivial task, an implicit one, even for most programming languages not intended for verification, as `member` is obviously never even involved in the code.

```
method test()
  requires this.member == null ||
this.member.valid();
  modifies this;
  ensures  this.member == null ||
this.member.valid();
{
  //do stuff not involving member
}
method somewhere_else()
{
  this.member := null;
  this.test();
  assert this.member == null;
}
```

Dafny however, simply discards all accumulated knowledge about all objects found in the `modifies` clause, leaving us having to explicitly tell what to regain.

One would think two equal `requires` and `ensures` clauses would be capable of that, but the problem is that we accept more than one state, and Dafny cannot see that there is no switching between them. The verification still fails.

Our task becomes one of isolating the states. Knowing that the different clauses are independent of each other, and that the single expression allowed in each one is not capable of saving information to be used in the other clauses, this, rather modest verification, could provide a significant challenge.

One naïve solution would be to make one copy of the method for each state, and test the state at runtime to figure out which one to call. Another one is creating a `ghost` variable copy of the object you are verifying before the call and assert their equality afterwards. Dafny has no deep-copy feature, however, making us having to provide one at the instantiation of the original, and mirroring every interaction with it throughout the object's life. Both of these solutions may be horrible options. Fortunately, one doesn't need a complete copy of the object (ghost or not). In fact, we only need an integer that can be associated with each possible state. Make sure that the integer is not referred to by any of the objects in the `modifies` clause, as this will cause its information to be discarded as well. This technique will be referred to as *State anchoring*.

```
method test(state: int)
  requires (state == 0 && this.member == null)
        || (state == 1 && this.member != null
          && this.member.valid());
  modifies this;
  ensures  (state == 0 && this.member == null)
        || (state == 1 && this.member != null
          && this.member.valid());
{
  //do stuff not involving member
}
method somewhere_else()
{
  var state := 0;
  this.member := null;
  this.test(state);
  assert this.member == null;
}
```

This is a solution, albeit not a pretty one. The parentheses are required. And and Or does not play nice together as in other languages. Also, remember that when one of two Booleans fail on the left hand side of an Or, neither's state is determined on the right hand side, thus checking for not null before calling `valid()`.

Of course since we in this example were working with no more than two states, we could have just as well used a Boolean variable instead of an integer. This is more scalable, however. It would be very useful to have a method where one would not need to know the state beforehand that is capable of this proof. This will probably include ghost variables in some way, but not via the modified object and not as an argument. Since we cannot pass ghost variables as arguments to methods, our `state` becomes unnecessary baggage at runtime.

## Bugs

Compiler bugs and issues due to language features are quite often indistinguishable to the novice. The former tends to appear sporadically and not disappearing before one has painstakingly worked around the problem. Changing unrelated code seems to remove most of them, so when an issue has been circumvented, one might want to revisit the code after some time if one suspects it was a bug.

One such an example was the missing `Length` method, as stated by the compiler error, for the Dafny syntax `|myVar|` for both the native types sets and maps. It caused a lot of unsightly workarounds with sequences as a replacement. These are still visible in the v0.2 source. The syntax was evidently correct, but the code did not work from the onset, and we figured that's just the state of the language at this point. By chance the decision to change it back to a set for one of the methods was made after working on other parts of the code and it strangely worked again.

Another time, this situation arose where an absolutely unique variable were supposedly declared twice in a syntactically correct set declaration. Renaming did not help.

```
set a | a in this.mymap :: this.mymap[a]
```

"Error: more than one declaration of variable name: a#5
Error: more than one declaration of variable name: a#5"

No documentation of what the error messages are supposed to mean was found unfortunately, and the development was delayed for days. Again, working with other parts of the code (although without the luxury of verification as the syntax error stopped the process) the issue disappeared after a while. This is not entirely unexpected as Dafny is still in an experimental language.

## Extreme Redundancy

Consider which annotations cannot be derived directly or indirectly from either the code in the body or the context around calls, and you can see that most of it is simply a copy of how the code behaves.

Almost all verification annotations are redundant! Due to artificial restrictions by Dafny developers, intended to reduce verification-time (at the severe expense of development time), the scope of verification does not extend beyond a single method or function. This would of course be a good thing if the verifiable statements were automatically derived at least to some degree. Forcing the developer to do redundant work stagnates development and increases the amount of annotations needed by orders of magnitude, not even proportional to the computing time saved.

The code's behavior is independent of its verification, and by inference, all statements about the code also is. As such, no information provided in annotations is crucial to the verification of any statement apart from assumptions made about data from outside the system!

The only useful part of this explicit annotation, which is ensuring the programmer that a complicated system works "as intended", is vastly overshadowed by the sheer amount of secondary annotations needed to manually and meticulously set the stage for the few interesting verification statements to work. The majority of one's time goes into verifying things that could be implicitly derived. Days of work for the programmer is sacrificed for seconds of computing time.

As a closing note, if Dafny has the degree of insight to provide the following error message, it is not far behind the ability to figure out which objects are being modified. Such an ability would make the `modifies` clause moot and save a whole lot of trouble already.

"Error: assignment may update an object not in the enclosing context's modifies clause"

# 5

# Version One

In the following three chapters three generations of the project will be highlighted and some context around the development will be provided.

Because the verification of the Paxos algorithm depends on the asynchronous interaction between various agents, and because Dafny has no concept of asynchronicity, the first solution arrived at was that Dafny had to be cheated, somehow, into believing the entire system is self-contained in one process, while allowing the client program to utilize parts of the library independently.

The first attempt include only one file that incorporates a hierarchical architecture, the class `DummyNetwork` at the top, `Interface`, `Group` and then `Proposer`, `Acceptor` and `Learner` as leaf objects, responsible for Paxos' core work.

## paxos.dfy

`DummyNetwork`, the overarching singleton, is supposed to pass along messages to the correct Interface based on `dest_ID`, the first parameter in all its methods. The class is useless during runtime, as a real Ethernet connection is doing this job, but it serves as a way for Dafny to see that a message gets where it is supposed to within the system. Because Paxos must work even when some messages fail to reach their destination, an implementation of random failures was considered, but Dafny has no way to get random values. Those would need to be provided through the client application, in which case we wouldn't really have to do anything as we don't use the class at all. Dafny would in all likelihood complain about the scary unknown value it could get, and not enough time has been spent to figure out how to verify non-deterministic code. Dafny has no concept of probability, so it might even be impossible.

`Interface` has two sides, one inside for receiving calls from Paxos roles and one outside for receiving from the network. An interface represents one agent. By default, the object simply bridges calls between the `DummyNetwork` and the agents' Paxos objects. The client application should create a subclass of `Interface`, substitute `DummyNetwork` with a real network and choose a fitting protocol.

`Group` represents an isolated collection of agents from one agents perspective. It includes arrays of all participants' IDs for each Paxos role. These are used as a list of recipients when broadcasting to all agents performing a given role. A single agent's ID may be mentioned in all three arrays if all roles are performed. Group also maps `slot_ID`s to local instances of the Paxos algorithm. Also here, one map for each role. No Multi-Paxos optimization is implemented for multiple slots.

The `Proposer`, `Acceptor` and `Learner` classes will be covered in the third version, where they are the most up-to-date.

```
requires grp != null && grp.valid()
  && grp.interface.valid()
  && forall i ::
    i in grp.interface.net.interfaces ==>
    grp.interface.net.interfaces[i] != null;
```

Since `valid()` is a member of `grp` we can't include a check for not null inside it because we depend on the condition to call `valid()` in the first place.

The objects require its argument to refer to valid objects throughout the entire system. Dafny does not settle with proving that invalid objects cannot be created. Methods are verified in isolation and can potentially be called with any input from the client application. To prove that validity is ensured throughout the entire message traversal and response a cascading tree verification was used.

## Cascading Validation

`DummyNetwork` would verify all the interfaces and `Interface` would verify all its groups, etc. For one predicate to call another, one must specify all the read objects from the nested predicate in the top predicates reads clause. This can be done with a built-in function on all predicates and functions called `reads()`. This takes as arguments whatever the referent function takes, (e.g. `noArgFnc.reads()` takes no arguments) and returns the set of objects read by that function when called.

```
reads if 0 in this.myMap
    && this.myMap[0] != null
  then this.myMap[0].valid.reads()
  else {};

reads set x | forall y :: y in this.myMap
  && this.myMap[y] != null
  && x in this.mymap[y].valid.reads();
```

The first `reads` checks if 0 maps to a not null object and returns its `valid` predicate's read objects, else empty. The second clause returns `X such that ∀x ∈ X and ∀y ∈ myMap, myMap[y] ≠ null and x ∈ myMap[y].valid.reads()`. In English this becomes: For each object in `myMap`, add all the `valid` predicate's `reads` to a set and finally return that set. So in other words. Since we require all these mapped object to be valid, we must also read everything that the `valid` predicate's reads for all objects.

If a constant is asserted as a key in the map, the value's members are accessed successfully. If, however, we generalize the expression to account for all the elements in the map with a `forall` loop, Dafny seems to fail with the error: insufficient reads clause to invoke function. If it works with a verified constant it should work with verified elements of an iteration.

This was the last roadblock encountered before the code was abandoned and version two was started from scratch. It wasted a lot of time and the issue remains unsolved.

The initial approach was to write the program so that is works, then verify it. That was a very bad idea. The program will only finish compiling successfully if the program is verified. When the verification process started, the program was already big enough to loose one's overview working with it. Having to further add more than twice the code's worth of annotations, partly due to Dafny's severely limited scope of verification, this environment was the worst kind to learn in.

Additionally, Visual Studio 2010, for which the Dafny plugin was designed, was not working, so the code was developed in *Sublime Text*[12] and compiled occasionally with a batch script. The console greets one indifferently with a wall of errors.

# 6

# Version Two

In the hope that the Dafny plugin for Visual Studio 2010 would work on newer versions, a successful second chance was given by upgrading to VS 2013.

As Dafny is developed with C#[9] and this is available in Visual Studio, it was chosen as the tool with which to program the client application with the assumption that it would prove to be the least resistant towards integration. This, despite no prior experience with the language.

This version was developed as separate Paxos files, with the plan to finish them independently and use them with a client. The final proof cannot be verified with this disjoint architecture but the first milestone was to get something working. Being a straight forward task using what had been learnt so far, this only took a single week, whereof most of the time was spent learning C# and how the library interface behaved.

The C# source is thrown together as a simple means to an end without plans of revisiting it and is therefore not commented. It provides a command interface console and a simple UDP server and client. A command interface library was first used, but it surprisingly proved to be both more work and way more code than the final straight forward solution.

The Paxos roles was split into three files, named accordingly, and generics was introduced on a whim. Value is no longer constrained to be an integer. `class Myclass<T(==)>` means the generic type T is numerically comparable. T can be either be a native type or an object. This means one cannot compare T to null without constraints.

## The Library Interface

Nothing is said about the interface in the documentation read in preparation for the project. This section highlights some of the peculiarities of the subject. When Dafny is compiled without a Main method, it becomes a .NET library rather than an executable.

In the src/v0.2/proposer.dfy file in the method `Evaluate_majority` you can see this comment:

// else return nothing? 0, 0? 0, null?

After working with the library in C# it became apparent that methods do not return values. They modify *out*-pointers given as arguments. And in this case, the pointers are simply left unchanged. C# as a language proved to be fairly easy to learn. Boasting a mature community, most of the questions one might have are already posted and answered.

Anonymous constructors are named `_ctor()`. Named constructors are probably called `name_ctor()`, but this is left unconfirmed. The constructors in Dafny works in the same manner as the initializer `__init__()` from *Python*[10]. The difference is that it is not called automatically upon instantiation. Objects are instantiated without arguments and may require an explicit call to this constructor if one is made. It took a receding hairline and a dive into the object model in Visual Studio to discover these members.

## A Working Application

Making sure each Dafny file verified correctly and compiled the libraries could be imported and used by the C# application. Although generic types were used the client limits the value strictly to integers. The possibility of sending text would require more parsing work.

The console has only two commands,

`connect <ip> <port>` and
`propose <round> <value>`

Since the user has full control over the values proposed, the responsibility falls upon the user to follow the rules of the proposer. One can easily change the consensus by proposing a higher round and a different value than before.

The application may run several instances on a single machine. The listening port number will then iterate until binding is successful.

A capture of two connected applications is displayed below. The entered commands are highlighted in blue for clarity. The rows of output showing up after the propose command is the messages received from the different roles. The source IP address and listening port is the first part of each message. All localhost addresses comes from the replica's own roles. Next is the message name and its payload, round and value in parenthesis. A second indented line describes the replica's state after reading the message.

These applications ran on the same machine as evidenced by showing the same IP address. The second replica also failed to bind the standard port as it was occupied and chose the next one. A proposal is sent and each replica's learner print the learned value once both acceptors agree.

```
####################### PAXOS ######################

hosting at 192.168.1.100 : 24805
connect 192.168.1.100 24806
Reconfigured to include 2 replicas.
Connected to 192.168.1.100:24806
propose 2 37
127.0.0.1:24805>> Prepare( round=2, value=37 )
    accepted none yet
127.0.0.1:24805>> Promise( round=-1, value=0 )
    proposer - 2:37
192.168.1.100:24806>> Promise( round=-1, value=0 )
    proposer - 2:37
127.0.0.1:24805>> Accept( round=2, value=37 )
    acceptor - 2:37
127.0.0.1:24805>> Learn( round=2, value=37 )
    learner waits
192.168.1.100:24806>> Learn( round=2, value=37 )
    learner - 2:37
```

```
####################### PAXOS ######################

Error: Could not listen at port 24805
hosting at 192.168.1.100 : 24806
Reconfigured to include 2 replicas.
Connected to 192.168.1.100:24805
192.168.1.100:24805>> Prepare( round=2, value=37 )
    accepted none yet
192.168.1.100:24805>> Accept( round=2, value=37 )
    acceptor - 2:37
192.168.1.100:24805>> Learn( round=2, value=37 )
    learner waits
127.0.0.1:24806>> Learn( round=2, value=37 )
    learner - 2:37
```

Does this prove the algorithm is correct? Unfortunately not. The annotations in the isolated Paxos roles does no more than verify the library will not crash.

# 7

# Version Three

In this chapter some explanation for the different Paxos classes will be provided. In addition we will introduce a new class, `Replica` that will handle the communication between roles in much the same manner as the C #application did, but now hopefully also give the opportunity to verify the system as a whole.

## proposer.dfy

Proposer's valid predicate contain statements about its state independent of arguments. The larger part of the annotations usually contain arguments and can't be refactored nicely. Moreover, when annotations are refactored out, the verification tool does not give accurate errors as they will point to the call and not the contents of the predicate.

The ghost map called `acceptors` is a collection of the acceptors that answered the Proposer by calling its `Promise` method and stores its accepted round and value. This is necessary when we need to say something about the state of the answered acceptors.

```
// majority implies no accepted round is > largest
encountered
ensures ok ==> ( forall rnd :: rnd in acceptors ==> (
    rnd <= largest
));
```

`ok` is a Boolean variable returned by `Evaluate_majority()` and signals that a majority of acceptors has answered. The annotation states that if `ok` is true, no round in `acceptors` is larger than `largest`. In other words, we know the highest accepted round of all the answers received. At this time, the verification fails, indicating the need for a more expressive `requires` annotation.

The reconfigure method is not something in use yet. Changing the amount of acceptors in a slot before a consensus has been reached causes a lot of problems but can also be very helpful when some replicas become unresponsive, drop out and change the majority threshold.

## acceptor.dfy

This class follows a more object oriented principle simply to learn how Dafny behaves under different circumstances. The methods `Prepare` and `Accept` return `Accepted` objects, a second class defined in this module. It simply contains a single round - value pair. This allows the methods to also return null, meaning no response will be sent via Ethernet. The annotations in both is simply a redundant description of the `if` clause in the code. Acceptors are very simple in operation and does not require a lot of verification. The entire module is successfully verified.

## learner.dfy

The constructor is left empty for the `SingleLearner` class to see what effect this would have on the initial values of the object's members. Recall, types cannot be null. It is unfortunate that it has not been successfully compiled yet. Its only method, `Learn`, needs no requirements or valid state and does verify correctly nevertheless. The returned Boolean value `learned` performs the same task as the proposer's `ok`.

## replica.dfy

`import opened` makes all the contents of the named module available without needing to refer through its name each time. The import feature does not work with separate files unfortunately, even when the filenames matches the module name and it exists in the same folder. That is why we have the file "replica imported.dfy", which concatenates the different files. A script could do this pre-compilation but no continuous verification in Visual Studio would be possible by working on replica in isolation. This module has a trait called `EndPoint`. Traits are similar to interfaces but may also extend classes by providing a full method body. `EndPoint` is going to refer to an `IPEndPoint` object passed from the client. This object has an `Equals` method that compares two IP addresses. For our purposes in Dafny this will through the trait be reduced to equivalency.

The `Replica` class' constructor takes into account the possibility of not participating in certain Paxos roles. Some object pointers will thus be allowed to be null. The constructor takes a bitmap integer, `roles`, and determines the instantiation of its objects based on three separate digits, much in the same manner as the `chmod` command on unix systems. It does only distinguish between 0 and anything else though as we have only one potential object of each. If the leftmost digit is not 0 a proposer will be made with starting round -1. We choose -1 because proposals may start at 0. Subsequent roles gets determined by testing the division remainder. The `state_pro` and `state_acp` variables are a consequence of allowing these objects to be null. These were thrown around so much that we allowed the valid predicate to depend on these as arguments. Here, the state anchoring technique is used, albeit with Booleans and not integers.

A ghost map of acceptors called `ghostacp` perform the same function as the one in the proposer class except that this stores the acceptor object itself. Since the proposer does not have the capacity to keep track of all the acceptors, where some might not answer, that will be the task of this map. It is unfortunately not used yet, but from what we have learned so far, it might be required in order to finish the proof.

Most of the methods are simply wrappers for the underlying Paxos objects in the same manner as in the `Interface` class of the first project version. One can see that these methods does not use state anchoring as it requires one to pass the known state as an argument. We must check the state inside the method and thus we cannot guarantee the state being unchanged after returning. If this later turns out to be necessary, a new solution must be discovered.

There is currently nothing in place to prevent the addition of acceptors in the middle of establishing consensus. This is disruptive of the algorithm and may cause replica desynchronization by lost consensus. Removal of unresponsive agents must be allowed however, but once an acceptor is removed, the new majority threshold must be recalculated and collected responses must be compared again. Consider a majority of answers is received and one might not receive any more. If the comparison was skipped, the algorithm would stall, not knowing if a majority was already attained. This is currently only mentioned in the code as TODO comments.

# 8

# Verification in Integrated Development Environments

It shouldn't be unreasonable to expect any moderately clever IDE to restrict its focus to only the code block being edited if it had any features that require parsing code while the user is typing. Once the behavior of that block has been abstracted, any cascading consequences due to edits by the user may change the state of the outer closures. It may not, of course, change any conclusions reached about other code blocks in the same closure. This allows for concurrency, restricts the domain affected by change, eliminates redundant parsing and verification, and thus speeds up future processing. That is something syntax-highlighting, verification, compilation and surely plenty other features all can benefit from.

## Continuous Verification

We would be completely fine with several hours of verification time for a system comparable to the one presented here. No one will probably be able to program faster than the computer can verify anyway. The problem with verification and compilation time is that everything is done at once, and we are forced to wait for it. Dafny evaluates the entire system statelessly in one go after compiling. Repeatedly and redundantly so, in case of Visual Studio's language extension. This does not allow for any coding to be done in the meantime.

Many applications today still exhibits this archaic behavior of starting with arguments, processing it, spitting out a result and then terminating, when a potentially continuous service model would better fit the task. The naïve approach of repeatedly parsing, compiling and verifying the entire source is something not expected to be seen from professionals in a project that's not on a strict deadline.

This choice is unfortunately mainly because all verification is done on the compiled product. The fastest option however is very likely to verify the parser's abstract syntax tree (AST) or a derived higher order abstraction. The benefit of an end product verification is that it is independent of language and compiler. That said, there is nothing stopping us from using both methods. The developer can have rapid AST verification while working, which may in turn produce a detailed verification scheme for low level processing when needed after compilation, probably making it even faster than if by itself!

## Verifying Existing Languages

If you access something through a pointer or object without checking for null, it is obvious yet implicit that not null is an assumption that must be satisfied in that block of code. This is mostly independent of language and may be checked at almost any point in the compilation process. Everything mentioned so far in this chapter can be applied regardless of the language employed. Annotations may not necessarily be a syntactical part of the source. They may be pointers to the code from the IDE describing what to assert or assume at a given point. We might want our source to remain compatible with other applications.

If you wanted to assert something while working on your code, it should be quick to add an annotation, let the verification process asynchronously cascade through the syntax tree, see the results, and move on. Subsequent annotations may build on previous conclusions and finish their evaluation a lot faster. For most verification purposes we won't even need explicit annotations as they can be reduced to something like the statement "assert that unhandled exceptions cannot happen".

If you want to verify your code, you need to, on some level, describe how it is supposed to behave, somewhat implying that that's not what the code already does. Is not the need for a verifying statement an admittance of a lack of clarity in either your code or the language itself?

# 9

# Conclusion

The goal of this project has, besides verifying Paxos' correctness in Dafny, been very loose and explorative. It was unknown to all parties involved at the start what the difficulty of it would be. By the end of this we have taught ourselves to use two new languages, one of which is pretty unique.

Even with examples provided and explained by online resources, scarce though they may be, when the foundational concepts are still alien, it can be very hard to generalize and port the necessary lessons to benefit one's own cases. The concept of code verification is not a difficult one by any means, but the very restrictively interdependent and redundant procedure Dafny encourages makes this harder than necessary.

## Pick Up the Ball

If the reader finds the subject intriguing, the task still remains to finish the proof, additionally one might implement fast/multi-paxos. Implementing verification in an IDE based on chapter 8 or building one is of course also an interesting direction, one I would pursue if time allows.

# 10

# Related work

Nuprl[4] is a proof assistant that implements constructive type theory (CTT).

EventML[3] is a ML-like constructive specification language that cooperates with Nuprl and is specifically designed to cope with distributed systems such as consensus algorithms.

*Raft*[13] is another interesting consensus algorithm with a different and simpler concept said to be easier to understand yet with comparable performance to Paxos.

*Verve*[14] is a fully verified operating system.

*Vgo*[15] – verifiable go adds new constructs for Google's programming language go.

*IronClad*[16] has a goal of, on top of verve, providing certification of functional correctness of server code to clients, guaranteeing information isolation and proper security standards without needing to disclose proprietary source code.

Amazon Web Services' use of *TLA+*[17] (Temporal Logic of Actions) to find bugs in complex, scalable distributed systems. TLA+ does not verify executable code, but a mathematical model of it.

Other projects that use TLA+ for verification include [18,19,20,21,22]

# 11

# References

[1] Microsoft Reseach – Dafny
http://research.microsoft.com/en-us/projects/dafny/

[2] Leslie Lamport - Paxos Made Simple
http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

[3] Cornell University - EventML
http://www.nuprl.org/software/
http://www.cs.cornell.edu/~rahli/

[4] Cornell University - Nuprl
http://www.nuprl.org/

[5] CodePlex - Dafny documentation
https://dafny.codeplex.com/documentation

[6] Microsoft Reseach - Boogie
http://research.microsoft.com/en-us/projects/boogie/

[7] Microsoft Reseach - STM solver Z3
https://github.com/z3prover/z3/wiki
http://research.microsoft.com/en-us/um/redmond/projects/z3/

[8] rise4fun - Getting Started with Dafny: A Guide
http://rise4fun.com/Dafny/tutorial/guide

[9] Microsoft Developer Network - Visual C#
https://msdn.microsoft.com/en-us/library/kx37x362.aspx

[10] Python Docs - __init__
https://docs.python.org/3.4/reference/datamodel.html#object.__init__

[11] Microsoft Visual Studio
https://www.visualstudio.com/

[12] Sublime Text Editor
http://www.sublimetext.com/

[13] Diego Ongaro and John Ousterhout - Raft
https://ramcloud.stanford.edu/raft.pdf

[14] Jean Yang and Chris Hawblitzel - Verve
http://research.microsoft.com/pubs/122884/pldi117-yang.pdf

[15] vgo
http://vgo.readthedocs.org/en/latest/tutorial.html

[16] IronClad Apps
http://research.microsoft.com/pubs/230123/osdi-2014.pdf

[17] Amazon Web Services - TLA+
http://cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/fulltext
http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

[18] Miguel Castro and Barbara Liskov - Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm
http://research.microsoft.com/en-us/um/people/lamport/tla/byzpaxos.html

[19] Leslie Lamport - Fast Paxos
http://research.microsoft.com/apps/pubs/default.aspx?id=64624

[20] Leslie Lamport - Generalized Consensus and Paxos
http://research.microsoft.com/apps/pubs/default.aspx?id=64631

[21] Leslie Lamport and Brannon Batson - High-Level Specifications: Lessons from Industry
http://research.microsoft.com/apps/pubs/default.aspx?id=64638

[22] Iulian Moraru, David G. Andersen and Michael Kaminsky - There is more consensus in Egalitarian parliaments
http://dl.acm.org/citation.cfm?id=2517350