Exploring the program verifier Dafny that can compile to other languages

A case-study of Dafny, a formal verification tool

Jeroen Koelewijn[1]
Supervisor(s): Dr. Benedikt Ahrens[1]**, Kobe Wullaert**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
22nd June 2025

Name of the student: Jeroen Koelewijn
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Kobe Wullaert, Maliheh Izadi

An electronic version of this thesis is available at https://repository.tudelft.nl/.

Abstract

Formal verification is a stricter way of ensuring correctness of a program, but sitting down and writing the proof yourself is often time-consuming. SMT solvers try to automate parts of this process. This paper aims to explore Dafny, a programming language and verifier that uses an SMT solver underneath to do this verification. This paper will go over its logical foundations, and how it can be used to verify an in-place selection sort algorithm as well as a key-value store. It will also explore a feature of Dafny that allows the user to compile to other languages, and will discuss its usefulness in the industry. Verifying the sorting algorithm was very straightforward while the key-value store posed some problems. However, Dafny itself seems to be very straightforward to program in. Its ability to compile to other languages leaves a lot to be desired. While the compiled code is fully functional, the code is barely readable and less than ideal to work with. A further study discussing Dafny's ease of use compared to other tools could be conducted to see if the lacking compiler could outweigh having a native verifier specifically designed for a high-level programming language.

Keywords: Dafny, Automated Formal Verification, software verification

Acknowledgements

I would like to thank Kobe Wullaert and Benedikt Ahrens for guiding me through this project. I would also like to thank my peer group members Kaj Neumann, Mohammed Balfakeih, Tejas Kochar, and Dinu Blanavschi for the countless support during this project. Their feedback helped a lot in getting this paper to where it is now.

1 Introduction

Ensuring correctness of failure sensitive software such as security, automotive, control, and aerospace systems is crucial. There are many ways to check the validity of written code, with writing tests being the most widely used practice. However, "program testing can be used to show the presence of bugs, but never to show their absence!" [1, pg.7], as Edsger W. Dijkstra said. This is why writing tests is not strict enough to ensure total correctness of software. One such stricter form is formal verification, which is the mathematical approach to checking if a program satisfies given properties, through a formal model of the program with Hoare logic as its basis [2].

However, one major disadvantage of formal verification is that it had to be done by hand. Satisfiability modulo theories (SMT) solvers try to partially solve this problem by automating parts of this formal verification. This is done by operating on first-order logic and reasoning on the negation of a given premise. With this, they try to find a model that satisfies a given set of clauses. Examples of these type of SMT solvers are Z3 [3] and Alt-Ergo [4].

Over the years, many different programs have been created using these SMT solvers as their underlying foundation to help with formal verification of written software such as Spec# [5], KeY [6], VCC [7] and Dafny [8]. This paper will focus on Dafny, a programming language designed with formal verification in mind, exploring its logical foundations. This paper will also show how Dafny can be used to verify an in-place selection sort algorithm, as well as a key-value store using a custom hashmap. The latter is to my knowledge never done before. Furthermore, Dafny allows for compilation of its written code to other languages such as C#, Java, Rust, and Python. This paper will also explore the usefulness of using this feature as part of the programming pipeline to verify certain aspects of a program. While a lot of research is done surrounding Dafny, we found lacking information regarding why this language has low usage in the industry considering its ability to compile to other languages. By exploring this ability we hope to find out as to why this may be the case.

The structure of this paper is as follows. Section 2 will go over the background of Dafny as well as a simple explanation on how Dafny can be used for verification. Section 3 will go over the formal

problem description of the in-place selection sort and key value store. Section 4 will give a structured overview of how these were verified using Dafny. It will also go over the outcome of compiling the verified code to C#. Section 5 will discuss the results, go over future recommendations for research, and conclude the paper. Finally, Section 6 will be about reproducibility and LLM usage.

2 Background

This section will consists of three parts. A brief introduction of the Dafny language itself, an overview of basic Dafny constructs, and finally a look at the verification pipeline of Dafny.

2.1 What is Dafny?

Dafny is a programming language and verifier made by the Research in Software Engineering group at Microsoft Research in 2009 under the lead of K. Rustan M. Leino [8], [9]. Dafny is built on top of Boogie, which is an intermediate language created to design program verifiers [10]. Boogie has already been used before to build various program verifiers for different languages such as Spec# for C#, and VCC for C. Unlike those, Dafny offers its own unique language called Dafny. Because Dafny has its own unique language it could fully be made with formal verification in mind. This means verifying and implementing the software are more integrated with each other.

VS code extension vs binary There is more than one way of getting started with Dafny. The easiest one by far is using the Dafny Visual Studio Code extension[1]. This extension acts like an IDE for Dafny. It automatically checks the syntax and compiles to Boogie. It makes the Dafny coding experience a lot easier to manage.

Dafny can also be used in the command line, by installing a binary[2]. While less convenient when it comes to writing code, it allows for more freedom when compiling code to other languages.

related works Work on Dafny has been consistently active, with the most recent paper published in 2025 about improving axioms in Dafny [11]. Dafny is primarily used for teaching in academic settings but that is not the only place where it has been used over the years. Usage of Dafny in industry has been increasing ever since its release. It was used by Amazon to model both the authorization engine and the validator of Cedar which is a authorisation-policy language [12]. Microsoft has also used Dafny for verification in their IronFleet project [13]. Dafny has also been used for Qafny, which is a verifier for quantum-programs using Dafny as a backend. Qafny has been used to verify notable quantum algorithms like quantum-walk algorithms, Grover's algorithm and Shor's algorithm [14].

2.2 Dafny basics

Classes Dafny allows for the construction of classes, which in return allows for the creation of objects.

Methods These are imperative executable pieces of code. They are allowed to modify the heap. They are allowed to be defined recursively but their termination has to be proving using a **decreases** clause.

[1] https://marketplace.visualstudio.com/items?itemName=dafny-lang.ide-vscode
[2] https://dafny.org/latest/Installation#windows-binary

Functions A **function** in Dafny is different from a **method**. They must be pure (so no side effects), total (must be defined for all inputs of their domain) and must always terminate. While termination still has to be proving, Dafny can more often figure this itself when it comes to **functions**. Because functions must be pure they are not allowed to modify the heap, but in return are allowed for reasoning in clauses.

Clauses **Requires** describe what must hold at the input of a **method** or **function**. On the opposite end, **ensures** are the clauses that describe what must hold at the end of a **function** or **method**. They are the proof goals. **Invariants** are the clauses used for loops. They must hold at every iteration of the loop.

Imagine you want to take a positive integer a, multiply this integer by 3 and then take the modulo of a positive integer b of the result. One could use a Dafny **function**, using requires and ensures clauses, to verify the correctness of said function. That by ensuring that the output is always between 0 and b:

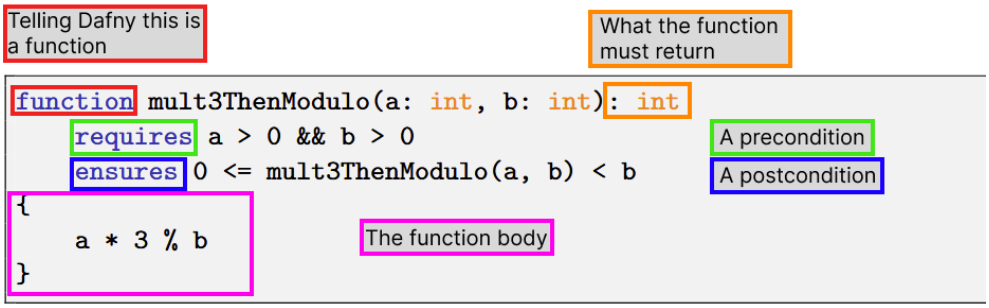


Figure 1: example function in Dafny

Datatypes Dafny has your standard types that you would expect of a language like integers, booleans, strings, etc. Next to that Dafny also has **arrays**, which is an immutable object in memory consisting of a sequence of mutable locations. This is different from Dafny **sequences**, which are not objects by themselves, but rather a collection of heap allocated objects. Other collection types of Dafny are **sets**, **multisets**, and **maps**.

Reads, Modifies Dafny allows **functions** and **methods** to access memory and allows **methods** to mutate. To ensure that all of this works out, everything in memory is off the table unless told to. To accomplish this Dafny uses Dynamic frames [15]. Dynamic frames in dafny model the heap as a map of object references and their fields. You then frame, using the object itself or a collection of objects, what parts of it can be accessed (**reads**) and even what parts can be modified (**modifies**). This is checked by the Dafny language itself.

Predicates A **predicate** in Dafny is simply put a **function** that returns a boolean. While they don't add anything new in that sense, they are a great way of taking large clauses and condensing them into a single **function** taking an input.

Lemmas A **lemma** in Dafny is basically a part of a proof. When going beyond toy examples Dafny often has a hard time proving something from just the pre- and postconditions. **Lemmas** are a great

way of breaking up this proof into smaller blocks to guide Dafny in the right direction. Because of this, **lemmas** are purely intermediate steps rather than a goal unlike **predicates** which can be either a pre- or postcondition.

Asserts **Asserts** in Dafny are rarely something needed for actual verification. They are however a great way of debugging your proof. When an **assert** is encountered it tells Dafny to verify it right now with everything that it knows thus far. With this you can break up your proofs into smaller chunks to see what Dafny is and is not able to verify allowing you to find the core issue.

Dafny has much more beyond the basics given here, which you can find in their reference manual [16]. However, these are all the constructs that we will need to verify the in-place selection sort algorithm and the key-value store. Dafny is capable of a lot more than what will be shown here. M. Leino has created a website with several papers outlining many advanced features of Dafny [17].

2.3 The verification pipeline

To verify your programs, Dafny takes all written code and all written clauses (**requires**, **ensures**, **invariant**, **decreases**, **asserts**) and translates these to the Boogie language. The Boogie program that is created from this conversion is checked by the Boogie language. This check creates **verification conditions** (VC), which are logical formulae generated from all the clauses Boogie finds that must hold. Depending on the type of clause, it generates a different type of VC. These generated VCs are then sent to Z3 [3], which is the default SMT solver Boogie uses. Z3 then tries to verify all these VCs. If there are one or more VCs it cannot verify, it will send that to Dafny.

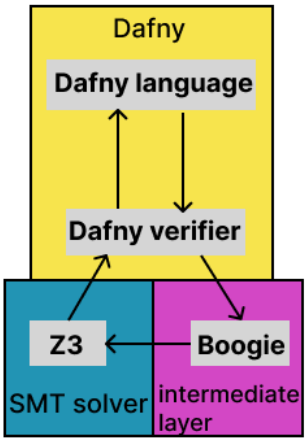


Figure 2: The verification pipeline of Danfy

3 Formal problem description

The following versions were used for the different tools used in the research:

Tool	Version
Dafny Visual Studio Code Extension	3.4.4
dotnet[3]	9.0.203
Dafny Binary	4.10.0
C# Dev Kit Visual Studio Code Extension[4]	1.20.35

Table 1: Tools used and their version

It should be noted that the Dafny visual studio code extension uses Dafny 4.10.0.0 which is the same version as the binary. When it comes to the Binary, the hash of the commit for the version we used is `f24efae13647804624723de981bb5c95ea83e177`.

[3]https://dotnet.microsoft.com/en-us/download/dotnet/9.0
[4]https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit

3.1 In-place selection sort

The selection sort we will be implementing, visible in Listing 1, consists of 3 main parts each needing its own axioms for verification. We will make use of a *pivot* which is a position in the array that divides the sorted part of the array from the unsorted part.

minimum value This needs to ensure that the minimum value is found in the unsorted part of the array. We can implement this using a while-loop, for which i is the loop index. With an **invariant** for each iteration, we ensure that the current minimum value is indeed the smallest in the already explored part:

$$\forall j \in [\text{pivot}, i) : v[\text{min\_index}] \leq v[j]$$

swap This must ensure that the *pivot* and previously obtained minimum value are swapped correctly. That means that at the end of the swap everything up until the pivot needs to be smaller than everything after the pivot:

$$\forall i \in [0, \text{pivot}), \forall j \in [\text{pivot}, \text{len}) : v[i] \leq v[j]$$

main loop This needs to combine the two previous parts in such a way that in the end the list is fully sorted. For every iteration, the sorted part of the array must be sorted incrementally. *len* here denotes the length of the array itself:

$$\forall j \in [0, \text{len}), \forall i \in [0, j) : v[i] \leq v[j]$$

```
method selectionSort(v: array<int>) {
    // for convience we give the length of the array a name: len
    var len := v.Length;
    var pivot := 0;
    while pivot < len {                    |  main loop
        var min_index := pivot;
        var i := pivot + 1;
        while i < len {                    |
            if v[i] < v[min_index] {       |
                min_index := i             |  minimum value
            }                              |
            i := i + 1;                    |
        }
        var temp = := v[pivot];            |
        v[pivot] := v[min_index];          |  swap
        v[min_index] := temp;              |
        pivot := pivot + 1;
    }
}
```

Listing 1: Implemenaton of selection sort in Dafny. Note that the sectioning on the right is not part of the language

3.2 Key-value store

The key-value store is implemented using a custom-built hash map. It consists of the following components:

- **Fixed-size array:** The main storage is an array named *table* of size $SIZE$, which is set to 512.
- **Hashing and Collisions:** A hash function, which is passed as a lambda of type $K \rightarrow$ int, maps each key of generic type K to an index in the array. When multiple keys hash to the same index, the corresponding values are stored in a collision chain.
- **Collision chains:** These chains are implemented as Dafny **sequences**, allowing dynamic, ordered storage at each array slot.
- **Entry storage:** Each element in a **sequence** is an instance of the *Entry<K, V>* class, which encapsulates a key and its corresponding value of type *V*.

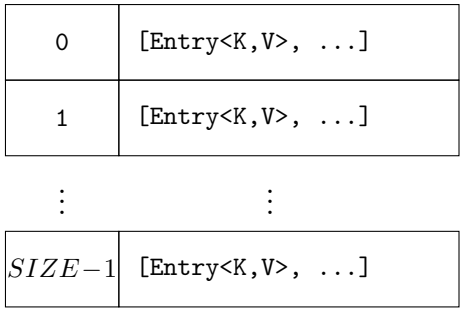


Figure 3: Structure of the custom hash map. Each array index stores a sequence of entries.

Key uniqueness It is important that all keys contained in the map are unique. The easiest way to define this is using set notation. Let Map be this set containing all key-value pairs contained in the hash map. Then we define a function `keysOfMap(`Map`)`:

$$\texttt{keysOfMap}(Map) = \{k \mid (k, v) \in Map\}$$

Uniqueness is something inherent of sets so uniqueness of keys would be defined as the length of `keysOfMap(`Map`)` being equal to the length of Map:

$$|\texttt{keysOfMap}(Map)| = |Map|$$

Within the class structure of the hash map we define 3 methods, all ensuring key uniqueness at its in- and output:

Get(key) This takes in a key. If the key exists within *table*, it returns the value attached to this key. If not then *nothing* is returned. For this method we need to prove that if the key indeed exists within the map, a value is returned. This value then must be what was attached to the key. If the key does not exist in the map, we need to prove that *nothing* is returned.

Delete(key) This takes in a key. If the key exists within *table*, it deletes this key value pair and returns the value that was attached to this key. If not then *nothing* is returned and *table* remains as is. We need to prove the same here as **Get(key)** in addition to making sure that the key no longer exists in *table*.

Put(key, value) This takes in a key and value. If the key exists within *table*, it replaces the value of that key with the one given. If the key does not exist in *table*, it is added at the end of the *chain*. Here we need to prove that at the end of the method the key exists in *table* and the value attached to it is what was given at the input.

In order to return this *nothing* we use a Dafny **datatype** to construct *Option<T>*. This can either be of type *Some* which will have a value in it or of type *None* which contains nothing:

```
datatype Option<T> = Some(value: T) | None
```

Listing 2: implementation of the Option<T> datatype in Dafny

3.3 Compiling to C#

We are very limited regarding testing the correctness of the compiled code. The main issue is that just writing tests to check for functionality is not enough to ensure correctness. To make sure the compiled code is correct we would need to verify the Dafny compiler to C# itself, as well as verifying the C# compiler. This is way beyond the scope of this research. So for this case we will test on how easy it is to use the compiled code, the quality of the compiled code. This will be tested by taking what has been compiled and then taking that code and check its functionalities against a few toy examples. These toy examples will be simple if-else statements that check the obtained output is correct given an input.

4 Implementation

Most of this section will focus on the verification of two problems rather than diving into how they were implemented. However, for the key-value store several aspects of the methods were extracted into separate functions for easier verification. Which parts got extracted will be mentioned in Section 4.2.

4.1 Sorting Algorithm

All core components were verified very closely to what was described in Section 3.1. Nothing needed to be changed about the implementation.

minimum value The syntax for actually writing down these clauses is very similar to what you would have for a mathematical definition. The axiom we set out to verify here can then also be translated as follows to a Dafny **invariant** for verifying finding the minimum value in the unsorted part of the array:

$$\forall j \in [\text{pivot}, i) : v[\text{min\_index}] \leq v[j]$$

```
invariant forall j :: pivot <= j < i ==> v[min_index] <= v[j]
```

One extra thing Dafny in this case needs is confirmation that what we are indexing over is indeed in bounds. In this case, we need to check this for both i and min_index.

swap Similarly, we can do a pretty direct translation to Dafny code. However, instead of defining the two domains separately, they are defined in one loop:

$$\forall i \in [0, \text{pivot}) \,, \forall j \in [\text{pivot}, \text{len}) : v[i] \leq v[j]$$

```
invariant forall i, j :: 0 <= i < pivot <= j < len ==> v[i] <= v[j]
```

Main loop This algorithm modifies the contents of the input array. Besides giving a **reads** clause so that Dafny allows us to do this, we also need to verify that the content of the array only ever gets moved around within the array and not removed. This is very important when modifying content within the heap. To ensure this, we will construct a **predicate**. This predicate makes use of Dafny's **multisets**. Specifically, we will be converting the contents of the array to a Dafny **sequence**. This allows us to convert it directly to a Dafny **multiset**, since arrays in Dafny are objects. A **multiset** in Dafny keeps track of all unique elements and how many times they occur. This means if two **multisets** are the same, the contents of those sequences are the same. And since we took all the items of the array into a **sequence**, that also means that the two arrays have the same content. We will call this predicate `sameContent(`$arr1$`,` $arr2$`)` and construct it as follows:

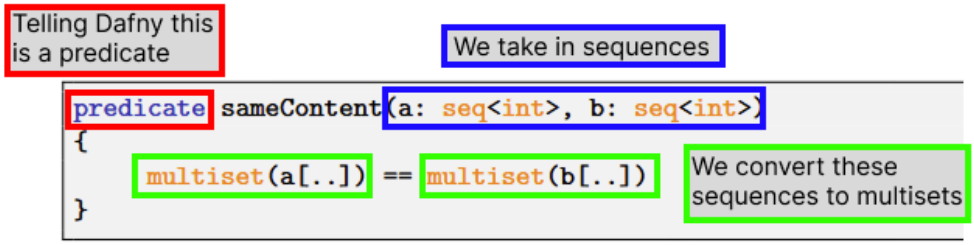


Figure 4: predicate to ensure the content stays the same

We call this predicate in the main loop using an **invariant**. In order to call the previous state of the array we can make use of Dafny's *old()* function. *old()* in Dafny is a reference to the previous state of the input, which can be either the beginning of a method/function or a previous step of a loop iteration.

With that the only thing left to do is construct the clause defined earlier into something Dafny can understand. This to prove that at every iteration of the loop, the sorted part of the array is sorted in ascending order:

$$\forall j \in [0, \text{len}) \,, \forall i \in [0, j) : v[i] \leq v[j]$$

```
invariant forall i, j :: 0 <= i < j < pivot ==> v[i] <= v[j]
```

4.2 Key-Value store

Verifying this key value store ended up deviating in a lot of places. Extracting parts of the **methods** into **functions** instead was done for convenience. This made proving these **methods** a lot easier, as reasoning about recursion is often more straightforward than having to define loop **invariants** that must always hold. By far the biggest deviation was in verifying the correctness of unique keys.

Key Uniqueness The main reason why we cannot define uniqueness the way we did in Section 3.2, is that Dafny cannot reason backwards from it. Showing Dafny two sets are of equal length does not give it any information about what this means for the elements inside the sets. If none of our other proofs relied on keys being unique, this would not have been a problem. Unfortunately, all three methods require this uniqueness for other clauses to work. Otherwise, Dafny is unsure that only one possible output exists. So we need to come to a definition that Dafny can use to reason backwards.

Updating Key Uniqueness Since we need a definition that reasons about the content of the map, we can first define the following expression for what it means for a single chain to be unique:

$$\forall i, j \in [0, |chain|) : i \neq j \Rightarrow chain[i].key \neq chain[j].key$$

```
forall i, j :: 0 <= i < |chain| && 0 <= j < |chain| && i != j ==>
chain[i].key != chain[j].key
```

We define this as a **predicate** under the name of `keyUniquenessChain(`*chain*`)`. Since our map, which we named table, is just an array containing all chains, we can ensure uniqueness of all chains in the map as follows:

$$\forall chain \in table : keyUniqueness(chain)$$

Of course, this is not enough to ensure uniqueness over the entire map since we don't know if the same key exists in different maps. However, we are only ever operating on a single chain with any given method. Every other chain is never explored. In order for our future proofs to function Dafny only ever needs to be aware that keys within a chain are unique. Keys over the whole map being unique can be a separate concept of its own. This makes proving the reset easier. Instead of matching each chain to check if they have no overlapping keys we are going to prove that any given key will also end up in the same chain. We can do this because Dafny **functions** must be pure, total and must always terminate. This means they are deterministic. Determinism meaning that if two inputs, $x1$ and $x2$, are the same, the output of a function $F(x)$ is always the same:

$$x1 = x2 \Rightarrow F(x1) = F(x2)$$

Since the hash function we defined is a Dafny **lambda**, which have to adhere to the same rules as functions, we now know that all keys in a chain being unique means all the keys in the full map are unique.

Updating the methods This notion however allows for more than just proving uniqueness over the entire map. It also allows us to reconstruct the methods. Since we know a key can only ever be in one chain we can directly go this chain and ignore all the other chains. This allows us to create the following template that all methods will use:

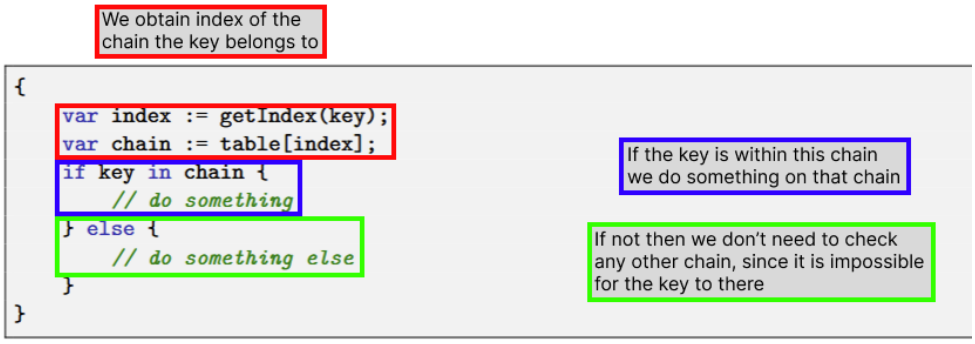


Figure 5: Formal for all three methods, get, delete and put

However, using it like this means we now have to make Dafny aware of it. This can be done by telling Dafny all keys within a chain have as output of the `getIndex(`*key*`)` function the position of the chain in the map.

$$\forall i \in [0, |table|) \, , \, \forall e \in table[i] \, : \, getIndex(e.key) == i$$

```
forall i ::  0 <= i < SIZE ==> forall e ::  e in table[i] ==>
getIndex(e.key) == i
```

We model this as another predicate under the name of `keyMapsToHashedIndex(`*table*`)`. Nothing else has to be proven to Dafny for this to work properly.

Creating a helper function Before moving on to the three main methods we first create a function to quickly collect all keys of a chain and all keys of the entire map. This to more easily reason about just keys. All we will then need to do is call this function instead of creating a full loop again. We call these functions `keysOfChain(`*chain*`)` and `keysOfMap(`*table*`)`. Let's say we want to make sure a *key* exists within a *chain*. Instead of having to write:

$$\forall i \in [0, |chain|) : \exists e \in chain \; . \; e.key = key$$

We can simplify these statements to:

$$key \in \texttt{keysOfChain(}chain\texttt{)}$$

For both of these functions, we need to tell Dafny what they actually return in the form of two **ensures** clauses. In the case of `keysOfChain(`*chain*`)` we tell Dafny that all keys of the input are in the output and that all keys of the output are in the input:

$$\forall e \in chain : e.key \in \texttt{keysOfChain(}chain\texttt{)}$$

$$\forall k \in \texttt{keysOfChain(}chain\texttt{)} \, : \, \exists e \in chain \; . \; e.key = k$$

```
ensures ensures forall k ::  k in keysOfChain(chain) ==> exists e ::  e in chain
&& e.key == k
```

We use almost the exact same approach for `keysOfMap(`*table*`)`. Instead of reasoning over the individual entries of the map however we reason over the chains. We then take what we already know about the chains.

get(key) This function is very straightforward. Since nothing is modified, we also do not need to check that everything has remained unchanged. The part that was extracted here into its own function is when the key is inside of the chain. In this case, we need to find the key and return its value. We call this function `getEntryChain(`*chain*`, `*key*`)`. The most important thing this function has to prove, is that what we return is actually the value attached to our key.

$$\forall e \in chain : e.key = key \Rightarrow \texttt{getEntryKeyChain}(chain, key) = e.value$$

```
ensures forall e ::  e in chain && e.key == key ==>
getEntryChain(chain, key) == e.value
```

The value returned from this method is then wrapped in *Some*. We also give this function two preconditions. One tells it that the key exists in the chain. The other is that the chain has unique keys. This tells Dafny that we will always return something and that something can only be from one key. This means that our previous clause is verified without need of further support.

Forcing Dafny to reason backwards That just leaves us with the two cases that return what we want. If the key is in the table, we return *Some* with the correct value inside:

$$key \in \texttt{keysOfMap}(table) \Rightarrow out = \texttt{Some}(\texttt{getEntryChain}(chain, key))$$

Or when it is not we return *None*:

$$key \notin \texttt{keysOfMap}(table) \Rightarrow out = \texttt{None}$$

Dafny has a problem with the former. This is due to the way Dafny handles goals (**ensures** claues). It does not try to reason backwards from them by unfolding unless told to. This is to save runtime in case it does not need to. This is when a post condition is constructed in the same manner as the function or method. If what you are trying to prove is $A \Rightarrow B$ and your function/method is created going from A to B, Dafny has no problem. We encounter this problem here with `keysOfMap(`*table*`)`. In order to use the function `getEntryChain(`*chain*`, `*key*`)`, the key we give it must be inside of the chain. So Dafny needs to go from B and then reason backwards to A. This requires unfolding which Dafny does not do unless told to.

Normally, this can be solved with a **lemma** or **assert** inside of the function or method. This is not going to work here. The reason for this because `getEntryChain(`*chain*`, `*key*`)` wants this information to already be known when this goal is constructed. In this case, we will have to put an extra **ensures** clause telling Dafny what is going on:

$$key \in \texttt{keysOfMap}(table) \Rightarrow key \in \texttt{keysOfChain}(table[\texttt{getIndex}(key)])$$

```
ensures key in keysOfMap(table[..]) ==> key in keysOfChain(table[getIndex(key)])
```

This clause also fails to verify for the same reason, but this time we can use a **lemma** inside of the **get(key)** method. Dafny already knows the keys inside of a chain always has as output of **getIndex(**key**)** the index of this chain. It also knows that the key does not exist inside

of the chain where it is supposed to be. We give this information to a **lemma** which we call `keyNotInHashedChainNotInTable(`key`)`. We then construct the following **ensures** clauses to tell Dafny that this also means the key cannot be anywhere else. From `keyMapsToHashedIndex(`$table$`)` it knows that:

$$\forall i \in [0, SIZE) : i \neq \texttt{getIndex(}key\texttt{)} \Rightarrow key \notin table[i]$$

Combine this with the information we gave this **lemma** and we now know that the key is not in a chain:

$$\forall c \in table : key \notin \texttt{keysOfChain(}c\texttt{)}$$

A sum over all keys within every chain is the definition of `keysOfMap(`$table$`)`. We can directly create the following goal:

$$key \notin \texttt{keysOfMap(}table\texttt{)}$$

Then we call this lemma inside of the case our key is not inside of the chain. Dafny now links what it knows from the lemma and figures out that a key inside of a map means it is always inside of the chain where it is supposed to be.

delete(key) With **get(key)** verified, the biggest obstacles are gone. Everything we do after that is just a different combination of what we have already done. In the case of **delete(key)**, we extract *removing the key* into its own function. Since this method modifies our table we also need to proof that everything but the changed key gets touched. We can split this between the method and function. The method only has to proof that everything but the chain at `getIndex(`key`)` is untouched:

$$\forall i \in [0, SIZE) : i \neq \texttt{getIndex(}key\texttt{)} \Rightarrow table[i] = \texttt{old(}table[i]\texttt{)}$$

```
ensures forall i ::  0 <= i < SIZE && i != getIndex(key) ==>
old(table[i]) == table[i]
```

The function then verifies that all but the deleted key are still there. Checking that this function returns the value attached to this deleted key is done in the same way as we did for **get(key)**.

put(key, value) The only difference here is that we now need a function for both cases. One for adding a new entry and one for replacing an existing entry. Similarly to `getKeysOfChain(`$chain$`)` we verify that all keys are still there at the end. This function does not return anything. In both cases it also ensures the key to be present in the end. That means we can use the same **ensures** clauses for both cases. Namely that the key is present in the end and that its value is the value passed in the method definition:

$$key \in \texttt{keysOfMap(}table\texttt{)}$$

$$\texttt{getEntryChain(}table\texttt{[getIndex[}key\texttt{]], }key\texttt{)} = value$$

4.3 compiled to C# code

All code was successfully compiled to C# code using the built-in feature that the Dafny VS code extension has. Each produced about 6000 lines of code. Not all of these 6000 lines are the actual code written in Dafny. The majority of it is stuff that is always there, but that you cannot see when programming in Dafny. About 200 of these 6000 lines of code are what was actually written in Dafny.

The code was tested on basic functionality. For the sorting algorithm, it was sorting an array of ten integers correctly, which it did. For the key-value store an instance was created with *int* as key

and *string* as value. Then using the **put(key)** method, we added the following key-value pairs: `<0, ":)">`, `<512, ":(">`, `<10, "byebye">`. Then from this the following was tested:

Test Description	**Passing Condition**	**Result**
Using **get(key)** on key `15` which does not currently exist in the map.	*None* is returned.	Passed
Using **get(key)** on key `10` which has the value `"byebye"` attached to it.	*Some*(`"byebye"`) is returned.	Passed
Using **delete(key)** on key `10` and checking if `get(10)` returns *None* afterwards.	*None* is returned.	Passed
Using **get(key)** on both key `0` and `512`, which share the same chain.	*Some*(`":)"`) is returned for **get(0)** and *Some*(`":("`) is returned for **get(512)**.	Passed

Table 2: Key-value map test cases and results

5 Conclusions and Future Work

Dafny is a programming language built on Boogie which is an intermediate tool used to build program verifiers. Boogie uses the SMT solver Z3 underneath to verify everything written in Dafny. Dafny offers numerous amount of built-in tools to make verification of written software within the language as straightforward as possible. That being said, that doesn't make the verification process easy enough to be used everywhere. The larger a piece of software gets, the more complex the verification gets. When verifying software does not go as planned it isn't always clear if the problem lies with the written program or how it has been verified. Sufficient knowledge of formal verification is still required to fully verify anything beyond some toy examples. Dafny offers a way to generate counter examples, however, this has been near useless for the algorithms verified here. It seems to be able to find the clauses it has issues with but often fails to generate counterexamples for it. While this process is not nearly as straightforward as simply writing tests, once something is verified the stronger guarantee of correctness is very valuable. Hence with how these tools are right now if total correctness is really important like in security systems, tools like Dafny offer a great solution to this problem.

Verifying the sorting algorithm was very straightforward. The many built-in tools of Dafny helped a lot here since they reduced the complexity of what had to be proven. This was less of a case when it came to the key-value store. Here you really felt that you really need to be explicit. This was most noticeable when trying to verify the uniqueness of keys. Something that would be sufficient for proving key uniqueness was in this case not good enough, because Dafny could not reason backwards from it. Beyond toy examples like the selection sort, verifying software in Dafny requires a good understanding of formal verification.

Dafny also allows for compilation to other languages. This paper aimed to explore the process of compiling the verified written Dafny code into C#. Although the compiled code is functional, it is not easy to read. Using the code is also less clean than what you would typically write, and trying to integrate it into an already existing project is less than ideal. That being said, it is fully functional and there have been companies (mainly Amazon and Microsoft) willing to put these inconveniences to the side to verify aspects of their programs.

Dafny's deep integration of formal verification tools in its language is a great way however of learning formal verification. Simple to understand errors (albeit lacking in guidance at times) make finding mistakes somewhat easy. Dafny's **assert** statements are a great way of breaking up the proof by yourself which can then also be used to debug the code. However, when it comes to actual industrial usage, a comparison between Dafny and a verifier like KeY would be interesting to see how useful Dafny's compilation to other languages really is. Would Dafny's ease of use over other tools outweigh having to compile to another language? And how would that compare to a verifier that already works with that language where cleaner code could be written?

This language is still actively being worked on trying to improve several aspects of the language, but what we find lacking is a way of trying to integrate formal verification into more programming pipelines where possible. Dafny is as it stands a nice bridge, but as a stand-alone language it leaves a lot to be desired when other more mainstream programming languages are used like C++, Java and Python. Some future work could be to look at ways of improving Dafny's integration into other languages, rather than simply compiling to that language. In addition, a case study could be conducted in finding to what degree a tool like Dafny is worth using before the complexity of the program becomes too much or too big to completely verify. Right now, the sorting algorithm and key-value store were manageable enough and did not require too much, but there will be a point where the tool needs too much guidance so that verification becomes the bottleneck rather than the implementation.

6 Responsible Research

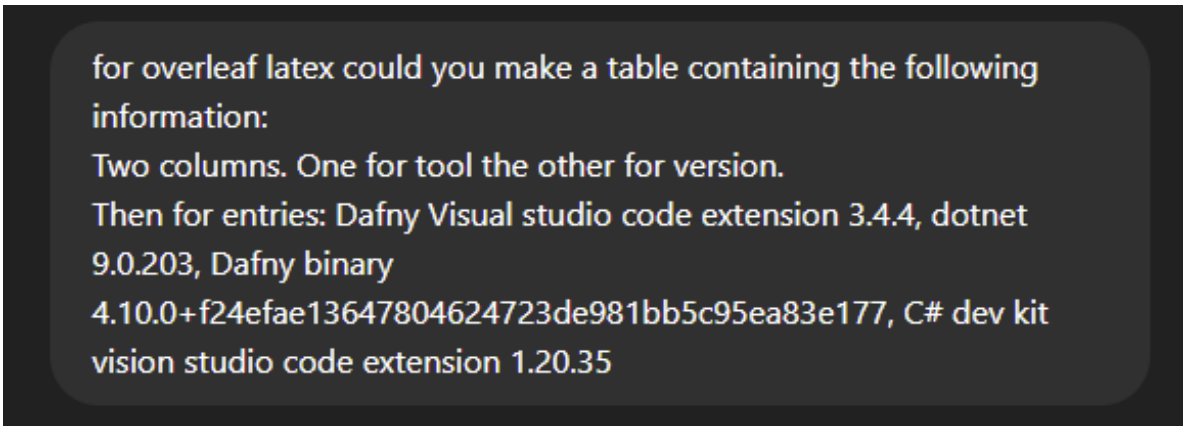
All written code and all generated code is available on my GitHub[5]. Next to that, Section 3 already outlined all the different versions of the tools that were used. It also offers a full explanation of the implementation of these algorithms. All this combined results in this research being reproducible.

Grammarly and the build-in grammar checker of Overleaf were primarily used for checking grammar and spelling of this paper. Large Language Models (LLMs), of which only ChatGPT 3.0 was used, were primarily used for help with writing Latex code for the figures and code blocks (see Appendix A). A few questions regarding the Dafny syntax were asked to ChatGPT 3.0 as well. However, the answers obtained were far from helpful as ChatGPT failed to write functional Dafny code, often mixing in syntax of other languages. Because of this, none of the Dafny code ChatGPT wrote was used.

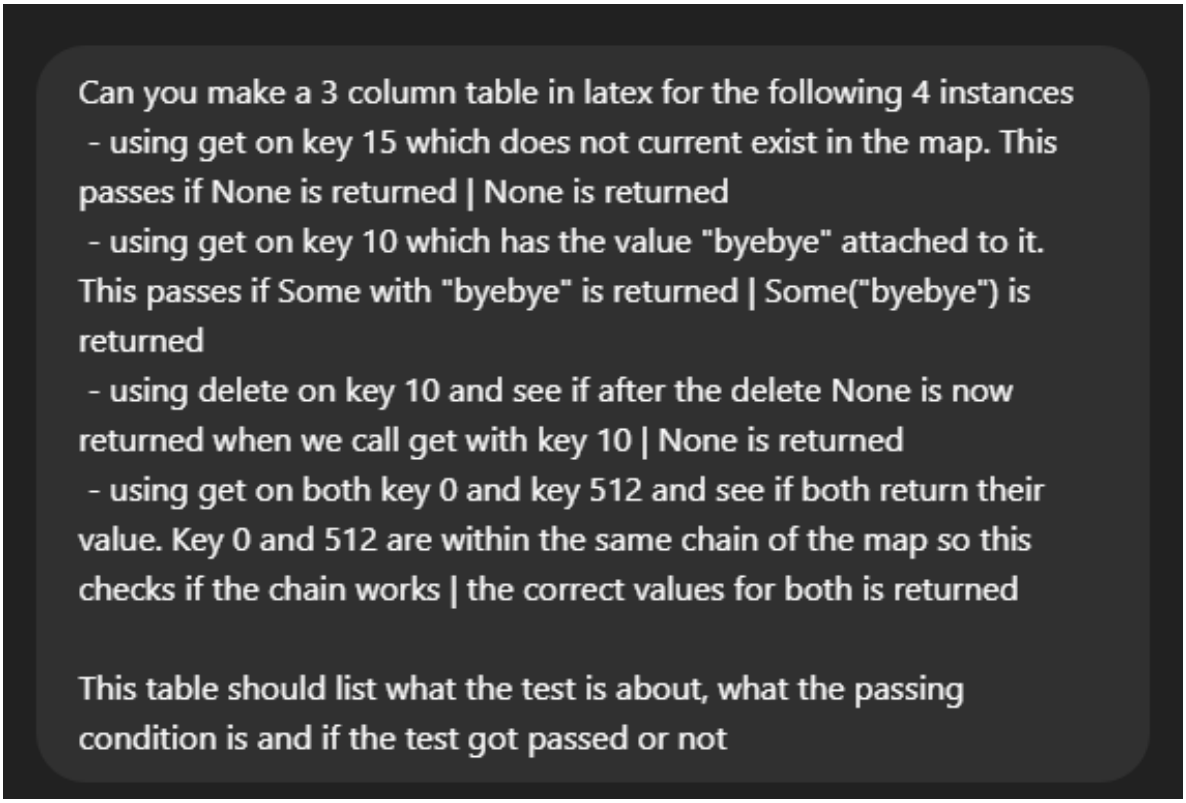
[5]https://github.com/jeroenkoel/Research-Project-Dafny.git

Appendix A LLM prompts used

For Table 1 the following prompt was used:



For Table 2 the following prompt was used:



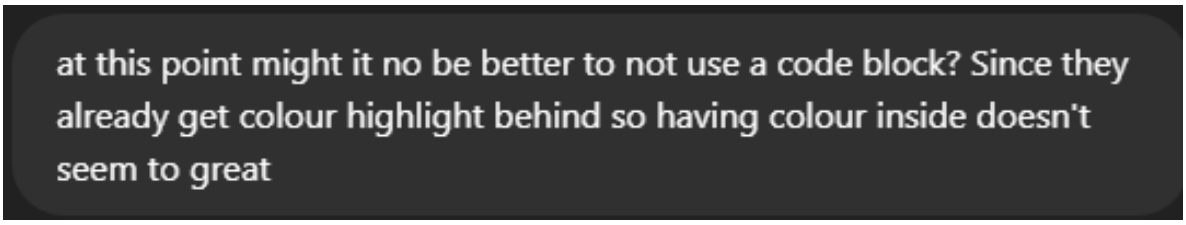
When it comes to creating the syntax highlighting the following prompts were used to generate the needed Listing variables:

so I currently have this for visualision a Dafny code block

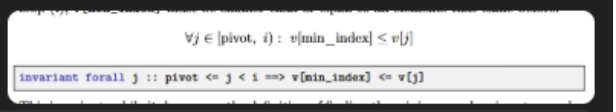
```
\begin{verbatim}
class Entry<K(==), V> {
    const key: K
    const value: V

    constructor(k: K, v: V)
        ensures key == k
        ensures value == v
    {
        key := k;
        value := v;
    }
}
\end{verbatim}
```

This of course has no syntax highlighting but I would like for it to be there. Unfortunately there is no package specifically for Dafny syntax highlight so I will have to do the colouring myself. How would I try to make it look as much like a code block as possible while being able to do the colouring myself?

Finally, for the creation of the colour coded comparison between math notation and Dafny syntax there isn't really one prompt that brought me to this situation. The prompt that ended up giving what I needed was the following:



However, before this I tried to make the colour highlight work on the Listing code blocks which ended up not working properly. Below all the prompt that came before this:



I have the following in a latex paper. I want to highlight what part of the math notation belongs to what part of the code but don't precisely know how in latex. Also because one part is inside of a custom code block. Any idea how I would put a coloured background behind the parts of the math and code that correlate. Here is the stuff for the listing code block btw:

```
\usepackage{xcolor}
\usepackage{listings}

\definecolor{keywordcolor}{rgb}{0.2,0.2,0.7}
\definecolor{stringcolor}{rgb}{0.6,0.1,0.1}
\definecolor{commentcolor}{rgb}{0.0,0.5,0.0}
\definecolor{backgroundcolor}{rgb}{0.95,0.95,0.95}
\definecolor{datatypes}{rgb}{1,0.5,0}

\lstdefinelanguage{Dafny}{
  keywords=[1]{class, const, constructor, ensures, requires, function, method, if, else, then, in, forall, returns, var, old, new, datatype, while, invariant, decreases, reads, modifies, predicate, exists},
  keywordstyle=[1]\color{keywordcolor}\bfseries,
  keywords=[2]{int, bool, string, K, V, T, Option, Entry, multiset, seq, array},
  keywordstyle=[2]\color{datatypes}\bfseries,
  sensitive=true,
  comment=[l]{//},
  morecomment=[s]{/*}{*/},
  commentstyle=\color{commentcolor}\itshape,
  morestring=[b]",
  stringstyle=\color{stringcolor},
  moredelim=[is][\bfseries\color{keywordcolor}]{}{}, % for manually highlighted words
}

\lstset{
  language=Dafny,
  backgroundcolor=\color{backgroundcolor},
  basicstyle=\ttfamily\small,
  frame=single,
  breaklines=true,
  tabsize=2,
  columns=fullflexible,
  keepspaces=true,
}
```

References

[1] E. W. Dijkstra, *Notes on Structured Programming*, en, Apr. 1970. DOI: 10.26153/TSW/53177. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/126640 (visited on 30/05/2025).

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," en, *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[3] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture notes in computer science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[4] The Alt-ergo community. "Alter-ergo repository." (2025), [Online]. Available: https://github.com/OCamlPro/alt-ergo (visited on 29/05/2025).

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte and H. Venter, "Specification and verification," en, *Commun. ACM*, vol. 54, no. 6, pp. 81–91, Jun. 2011.

[6] "Verification of object-oriented software: The key approach," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4334 LNAI, pp. 1–678, 2007, Cited by: 75. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-38849137985&partnerID=40&md5=f9a576036311188ed05e30b6cea5fb50.

[7] E. Cohen, M. Dahlweid, M. Hillebrand *et al.*, "Vcc: A practical system for verifying concurrent c," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5674 LNCS, pp. 23–42, 2009, Cited by: 383. DOI: 10.1007/978-3-642-03359-9_2. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-70349325391&doi=10.1007%2f978-3-642-03359-9_2&partnerID=40&md5=891dcf1236c7c77b0d4fdfea6bc6091b.

[8] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture notes in computer science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[9] The Dafny-lang community, *Dafny programming language*, 2025. [Online]. Available: https://github.com/dafny-lang/dafny (visited on 29/05/2025).

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, ser. Lecture notes in computer science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[11] T. Bordis and K. R. M. Leino, "Free facts: An alternative to inefficient axioms in dafny," en, in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science, Cham: Springer Nature Switzerland, 2025, pp. 151–169.

[12] M. Hicks. "How we built cedar with automated reasoning and differential testing." (2023), [Online]. Available: https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing (visited on 28/05/2025).

[13] C. Hawblitzel, J. Howell, M. Kapritsos *et al.*, "IronFleet: Proving safety and liveness of practical distributed systems," *Communications of the ACM*, vol. 60, no. 7, pp. 83–92, 2017. DOI: 10.1145/3068608.

[14] L. Li, M. Zhu, R. Cleaveland *et al.*, "Qafny: A quantum-program verifier," Cited by: 0, vol. 313, 2024. DOI: 10.4230/LIPIcs.ECOOP.2024.24. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85204953266&doi=10.4230%2fLIPIcs.ECOOP.2024.24&partnerID=40&md5=1dd67c1490d311d314c02de6f61d2ced.

[15] I. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," vol. 4085 LNCS, 2006, pp. 268–283. DOI: 10.1007/11813040_19. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-33749395393&doi=10.1007%2f11813040_19&partnerID=40&md5=a0b0a4cc124d76aa33a61531970b8520.

[16] The Dafny-lang community. "Dafny reference manual." (2025), [Online]. Available: https://dafny.org/dafny/DafnyRef/DafnyRef (visited on 29/05/2025).

[17] M. Leino. "Dafny power user notes." (2025), [Online]. Available: https://leino.science/dafny-power-user/ (visited on 08/06/2025).