The Dafny and Boogie Verification Tools

Philipp Schepper

February 9, 2018

Abstract

The functional correctness of programs was traditionally shown by a proof with pen and paper. But meanwhile tools were developed, which can verify a program automatically without user interaction.

This paper will give an introduction into the language and the automatic verification tool Dafny. Dafny can be used to verify class-based pointer programs written in the Dafny language. Features to introduce of a new layer of abstraction can be used in the verification process. By giving specific conditions one can proof the termination of a program. To understand the verification process, the underlying verification tool Boogie with its identically named language is introduced briefly.

The main part of this paper is based on [3] for Dafny and on [1] for the Boogie tool. Some parts and ideas are taken from [5].

1 Introduction

Developing correct algorithms can be decomposed into two main parts. First the design and writing of the algorithm and second the verification of the algorithm. By this verification, one usually shows the program fulfills given properties on all valid inputs. This last step, proofing the correctness of the algorithm, is the harder part.

Testing the algorithm will not necessarily lead to the conclusion the given program is correct since usually not all possible inputs can be tested. And finding appropriate test cases can also be hard (*e.g.* finding equivalence classes, case-coverage). Therefore, we have to find a way to formally *proof* the correctness of a program. This cannot be done by type checkers for example, these may only consider the types of method parameters or other variables. Instead, we have to do a formal static verification (it is called static since we do not execute the program, this would be a dynamic analysis, *e.g.* testing). This static verification takes the semantic of the programming language into account to show the functional correctness of the program.

For this verification, a program is converted into a first-order formula. This formula is handed to a satisfiability-modulo-theory (SMT) solver. This solver checks whether the formula is satisfiable or not, therefore the conditions in the program hold, or not.

In the past, several tools have been developed to attempt such a static verification. We will now introduce two of the three tools presented in [3].

For the ESC/Java tool Java code is annotated in comments written in the Java Modeling Language (JML). ESC stands for **E**xtended **S**tatic **C**hecking, which is some kind of type checking and not a formal verification. This is because it uses the compiled code for the verification and misses some errors in the program to make the use of the tool easier. For this, it also states some parts of the program as wrong despite they are correct. Therefore the tool is neither complete nor sound[1]. So the verifier does not detect all errors and considers some conditions as wrong although they are correct. There is the possibility to use algebraic data structures like (multi)sets, lists or maps. These algebraic datatypes can be used to have an abstract view of the data structure (*e.g.* represent a set, but the values are actually stored in a balanced tree).

The C# based, object-oriented language Spec# comes with a verifier for the language. In contrast to ESC/Java the annotations are a part of the code. The use of quantifiers is very restricted: While in ESC/Java we can quantify over different types, in Spec# we can only quantify over integers in an interval. There are no integrated algebraic datatypes as in ESC/Java. To detect infinite loops or to avoid endless recursion, termination metrics are needed. These techniques can be used to annotate a method such that the input parameters have to decrease between two nested calls. Therefore an ordering of the values has to be established and each decreasing sequence has to be finite. While ESC/Java supports termination metrics, Spec# misses them.

Another useful feature are side-effect free functions. Such a function represents a deterministic mathematical function. They can be used to compute auxiliary values which are needed during the verification of the program. Side-effect free means, they do not change other objects. If we treat the heap as an object, this restricts the structure of such functions very much: The function is not allowed to allocate new global objects or to create new local variables since this would change the heap too. Both tools, ESC/Java and Spec#, do not provide such side-effect free functions. They use so called pure methods, which are allowed to allocate new objects.

But there is quite another problem. Java and C#, on which Spec# is based, are huge languages in the following meaning: Huge libraries can be accessed and the language have many features. To use these libraries in a reasonable way in the code, they have to be verified and therefore to be annotated. But there are some features which are very hard to verify (*e.g.* concurrency). Even if the annotated code is shipped to the developer, such that the verifier does not have to verify the library and can simply use the pre- and postconditions, these conditions and annotations have to be found. Since each of the more than 6000 classes in the Java 9 JDK contains several methods, the verification of the library would take very long.

To avoid these problems, the imperative, class-based programming language Dafny has been developed [3, 5]. This language comes with a verifier called Dafny too. Dafny uses the automatic verifier Boogie [1] as an underlying system. Boogie comes with the

[1]Especially the last one holds also for the most automatic verifiers.

programming language BoogiePL[2][3]. The annotations in the Dafny program are part of the program one wants to verify. These programs are translated into Boogie programs. The main verification task is therefore done by the Boogie tool since it generates the logical formulas and passes them to the SMT-solver.

The rest of this paper is organized as follows: Section 2 gives a short introduction in the theoretical concepts of verification. The tool Dafny and its features are presented in section 3. In section 4 we will give a short introduction to the language Boogie and how a Dafny program is translated into Boogie code. In the last section 5 we will draw a conclusion about Dafny and Boogie.

2 Theoretical Foundations

For the verification, one cannot simply pass the program code (written in Java, C, ...) to a verifier since the verifier does not know what the program should do. Therefore we have to annotate procedures, loops and other parts of the program with first-order conditions. These conditions describe the behavior of the program. For example: What values does a procedure compute? How does it change objects? Are there side effects?

Conditions Given a program C. The input of this program has to satisfy given *preconditions* P before it can enter the program. As any other condition, they are usually given in predicate logic (first-order logic) (*e.g.* `x > 0`). The result being computed by the program has to satisfy some *postconditions* Q. These postconditions usually encode the required property, we want to compute (*e.g.* computing the square root: `result*result == x`). If Q holds for all variables after the computation, for which P had held, the program is correct. The Hoare-Logic[4] introduces a method to express such a successful execution by the following so called Hoare-Triple: $\{P\}\,C\,\{Q\}$

If we want to decide, which conditions hold after a program C with a precondition P has been executed, we compute the *strongest postcondition* $sp(P, C)$. This postcondition implies all other conditions holding after the execution. Analogously we can define the *weakest precondition* $wp(C, Q)$. These are the minimal conditions to the input such that the program C gets executed and the output satisfies the postcondition Q. In this manner we can encode a program into a predicate logic formula, the Verification Condition (VC). This formula is independent of the source language, notwithstanding that the behavior of different language constrains has to be encoded. In a perfect verification the VC satisfies the following property: The program is correct $\iff$ The VC is satisfiable. But usually we only get that a satisfiable VC implies the correctness of the program (*i.e.* we detect all errors in the source program, the completeness).

While this was first done manually, by writing down the proof with pen and pa-

[2]A newer version of BoogiePL is called Boogie2, see [4] for more information.
[3]In the following we will use Boogie for the verifier and for the language.
[4]By Charles Antony Richard Hoare

per[5], later methods were developed to do an interactive proof of the program. This means: The verifier computes something, then the user has to enter some data and the verifier proceeds. This is repeated until the program has been verified. Since more powerful SMT-solvers have been developed, the verification process was done *without* user-interaction; automatic verifiers were created.

Loops Verifying loops is a lot more complex than verifying if-statements since a loop can be executed more than one time. The verifier (and often the programmer too) does not know how often the loop has to be executed since this usually depends in a more or less complex way on the input. But the verifier has to verify the correctness of the program for all valid inputs.

To proof the correctness of loops the programmer has to find loop invariants. One can think of them as pre- and postcondition of a loop. They have to hold at the entry of the loop (*i.e.* before the first iteration) and they have to be reestablished after each iteration (*i.e.* are they maintained by the loop body). Only if these two verifications are successful, the loop invariant is correct. Since the verifier does not know how many times the loop will been executed, it can only use the loop invariants (and no other conditions) as a foundation to proof the correctness of the succeeding program code.

But the major problem with loop invariants is the hardness of finding them. While the easy ones can usually be found quickly, the more complex ones are not obvious to find and require hard work.

Dafny and Boogie In this paper the programs are written in Dafny. The Dafny tool translates these programs into Boogie programs (section 4.2.). The VC created by Boogie is handed to the SMT-solver Z3 [6]. This solver verifies the formula and provides a counterexample if the program could not be successfully verified. Using a special developer environment providing design-time-feedback, we can test more invariants as if the verification is only done after pressing a button. A schematic representation of this verification process can be found in Figure 1.

Since the Dafny and Boogie verifiers are complete, they find all errors in the code. But by their unsoundness the verifiers may state some parts of the program as wrong although they are correct. It is the duty of the programmer to distinguish these cases, which also arise from the translation of the Dafny code into Boogie code, from true errors in the program.

3 Features of Dafny

Dafny is a class-based, imperative programming language. Since the conditions are part of the code, they are marked by special keywords. Quantifiers for all types are supported. But the range of the quantified variables should be restricted by conditions. This helps

[5]Usually the conditions have not been constructed explicitly, instead it was done in an iterative way.
[6]Examples: `https://rise4fun.com/Z3/` and Source Code: `https://github.com/Z3Prover/z3`

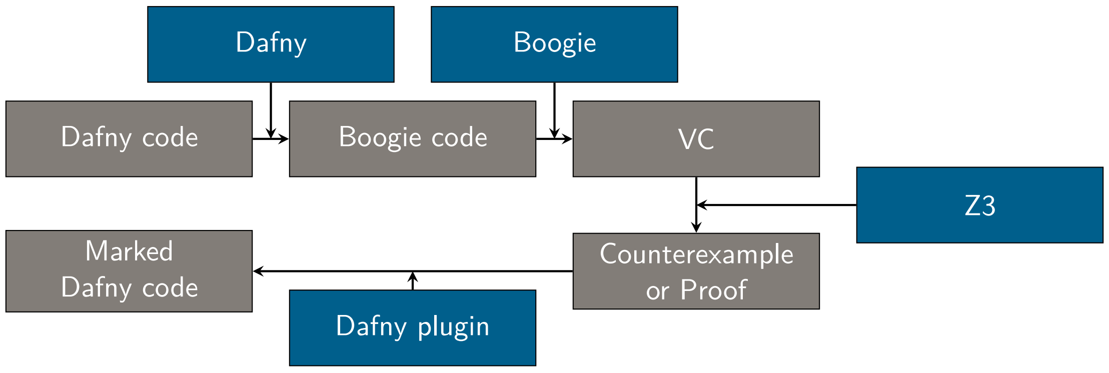

Figure 1: A schematic representation of the verification process

the verifier not to quantify over all possible inputs but to use only specific values (*e.g.* use only integers in a set and not all integers).

By using appropriate tools (*e.g.* Microsoft Visual Studio Code and the plugin Dafny-VSCode) a just-in-time verification[7] of the code can be obtained. Since the verification can be done during the writing of the algorithm, the programmer can see which of the program parts and conditions are probably wrong. This development process for the programs can be summarized as follows:

1. Write the code in Dafny.
2. Find pre- and postconditions if they are not given. Concretize and formalize them, to write them down as a first-order formula. See section 3.1.
3. Find loop invariants if necessary. Try different ones or change existing ones. Find termination metrics for loops and recursive functions. See section 3.2.
4. Do the verification: (a) If the code is wrong, correct the errors. (b) If loop invariants are missing, find new ones and correct the erroneous ones. (c) By the unsoundness of Dafny, correct program parts can be stated to be wrong. To determine, which of these conditions holds, is a tricky part. Repeat until the code is correct, but this is maybe not reported (unsoundness of Dafny).

3.1 Pre- and Postconditions and Predicates

The necessity of Pre- and Postconditions Methods or procedures are used in nearly every program. By the supported recursion, a method can call itself. Replacing the call of a method by the body of the method is not useful since we do not know how many times the method has to be inserted. This is because, the program has to be correct on all inputs. Therefore the pre- and postconditions summarize the methods. The verifier first checks if the postconditions hold after an execution of the method under the assumption of the preconditions (*i.e.* it verifies the method). After this the verifier checks for each

[7]Depending on how complex the code is and how easy it is for the SMT solver to verify the conditions.

call of the method if the preconditions hold. Then he assumes the postconditions for the output and uses these conditions to verify the rest of the program.

How to use in Dafny In Dafny preconditions are written after **`requires`**. Postconditions are introduced by **`ensures`**. The arguments have to be boolean expressions which are allowed to involve the parameters of the method and its return values. Complex or often used conditions can be summarized as a **`predicate`**. These predicates can be defined with several input parameters and have one implicit boolean return value (they are functions with a boolean return value, see section 3.3). The body of such a predicate consists of *one* first-order formula. The predicate in Example 2 checks if the array `a` is sorted on the interval given by the two integers. The **`method`** `swap` swaps the values at two positions in the array, while all other values stay the same. Therefore we have to access the original values of the array. This can be done by the **`old`**`(..)` operator. The expression in the parenthesis is evaluated before the computation of the method body starts. Therefore the computed values depend on the original values.

To describe which objects a method is allowed to change and which objects must stay unchanged, the **`reads`** and **`modifies`** clauses are introduced. Since functions are side-effect free, they have no modifies clause. Instead they can have a **`reads`** clause to determine on which objects the function may depend. The restriction in the reads and modifies clauses are at object-granularity. This means: The function/method may depend on any field of the objects given. So, if the method only changes the field `x` of object `o`, we have to write **`modifies`** `o`. To ensure no other fields have changed, one has to introduce a postcondition checking if the other values have stayed untouched. For objects, we can collect the references to objects on which they depend in a set. Then this set can be used as a parameter for the modifies clause.

```
predicate sorted(a: array<int>, start: int, end: int)
  requires a ≠ null
  requires 0 ≤ start ∧ end ≤ a.Length
  reads a {
    ∀ i,j • start ≤ i < j < end ⟹ a[i] ≤ a[j]
}

method swap (a: array<int>, i:int, j:int)
  requires a ≠ null
  requires 0 < i + 1 = j < a.Length
  modifies a
  ensures a[j] = old(a[i]) ∧ a[i] = old(a[j])
  ensures ∀ k • 0 ≤ k < a.Length ∧ k ≠ i ∧ k ≠ j ⟹ a[k] =
   old(a[k]) {
    a[i], a[j] := a[j], a[i];
}
```

Example 2: How to use pre- and postconditions

3.2 Loop Invariants and Termination Metrics

Loop invariants As we mentioned in section 2, the verifier does not know how often a loop is executed. Therefore the loop invariants summarize the computation of the loop. In Dafny, loop invariants are introduced by **`invariant`**. By the definition of loop invariants, they have to hold at the entry of the loop and they have to be maintained by the loop in each iteration. Similar to the annotations of methods, there is more than one invariant allowed. It is quite often helpful to split the invariants in different parts. Then the verifier can tell the programmer which invariants do not hold.

Termination metrics Although we do not know how often a loop gets executed, there is the possibility to provide some metric on how often a loop can be executed. This metric is also provided for (mutually) recursive methods. It is useful to avoid infinite recursion. By giving matching termination metrics after the keyword **`decreases`**, the verifier can detect if the recursion is finite or if the loop execution will terminate. But without these metrics the verifier cannot detect this. Between two nested calls of a recursive function, the argument of the decrease statement has to decrease.

To provide this decreasing chain of values, Dafny supports an ordering of the values of each type. Therefore only finite values are supported and each type has a least element. For integers we use 0 as the least element with regard to the decrease clause since the least element depends on the representable range of integers (*e.g.* $[-2^{31} \ldots 2^{31} - 1]$). For example `while (i < 9) decreases -i` would not work since `-i` is not bounded below by 0. Instead one must write `while (i < 9) decreases 9 - i`.

There is the possibility to provide more than one argument in such a decreases statement. They do not have to be of the same type (*e.g.* combine booleans, integer and sets). A list of expressions is interpreted as a lexicographic ordered tuple. Since Dafny supports the ordering for all types, we can use this metric also for user defined datatypes.

Example 3 is a recursive implementation of the insertion sort algorithm. The parameter `b` determines, up to which position the array should be sorted (see the postcondition). The method provides a termination metric based on `b` since the array is obviously sorted if `b = 1`. The loop has a decreases statement, too, to determine if the position where the values should be inserted is really decreasing (*i.e.* it is moved to the front). Therefore the loop will terminate.

3.3 Ghost Variables, Functions and Assertions

Ghost variables Sometimes it is very difficult to find the pre- or postconditions of methods and to formulate them in first-order logic. For example: What is the postcondition of the insert operation on a balanced tree? We could easier formulate these conditions if we would introduce a new level of abstraction. Dafny supports a technique to "build" a new layer of abstraction, which is only used to verify the program but is dropped in the final code. For this, **`ghost`** variables are introduced.

These ghost variable can be used in annotations like "physical" (*i.e.* non-ghost) variables. They can be changed like physical variables and are allowed to depend on any

```
method insertionSort(a: array<int>, b: int)
 requires a ≠ null
 requires a.Length ≥ b > 0
 modifies a
 ensures sorted(a, 0, b)
 decreases b {
  if (b = 1) { return; }
  insertionSort(a, b-1);
  var j: int := b-1;
  while j > 0 ∧ (a[j] < a[j-1])
   invariant b ≥ j ≥ 0
   invariant sorted(a, 0, j) ∧ sorted(a, j, b)
   invariant ∀ k,l • 0 ≤ k < j ∧ j < l < b ⟹ a[k] ≤ a[l]
   decreases j-1 {
    swap (a, j-1, j);
    j := j - 1;
  }
}
```

Example 3: A recursive implementation of insertion sort with termination metrics.

variable (this is usually the case). But the opposite direction is not allowed, a physical variable must not depend on a ghost variable. Since the ghost variables must not occur in control-flow statements (if or while conditions), they have no effect on the computation of the program and are therefore ignored by the compiler. So, they are only used for the verification of the program.

The class `OrderedSet` in Example 4 represents a set with ordered values. The data is stored in a (balanced) tree. To formulate the conditions for the `insert`, `remove` and `contains` methods, we use the ghost variable `values` which contains all values stored in the tree. Now we can write the postconditions for these operations in a very intuitive way with one short condition.

Functions In some cases the computation of ghost variables is not obvious and takes more than just one line. For this computation we could use methods. But methods are a part of the final program. If we use these methods only to compute values for ghost variables, we do not need the methods to be a part of the final compiled program. For this the concept of a **`function`** is introduced. These functions are a special type of method: They have only one anonymous return value and are not allowed to modify objects or to create new objects, neither local nor global. If the input of two calls is equal, the output is equal. Therefore they are deterministic and behave like real mathematical functions; they are side-effect free. If a function should be used in the compiled code, one can define a **`function method`**. This is a function which is part of the compiled code and can therefore be used to compute values for physical and ghost variables.

```
1  class OrderedSet {
     ghost var rep: set<object>; // The object-set representing
      the object
     ghost var values: set<int>; // The abstract view of the data
     var tree: Tree<int>; // The datastructure storing the values
5    function Valid(): bool
       reads {this} + rep {
       this in rep ∧ treeToSet(tree) = values
8    }
     method Init() modifies this
10     ensures Valid() ∧ fresh(rep - {this}) {
       rep := {this};
       values := {};
       tree := nil;
     }
15   method insert(value: int) modifies this
       ensures Valid() ∧ values = old(values) + {value}
     method remove(value: int) modifies this
       ensures Valid() ∧ values = old(values) - {value}
     method contains(value: int) returns (result: bool)
20     ensures result = (value in values)
   }
```

Example 4: A class representing an ordered set, while the data is stored in a binary tree.

We use the function `treeToSet(..)` in Example 5 to compute the value of the ghost variable `values` (see line 7 in Example 4).

Assert and Assume There are two other features that can be used to improve the verification process. **`assert`** `A` and **`assume`** `A` statements can be used to check if the given condition `A` is true. In contrast to the pre- and postconditions they can be used anywhere in a method. The verifier checks if the condition of the assert and assume statement does hold at the program point they are specified. If the condition is evaluated to true, the verifier proceeds and the statements behave as a no-op. But if the condition is evaluated to false, both statements differ in their behavior. In the case of the **`assert`** statements, the verification fails. Therefore the program is *not* correct. Since the verifier uses the conditions of the assume statements in the verification process, the evaluation of the **`assume`** condition to false results in a verification success for the following program path. This is because under the premiss **`false`**, one can imply all other conditions. By this behavior the program is verified to be *correct*. Such an **`assume false`** statement can be used intentionally in a program part which is not yet implemented. The assert statement in Example 5 checks if `treeToSet` works as expected on the given example.

Since the verifier uses the conditions in the verification process, the statements can be used to “feed” the verifier with new conditions. But the assume statement should be used very carefully since it could lead to an undesired verification success.

```
datatype Tree<T> = nil | Node(Tree<T>, T, Tree<T>)
function nodes<T>(tree: Tree<T>):int decreases tree {
  match tree
  case nil ⇒ 0
  case Node(l, v, r) ⇒ nodes(l) + nodes(r)
}
function method treeToSet<T>(tree: Tree<T>): set<T>
   decreases tree {
  match tree
  case nil ⇒ {}
  case Node(l, value, r) ⇒ treeToSet(l) + {value} +
   treeToSet(r)
}
method Main() {
  var n1 := Tree<int>.Node(nil, 1, nil);
  var n2 := Tree<int>.Node(nil, 2, nil);
  var n3 := Tree<int>.Node(nil, 3, nil);
  var n4 := Tree<int>.Node(n1, 4, n2);
  var n5 := Tree<int>.Node(n4, 5, n3);
  assert treeToSet(n5) = {1, 2, 3, 4, 5};
}
```

Example 5: The user defined datatype Tree and how to use datatypes.

3.4 Datatypes

Integrated Datatypes In addition to arrays and the basic datatypes **bool**, **int**, **real**, **char**, ...Dafny supports some other datatypes like sequences and sets. An unordered set of elements of type T has the type **set**<T>. These sets have to be finite and provide the operations union, difference, intersection, membership and (proper) subset (and therefore equality and non-equality). Complement is not implemented since the sets have to be of finite size. Cardinality is not supported since the sets are internally represented as maps. In Example 4 and 5 we show how to use sets. Since Dafny defines operations on sets via membership, it is sometimes necessary to provide the assertion **assert** s = t; for two sets s and t.

Sequences of elements of type T have type **seq**<T> and provide a very similar concept as sets. But they represent an ordered list of values. They support length, concatenation, slicing and selection of a specific element. In contrast to other datatypes, sequences are no objects but values, like integers for example. Therefore they must not appear in a reads or modifies clause. By there immutability (*i.e.* they cannot be changed), every “change” leads to a new sequence.

User defined Datatypes Beside these integrated datatypes, Dafny provides the possibility to create new datatypes. In Example 5 we define the data structure of a binary tree, which stores values in the (internal) nodes. By `nil` we denote the empty tree, with no nodes. `Node(..)` can be seen as some type of constructor. There is also the possibility to give more than two "constructors".

In the **`match`** statement one uses these "constructors" to distinguish between the structure of the object. Each "constructor" must be listed there, such that each way the object could be created is handled. This concept is required, since we cannot access the values of the datatype directly by fields. In addition each custom datatype comes with an ordering such that they can be used in the **`decreases`** statement.

Classes Since Dafny is a class-based language we can define new classes. But there is no sub-typing. Instead of constructors we create a new object by `var a := new ClassA` and initialize it by `s.Init()` (`Init()` is defined as any other method).

Since **`modifies`** works at object-granularity, one can use a (ghost) variable to store all objects a representative of the class depends on. To specify object invariants we use specific functions or predicates, usually called `Valid(..)`. The **`fresh`**`(..)` operation in a postcondition checks if the given objects have been created during the method call and have not existed before the entering of the method. See Example 4 for the generic usage.

We can use these concepts not only to maintain the specification of the object, but also to create a new layer of abstraction to proof the functional correctness of the program. For example, `treeToSet(tree) = values` in line 7 of Example 4 checks if the ghost variable actually contains all values of the tree. In conjunction with the ghost variable `values` this allows us an abstract view to the data.

4 Boogie

For the verification, the Dafny tool translates the Dafny program into Boogie code. Therefore different parts of the code will be changed since Boogie does not support all features of Dafny. This translation satisfies the following property: The correctness of the Boogie program implies the correctness of the Dafny program (*i.e.* all errors in the Dafny program are found). After this translation, the Boogie tool encodes the program into a VC and passes it to the SMT-solver. Apparently there is no reason to take a look at the Boogie code since the whole process is done without user interaction. But this is only true if the verification passes. If the verification fails, the verifier provides a cryptic error message (*e.g.* Figure 6). To understand these error messages we first take a look at the language Boogie and then show the translation from Dafny to Boogie by an example.

4.1 An Overview of the Language

Boogie is an unstructured, non-deterministic, intermediate verification language. Methods in Boogie consist of a declaration and an implementation. The implementation can

```
example.dfy(16,16): Error BP5005: This loop invariant might not be maintained by the loop.
Execution trace:
  (0,0): anon0
  example.dfy(14,3): anon6_LoopHead
  (0,0): anon6_LoopBody
  example.dfy(14,3): anon7_Else
  example.dfy(14,3): anon8_Else
Dafny program verifier finished with 2 verified, 1 error
```

Figure 6: An example error message thrown by the (Dafny) verifier.

also be a part of the declaration. This concept can be used to support methods with an unknown implementation or to override prior implementations. These methods are annotated with pre- and postconditions. These conditions can be marked as **`free`** such that their failed verification does not lead to a verification failure. As in the Dafny language, Boogie methods can be annotated with the modifies clause.

The assert and assume statements work as in Dafny and can be used everywhere in the program. While these statements give the possibility to state that a program goes wrong, they are primarily used to encode properties of the programming language and to give conditions for the verification.

There is the special command **`havoc`**. This command assigns arbitrary values to the variables mentioned as arguments. The range of the values is given by a condition in the optional **`where`** clause in the declaration of each variable. This concept of where clauses can also be applied to the in- and out-parameters of methods. For this we write the preconditions of the input parameters as a part of their declaration. This can also be applied to the postconditions and the return parameters.

A major difference to programming languages is the structure of the control-flow. In other languages the control-flow is structured by if-conditions and while-loops. In contrast, the if- and while-statements in Boogie are just some type of syntactic sugar, so they do not work as expected. This is because they are transformed into non-deterministic `goto` statements. These goto statements can be used like any other command in the program. With this command the program jumps to an arbitrary position given by the labels as the arguments of the goto command. Therefore the code is annotated with labels such that we can jump to specific points. By this non-determinism, the program cannot be executed since one cannot determine which path should be taken.

Because Boogie transforms the program into a VC, some restrictions for the language were made such that this transformation can be done easier and it is easier for the verifier to proof the correctness of the program. A major restriction is the explicit heap. While in other languages the heap is implicit and cannot be accessed directly, in Boogie it has been made explicit by the variable `Heap`. The heap is modeled as a two dimensional array which maps from objects and a field to the value of the field. The allocation of new objects is done by setting a specific field for each object in the heap. By the explicit construction of the heap we can combine the static fields of classes and the local fields of objects. This can be done if we treat the class as a specific object, which we can use to store data in the heap.

Another difference is the non existence of call-by-reference parameters. The parameters of a method are always values and not references. To imitate such a call-by-reference parameter we pass a copy of the object as a parameter and assign the modified value, which we get through an out-parameter, to the original variable. Axioms are supported to encode the behavior of different language features by formulas.

4.2 Dafny to Boogie

```
...
while (C) ensures I { S; }
...
```

(a) An example Dafny program

```
  ...
  goto LoopHead;
LoopHead:
  assert I;
  goto LoopDone, LoopBody;
LoopBody:
  goto Then, Else;
Then:
  assume ¬C;
  goto Continue;
Else:
  assume C;
  S;
  goto LoopHead;
LoopDone:
  assume false;
  goto Continue;
Continue:
  ...
```

(b) The first Boogie program

```
  ...
  goto LoopHead;
LoopHead:
  Heap#Old := Heap;
  assert I;
  goto LoopDone, LoopBody;
LoopBody:
  havoc x1,..,xn;
  assume I;
  goto Then, Else;
Then:
  assume ¬C;
  goto Continue;
Else:
  assume C;
  S;
  assert I;
  assume false;
  goto LoopHead;
LoopDone:
  assume false;
  goto Continue;
Continue:
  ...
```

(c) The “final” Boogie code.

Figure 7: The idea of the translation from Dafny into Boogie.

We will use loops to show in an exemplary way how the Dafny code is translated into Boogie code. As we noticed several times in this paper before, the verification of loops is more complicated than the verification of other code since the verifier does not know how often loops are executed. Therefore we need loop invariants for the verification. How the translation from the example Dafny code 7a into Boogie code is done, can be seen in the abstract and shortened example in Figure 7.

The computation of the loop depends only on the variables `x1 , ..., xn`. Since while-statements are only some kind of syntactic sugar in Boogie, they are not supported for the verification. Therefore we have to convert them into goto commands. The result of this step is shown in 7b. There one can see, how the condition of the loop has been expanded. By splitting up while-loops and if-statements new labels are introduced for each case. In each case we assume the corresponding condition for the following computation. The labels are used in the goto statement and by the verifier to print out the path to the violated condition in the error message, as seen in Figure 6. To verify the correctness of the loop invariant, we have to check if the loop invariant is maintained on all possible values satisfying the loop invariant. This last translation introduced the havoc command. The result is shown in 7c. As we can see, the verifier does not remember anything about the computation of the loop since the verifier checks if the loop invariant holds on arbitrary values.

5 Conclusion

Although there have been verifiers before Dafny, the tool comes with some features that can improve the verification of programs. The Dafny language is very slim, meaning there are no huge libraries which need to be verified or different types of loops. The language follows a very clear structure and leaves features out which are not needed (*e.g.* do-while-loops can be transformed into while-loops). The language supports custom classes but misses sub-typing and inheritance. These concepts are required to verify more complex code, maybe coming from another programming language, which is not supported at the moment. Since there are no constructors, one has to use the initialize methods. But this method is also intuitive and straight forward.

To get Dafny closer to be an object-oriented programming language, it has to support information hiding for the procedures and the variables. But to use Dafny as an verifier this would not be necessary since we could statically check if the conditions of the information hiding are followed.

The concept of invariants for objects could have been solved more intuitively. Instead of checking the validity of the objects in functions, the object-invariants should be a part of the definition of the classes and not of functions.

The interaction between Dafny and Boogie should, at some points, also be improved. In the case of a failed verification, the above mentioned plugin marks the violated condition in the code, while the Dafny command line tool provides an error message referencing the Boogie file. Therefore everyone writing Dafny code must understand the underlying Boogie, otherwise he will not understand the error messages. This Boogie file contains several hundred lines of code since the axioms encoding the language features have to be included. The programmer has to find the matching labels in the code the verifier is referencing. In a more complex case this can be annoying and time-consuming.

The counterexamples are also not formulated in a way such that a programmer without Boogie knowledge can always understand them. This is because Boogie introduces new variables used for the verification. Since they are no part of the Dafny code, the programmer cannot know how they have to be interpreted. To understand how they are used, the Boogie program has to be taken into account.

The Dafny tool should be improved in these cases without a change of the syntax of the existing language. In [3] some features of Dafny were explained whose syntax has changed during the evolution of Dafny and is now defined in a different way (*e.g.* definition of datatypes). This lead to an incompatibility of Dafny programs. In [3] it has been stated there is no compiler for Dafny and therefore the code is not executable. Luckily this does not hold anymore since a compiler has been build and the Dafny code can be executed.

While the current Dafny can already be used to verify programs, there is one major problem with all verifiers. Loops are a main component of programs and for their verification we need loop invariants. The search for the loop invariants is not much different from the time when the proof of correctness has been done manually. The new tools provide the possibility to check more invariants in the same time. But finding them is as hard as ever. But maybe there is the possibility that loop invariants are found sometimes by the verifier itself [2]. Then a program could be passed to the verifier and it would automatically check if the program is correct or not.

But until these tools can be used in productive environments, the verifiers have to be improved such that they can verify complex programs (object-oriented, ...) in a reasonable time. But a first step into the right direction is done by Dafny.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*, pages 364–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[2] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. *ICE: A Robust Framework for Learning Invariants*, pages 69–87. Springer International Publishing, Cham, 2014.

[3] K. Rustan M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness*, pages 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] K. Rustan M. Leino and Philipp Rümmer. *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*, pages 312–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] Microsoft Research. Getting started with dafny: A guide. `https://rise4fun.com/Dafny/tutorial/Guide`. Last checked: January 30, 2017.