Dafny with Traits:

Verifying Object Oriented Programs

M.Sc. Thesis

Reza Ahmadi

University of Tampere
School of Information Sciences
Computer Science / Software Development
M.Sc. Thesis
Supervisors:
Jyrki Nummenmaa, University of Tampere
K.Rustan M.Leino, Microsoft Research
October 2014

Abstract

Dafny is a programming language supporting verified high level programming. It has many features that a modern programming language has, like classes, generic classes, functions, and, methods. However, some aspects of object oriented programming do not exist in Dafny. For instance, it is not possible to write programs with classes and subclasses and then verify the subclasses. In order to enrich the language with the mentioned feature, this thesis introduces traits to Dafny. A trait in Dafny may introduce states, methods and functions with or without bodies. A class, then, inherits from a trait and may override the body-less methods and functions. There are also specifications for methods and functions in a trait that specify the intention of a particular method or function. In terms of the specifications, the class must provide the specifications, for annotating the functions and methods, possibly stronger. This has the drawback of repeating the specifications but it also increases readability as one can look at the class and immediately figure out what specifications govern the behavior of a method or a function.

The new feature, traits, provides polymorphism, information hiding, and reusability. Dynamic dispatch is now also available with the help of the introduced traits.

Keywords

Program Verification, Program Specification, Trait, Dafny, Boogie

Acknowledgments

I want to appreciate my supervisor Jyrki Nummenmaa for teaching me and guiding me throughout the studies and this work, and I want to appreciate Erkki Mäkinen for his comments on this work. I want to highly appreciate Rustan Leino at Microsoft Research for his all ideas and support during the implementation of this work. I want to give my special thanks to my beloved wife, Maryam, for supporting me and accompanying me in all the hard times, and last but not least, I want to thank my mother and my father for their love and support throughout my life.

Contents

1 INTRODUCTION

The software development process is supposed to transform the captured user requirements into working software. Formulating the recorded user requirements into a formal specification of the system and then studying those helps understanding the behavior of the system and figuring out probable misunderstandings in the requirements. Regardless of whether a formal specification of the system to be built exists, the challenges with the correctness of the software programs have to be dealt with. The reason would be that the programs and thereby their specifications include details that the specification of the system does not deal with. The significant role of specification of the programs would be to avoid "bugs" in programs and to ensure that the written programs conform to their associated specifications. The specifications of a program denote the intention of its developer. This problem is not new as it has been initially dealt with by Dijkstra [1976].

At the level of the programs, different program specification languages are offered, let alone some tools that capture the specifications to some extend automatically from a written program, which are beyond the scope if this work. The specifications not only describe the intention of the programs but are also used by automatic program verifiers to ensure the programs' correctness [Burdy et al., 2005]. Some examples of specification and verification tools are Java Modeling Language (JML) [Burdy et al., 2005], C# specification and verification language (Spec#) [Barnett et al., 2011], and C verification tool (VCC).

Introducing verification structures to traditional languages is somewhat problematic. First of all, the structures seem more or less artificial. Secondly, the programmers of those languages do not usually annotate their programs. Thirdly, adding those structures to written programs do not help developers in the software development phase as the specifications are supposed to be added to the programs as the programs grow.

Verification languages such as Boogie, on the other hand, are designed with verification in mind. Even though such languages include structures for "normal" programming, they lack the convenience developers may find in modern programming languages due to the structures and the different facilities that exist in the modern programming languages.

Dafny is a programming language that supports high level programming and, at the same time, includes the verification structures. It uses a program verifier, under the hood, for program verification. It supports different contract specifications like precondition, postcondition, object invariants and other specification contracts that many other verifiers support. It also supports defining classes and modules.

Dafny does not support interfaces, mix-ins or traits as a mean for polymorphic methods in types. Hence, it cannot support any form of dynamic dispatch. Basically, it is possible to have classes in Dafny and all classes have a superclass, object, but it is not possible to inherit from other types in the language [Leino, 2010].

As inheritance is a popular concept in programming, the limitation of not supporting such a feature can be seen as a short-coming of Dafny. As Dafny inherently does not support inheritance, it seems suitable to add a lightweight mechanism to support inheritance, which, in this work, is the trait. Shortly, a trait is like an interface with possible implementation for methods or functions.

The traits in Dafny will be able to include functions, methods and fields. Methods and functions may or may not have bodies. A trait may not be instantiated itself and may

not include a constructor. A class can implement a trait. In case a method or function in a trait does not have a body, the inheriting class can override that method or function and provide a body for that. If a trait has a body, the inheriting class also inherits the body by default and the class is only allowed to override the body-less methods or functions. In terms of the programs specifications, the overridden methods and functions must provide their own specifications anew. However, the specification may be stronger than the specifications in the parent trait.

Listing 0 shows a simple example of a trait in Dafny. The listing also shows how to inherit from a trait and how to override trait's members. In Listing 0, E is some expression whose result type is appropriate for the result type F.

```
trait t1
{
   var f: int;
   function method F(x: int): int
     requires x < 100;
     ensures F(x) < 100;
}

class c1 extends t1
{
   function method F(x: int): int
     requires x < 100;
     ensures F(x) < 100;
   {
      E
   }
}
```

Listing 0: A sample trait in Dafny

Classes, methods, functions, algebraic data types and datatype constructors in Dafny are generic [Leino, 2010]. However, this work does not support generic traits.

2 BACKGROUND

This chapter introduces the concepts and background required for understanding the discourse of this work. Initially, some of the most significant program verifiers are introduced, with an emphasis on those that Dafny uses. Then, different means of inheritance in Object Oriented Programming (OOP) is briefly introduced in order to also describe traits in OOP languages, and finally Dafny is discussed more as it is the focus of this work. The reader is expected to be familiar with OOP and some knowledge about compilers is helpful. Basic concepts of program verification are discussed during introducing Dafny and its features.

2.1. Program Verification

In this section program verification concept and some of the verification languages are discussed briefly. In addition, Dafny as a programming language and automatic program verifier is discussed. Boogie, an intermediate programming language which is used as a bridge between formula solvers like Z3 and higher level programming languages like Dafny, is also introduced and some of its features are discussed as those features will be used during translation of Dafny programs to Boogie. Translating Dafny programs basically happens in the Dafny translator and will be explained later.
Traditionally, program verification used to be an interactive work with a proof assistant that needed a user to have considerable knowledge about the prover and many tactics to apply [Leino, 2010]. Even before that program verification was done using pen and

paper. Automatic program verifier, however, has no interaction with the user during the solving procedure, in which satisfiability-modulo-theories (SMT) solvers are used in order to verify the input programs [Leino, 2010]. SMT-based program verifiers take a program with provided specifications, then, analyze the program and produce proper messages about the program. If the program does not satisfy the specifications, the verifier produces error messages about violated specifications, for example, if a method's postcondition or precondition does not hold [Leino, 2010]. The verification may work in the background while writing programs if the verifier has an IDE like Dafny programs that can be written in Visual Studio (VS) [Leino et al., 2014].

Basically, behind the scene, SMT solvers are fed with formulas in first order logic, and the solver determines if the input formula is satisfiable or not. It is the responsibility of the language to take the user input program and provide appropriate output to feed the solver for a proof. For instance, Z3 [De Moura and Bjørner, 2008], Microsoft SMT solver, has clients like Boogie [Leino, 2008], an intermediate verification language, PEX, an automatic program analyzer, and others. Boogie is used by those languages to play the role of a bridge between the languages and the Z3. Programs in Dafny, Spec#, HAVOC and some other languages are translated, first to Boogie and from Boogie to Verification Conditions (VCs) which are accepted by Z3 in order to verify the source program. Figure 0 shows how Boogie plays the role of an intermediate verification language [Boogie at Microsoft Research].

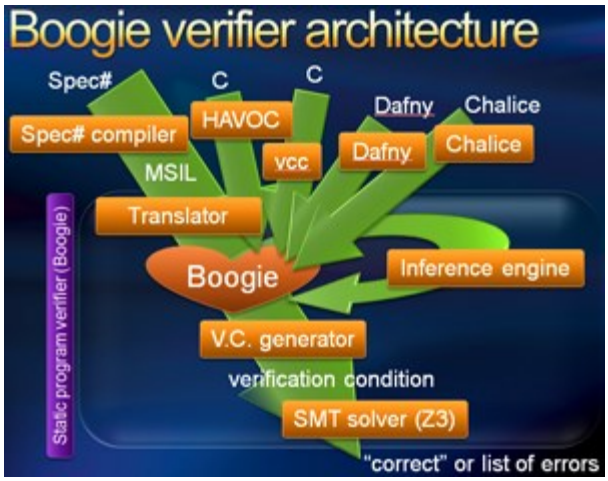

Figure 0 - Boogie verifier architecture

In Dafny, for example, input program (.dfy file) is translated to a Boogie program by Dafny translator and the result is sent to Boogie. Boogie produces VCs from the input program and sends those to Z3. Finally, Z3 tries to prove the formulas or VCs in which it produces verification error messages in case there are any violations [Herbert et al., 2012].

2.1.1. Spec# and JML

Spec# programming system is an extension to C# programming language, which is also supported in Visual Studio, to enrich the language with a program verification mechanism. Spec# has its own compiler and an automatic program verifier, Boogie. It does support dynamic checking of specification in addition to static checking. [Barnett et al., 2005]

In its static checking, one method at a time is checked by the verifier in terms of the specification violations and runtime semantics such as null-dereference, array index out of bound and division by zero, and errors are reported. Basically, testing cannot be replaced fully by static checking as static checking cannot check, for example, if the

requirements have been covered fully by the program or some other checks like stack overflow. [Barnett et al., 2005]

In its dynamic checking, Spec# compiler emits runtime checks on the target code using the recorded specifications [Barnett et al., 2005].

One interesting feature of Spec# is its non-null types. Based on that, by defining a non-null variable, null value is excluded from possible values of a variable. So, non-null reference types can be dereferenced safely and there is no need for a nullity check at runtime. [Barnett et al., 2005]

In comparison to Dafny, Spec# is an Object Oriented language with specifications but it does not have mathematical constructs like termination metrics (termination metrics ensure that a method or function or a loop terminates and does not loop forever), algebraic data types, ghost variables (ghost variables help in verification and are not part of the compiled output), built-in sets and sequences and few others which are necessary for full functional correctness verification. [Leino, 2010]

Java Modeling Language (JML) [Burdy et al., 2005] is a specification language for Java. JML also has common features of a specification language. However, its verifiers are either interactive (not automatic) like KeY tool or they are automatic like ESC/Java but do not perform full verification [Leino, 2010].

2.1.2. Boogie

In this subsection, different Boogie language constructs are explained briefly as they will be used in Dafny's verifier changes explained in the implementation section.

In program verification, a standard method is to take the source program, and generate logical formulas or VCs from that program. Then, the validity of the generated VCs implies that the source program satisfies its specified correctness properties. [Leino, 2008]

Modern programming languages split the task of program verification into two steps. In the first step, source program is transformed to an intermediate language. The resulting program is still close to a program than formulas. Then, the result is transformed to logical formulas, and then the formulas are solved by an SMT solver. [Leino, 2008]

Boogie is an intermediate verification language. Many languages such as Spec#, Eiffel and Dafny translate their source program into Boogie for program verification. [Leino, 2008]

A Boogie program's form is shown in Listing 1.

```
Program ::= Decl*
Decl ::=  TypeDecl | ConstantDecl | FunctionDecl | VarDecl | AxiomDecl  | ProcedureDecl | ImplementationDecl
```

Listing 1: Boogie program's form [Leino, 2008]

In Listing 1, some of the productions are typical for most other languages and their function is as their names suggest. Some of them, however, need some explanations:

Axiom declarations propose properties about functions and constants declarations [Leino, 2008]. For example, consider the declarations shown in Listing 2.

```
type Book;
const cpp: Book;
function price(Book) returns (int);
…
var favorite: Book;
function $IsGhostField<T>(Field T) : bool;
axiom price(cpp) == 100;
```

Listing 2: Some declarations in Boogie [Leino, 2008]

In Listing 2, the last line denotes an axiom that says *price* returns 100 for *cpp*.

Type in Boogie is a primitive type, an instantiated type constructor, a polymorphic map or a synonym for an already defined type [Leino, 2008]. Samples of primitive types are *int* and *bool*. Listing 3 shows examples of different type declarations in Boogie.

```
type Fiction a;
const m: [Fiction Book] Book;
const n: <a> [Fiction a] a;
```

Listing 3: Types in Boogie [Leino, 2008]

In Listing 3, *m* is a map from *fiction books* to individual *books* and *n* is a map from any kind of *fiction* to individuals of that kind [Leino, 2008], so *n* is called a *polymorphic map*.

Predicate is a function that returns a Boolean. In Listing 2, *$IsGhostField* is a predicate.

Variables are declared using *var* keyword as *favorite* has been declared in Listing 2.

Procedures in Boogie are declared using *procedure* keyword which denotes a set of execution traces that are specified by preconditions and postconditions [Leino, 2008]. A *procedure* also has an associated implementation which is declared using *implementation* keyword [Leino, 2008] as shown in Listing 4.

```
procedure SetNewBook(n: Book);
modifes favorite;
ensures favorite == n;
implementation SetNewBook(n: Book)
{
   favorite := n;
}
```

Listing 4: Procedure declaration in Boogie [Leino, 2008]

Assert introduces an expression that holds in every correct state of a program. An *assert* statement is used to check an expression. For example, if the expression x=y / (a-b) from a source language is translated to Boogie, it might generate the following statements [Leino, 2008]:

```
assert (a-b) != 0; x := y / (a-b);
```

Havoc statement is used to assign a random value to a variable. The value can be restricted by *axioms* or *assume* statements [Leino, 2008]. More explanations come below.

Assume introduces an expression that holds in every feasible trace of a program [Leino, 2008]. One usage of that statement is to accompany *havoc* statement to control the value that *havoc* assigns to a variable [Leino, 2008]. For example:

```
havoc x,y ; assume (x+y) > 10;
```

Quantifiers are supported by Boogie both in universal and existential forms [Leino, 2008]. Their usages are just like in Dafny so no more explanations are given here.

Functions in Boogie can appear in two forms. One form is just like in Listing 2, when a function appears without a body but its properties are described by axioms. Another possibility is to declare a function with a body as shown in Listing 5 [Leino, 2008].

```
function attrs F(args) returns (res)
{
  E
}
```

Listing 5: Function with a body in Boogie [Leino, 2008]

2.1.3. Dafny

Dafny is a programming language with specification support. Specifications are preconditions and postconditions, termination metrics, loop invariants and frame specifications [Leino, 2010]. The language also supports ghost variables and mathematical functions, which are just tools to assist the developer in the verification and are not a part of the final compiled assembly. So, the compiler ignores the whole specifications and ghost variables. For program verification, Dafny translates the program into a Boogie program [Barnett et al., 2006], the intermediate verification language, and from the Boogie program to verification conditions using Boogie.

Verification conditions are input to an SMT Solver, Z3, to prove them. If the Boogie program is correct, it implies that the Dafny program is also correct. Otherwise, any violations in the verifications are returned as verification errors [Herbert et al., 2012].

Dafny compiler uses C# compiler behind the scenes in order to build .Net MSIL byte

code from the source program [Herbert et al., 2012]. Basically, the source Dafny program is transformed to a string and then C# CodeDom API is used to generate the target .dll or .exe files from the generated string. If there is a *main* method in the source program then an .exe application is made by C# compiler. Otherwise, the output will be .dll libraries.

A difference between Dafny and other verification languages is that Dafny was created with verification in mind. That is, Dafny was created from scratch with programming and verification features included [Herbert et al., 2012] unlike languages like Spec# and JML. Therefore, Dafny programs are cleaner than of those verification tools whose specification mechanism is added to an existing language [Herbert et al., 2012].

A schematic view of how the Dafny system works is shown in Figure 1 [Herbert et al., 2012]. The figure shows how the Dafny compiler and verifier communicate with the .Net compiler and Boogie in order to produce executable code and to verify the input program.

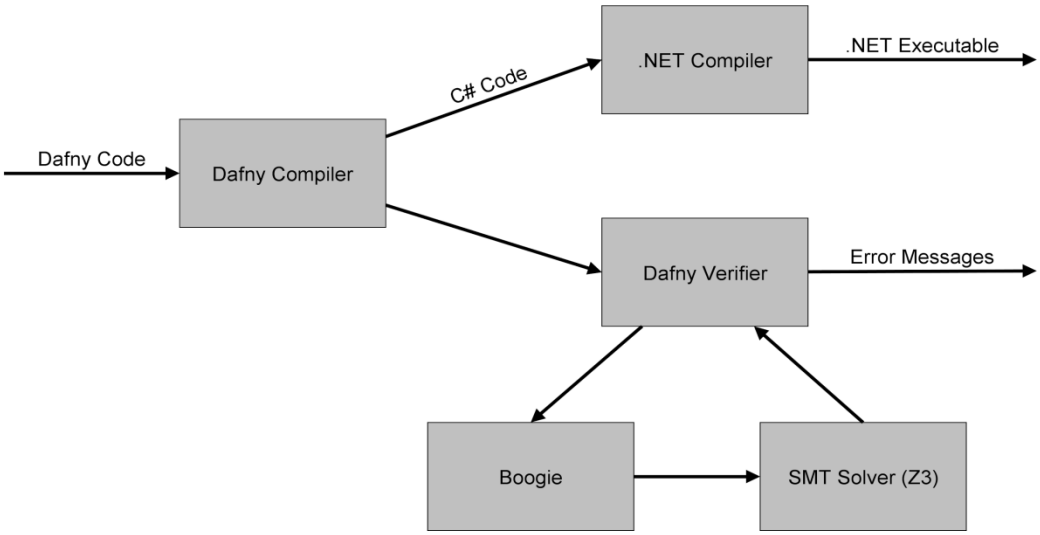

Figure 1- Dafny system

In the following subsections, different constructs of Dafny are discussed to better

understand the course of this thesis. For more details about the constructs, see [Leino, 2010; Koenig et al., 2012].

Annotations

A method specification is a contract between the implementation of the method and the caller of the method. A method specification has a precondition and a postcondition. A precondition specifies what a caller must establish on entry to the method. The implementor can assume the precondition. Postcondition is what the implementor must establish on exit of the method's body. The caller can assume the postcondition when returning from the invoked method. When reasoning about a method, only the specifications of that method, the contract between the caller and the callee, are considered.

Annotating functions with preconditions and postconditions are described further below in Dafny's style.

In Dafny *ensures* and *requires* are used for declaring a postcondition and precondition annotations, respectively [Koenig and Leino, 2012].

Dafny has features to prove termination. It proves, for instance, that a while loop ends finally, and a recursive function is not invoked forever. For example, if there is a series of calls to a function, and a natural number is assigned to each function call, and we ensure that every successive call will decrease that number, then Dafny can prove that the function terminates. Using the *decreases* clause one can introduce a termination metric [Herbert et al., 2012]. A method or function may have more than one termination metric. In that case Dafny looks at the first metric. If it decreases, then Dafny does not look at the next metrics. Otherwise, it looks at the next ones until it finds a metric that decreases. For the termination proof to be possible, one of the metrics must decrease whether the rest of the metrics do not change or even increase [Herbert et al., 2012].

If a method needs to modify a non-local object or a function needs to read a non-local

object, then the method and function must specify the required sets of objects as a *modifies* clause or *reads* clause. Those clauses are called *dynamic frames* in Dafny as an expression which denotes the set of objects may evaluate to a different one, dynamically at runtime, as the state of the program changes [Herbert et al., 2012]. Methods or functions receive frame violation errors if they try to access a non-local object which is not included in the sets of objects which are introduced by *modifies* and *reads* clauses. An example program annotated with dynamic frames is shown in Listing 6.

Methods and Functions

Methods in Dafny declare executable code in a unit just like procedure or function in other languages [Koenig and Leino, 2012]. A method *Withdraw* which takes out money from an account is shown in Listing 6.

```
class Account
{
  var balance: int;

  method Withdraw(val: int)
    requires balance >= 0 && balance >= val;
    ensures Valid();
    modifies this;
  {
      balance := balance - val;
  }

  method AddCredit (val: int)
    requires val >= 0;
    requires Valid();
    ensures balance >= 0;
    modifies this;
  {
     balance := balance + val;
  }

  function Valid(): bool
       reads this;
  {
     balance >= 0
  }
}
```

Listing 6: Declaring and annotating a method in Dafny

In Listing 6, the methods have been annotated by preconditions, postconditions, and dynamic frames. Postcondition and precondition are introduced by *ensures* and *requires* clauses, respectively, just before the body of the methods. Preconditions and postconditions are Boolean expressions.

It should be noted that Dafny remembers only the body of the method that it is working on. Hence, when Dafny is trying to prove that a method satisfies its specifications, it knows about that methods' body. But later, when execution is in other methods or functions, it will just look at the specifications of those functions or methods [Koenig and Leino, 2012]. It means that if a method satisfies its specification, Dafny will not check the method's body in every call. This helps Dafny to work with a reasonable speed [Koenig and Leino, 2012]. For example, in the method shown in Listing 7, Dafny is not able to prove that the second assert statement holds even though it is logically correct.

```
method GetMax (x: int, y: int) returns (z: int)
    ensures z >= x && z >= y;
{
    if (x >= y)
    {
        z := x;
    }
    else
    {
       z := y;
    }
}

method Testing()
{
   var x,y,z;
   x := 5;
   y := -5;
   z := GetMax(x, y);
   assert (z >= y && z >= x);   // 1st assert
   assert (z == x);             // 2nd assert
}
```

Listing 7: Dafny only remembers the body of the current method/function

Again the reason is that Dafny does not care about *GetMax* body and only looks at its specifications when running *Testing* method. The first assert holds as it says what *GetMax* specification says. Now, if *GetMax* specification is replaced with "ensures x >= y ==> z == x;", then the second assert holds. The reason is that the new specifications simply imply the second assert statement.

Functions are another unit of execution in Dafny. Function body must include only one expression which is a Boolean expression with a suitable result. For example, in Listing 8 the result is of type int.

```
function Negate (x : int) : int
{
    if x < 0 then x else -x
}
```

Listing 8: A simple function

As for termination metrics, Listing 9 shows how to annotate a function with termination metrics, which is introduced by *decreases* clause. The function calculates factorial of *n*. The *decreases* clause ensures that the function terminates finally as *n* decreases by every successive call.

```
function Factorial(n: int) : int
  requires n >= 0;
  decreases n;
{
     if (n == 0 || n == 1) then 1 else  n * Factorial(n - 1)
}
```

Listing 9: Annotating a method by termination metrics

Unlike methods, functions can appear in specifications only [Koenig and Leino, 2012]. Also unlike methods, Dafny does not forget the implementation inside a function's body when running other functions [Koenig and Leino, 2012].

For example, in Listing 10, assert statement holds.

```
method Testing()
{
  var x;
  x := 5;
  assert (Negate (x) == -5); // this holds
}
```

Listing 10: Function's body is evaluated every time it is called

Functions will not be included in the compiled assembly. They are just tools that assist verifying the programs [Koenig and Leino, 2012].

Function calls are allowed only in specifications (like explained before and used in an *assert* statement). However, there are cases when we need to call a function from other places than the specifications, in the code. We can make this by defining a function method. An example is in Listing 11.

```
function method max(a: int, b: int): int
{
   if a > b then a else b
}

method Testing()
{
    var a : int;
    a := max(2,3);
}
```

Listing 11: Function Methods [Koenig and Leino, 2012]

Assertion

An *assert* statement indicates that a particular expression must hold when the execution of the program reaches that part of the program [Koenig and Leino, 2012]. Otherwise, the program stops with an error message. The associated statement in Dafny is *assert* which can be placed anywhere in the code [Koenig and Leino, 2012].

Assert statements are usually used to check if a desirable expression holds in different parts of the code [Koenig and Leino, 2012]. An example is shown in Listing 12.

```
function GetMax2 (x: int, y: int) : int
{
    if x >= y then x else y
}
method Testing()
{
  var x,y,z;
  x := 5;
  y := -6;
  assert (GetMax2(x,y) == x);
}
```

Listing 12: Asset statements [Koenig and Leino, 2012]

Quantifiers

A quantifier is either universal or existential. Universal quantifier is used to quantify all elements of a set or array and its result is true if its expression holds for every individual item in the target collection [Koenig and Leino, 2012]. Existential quantifier is the same but its result is true if its expression holds for at least one item in the target collection. In Dafny, universal quantifier is declared using *forall* and existential quantifier using *exists* keywords [Koenig and Leino, 2012]. Listing 13 shows an example of a universal quantifier.

```
method Find(a: array<int>, key: int) returns (index: int)
requires a != null;
ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
  ...
}
```

Listing 13: Universal quantifiers [Koenig and Leino, 2012]

Classes

A Class in Dafny is the unit of abstraction like in other languages. Classes can include functions, methods, lemmas, and fields [Koenig et al., 2012; Leino, 2010]. A class is defined as in Listing 14.

```
class MyClass
{
       //member declarations
}
...
var c1 := new MyClass;
```

Listing 14: Class definition and initializing [Koenig and Leino, 2012]

Instantiating classes in Dafny are like in many other languages, using *new* keyword in an appropriate place in the code.

Modules

A module in Dafny, as its name suggests, hold classes that can be imported by other modules. A simple module and its usage in Dafny are shown in Listing 15.

```
module YY {
  class ClassG { }
  class SS {
     static method GetSum(x:int, y:int) returns (z: int)
      ensures z == x + y;
     {
      return  x+y;
     }
  }
  class MyClassY {
    method M() { }
    method P(g: ClassG) {
    }
  }
}

module XX{
  import JJ = YY;
  class C1
  {
    method FF (){
      var j, k,l : int;
      j := 10;
      k := 10;
      l := JJ.SS.GetSum(j,k);
      assert (l == 20);
    }
  }
}
```

Listing 15: A simple Module and its usage [Koenig and Leino, 2012]

2.2. Inheritance in OOP languages

Inheritance is a mechanism for incrementally refine and modify new programs using existing programs without altering the existing ones [Taivalsaari, 1996]. By Inheritance, new classes introduce new properties, along with inherited ones, in order to create new, modified or refined classes. Inheritance is a fundamental mechanism for code re-use. However, it has its deficiencies [Snyder, 1986]. For example, inheritance can lessen encapsulation or data abstraction [Snyder, 1986]. One problem is this: A class designer hides the implementation and internal data structure of a class and provides an external interface as a contract between the clients and implementer of that class. The designer can then freely re-implement the class as long as the changes made on the class preserve the interface contract and the changes are upward compatible [Snyder, 1986]. So, the designer is benefiting from the abstraction which simplifies program evolution and the maintenance [Snyder, 1986]. Now, if the inheritance is added, a class must play two different roles: as an instance for clients and as a parent for new classes. Hence, by inheriting instance variables from ancestor(s) the designer would not have full freedom to alter the implementation. To maximize the benefits of encapsulation one should lessen the exposure of implementation details to the clients [Snyder, 1986].

Inheritance in different languages has different applications than its apparent usage as a mechanism to enrich the child classes. Taivalsaari [12, p. 11] classifies the applications of inheritance into four types:

- *Inheritance for implementation*. This type of inheritance happens to enrich the child class with properties of its parent as those properties are needed for the new class under construction. The main emphasis for this sort of inheritance lies in reducing the required effort for development, saving the required storage space and for faster

code execution. In this type of inheritance, the descendant may restrict some inherited features (Cancellation) which is common in Smalltalk-80. It may override some features in order to provide better or more suitable implementation than its parent (Optimization) or just simply inherit the features without any change on those as all of those functionalities in the superclass are the right ones for the child class (Convenience) [Taivalsaari, 1996].

- *Inheritance for combination*. This sort of inheritance uses multiple inheritance to combine existing abstractions. For instance, inheriting *StudentTeacher* from *Student* class and *Teacher* class. This type of inheritance may involve combining classes with equal importance that leads to many conflicts to be handled by the subclass. A solution for this scenario is to define roles for every class rather than combining them. Mix-ins inheritance is also considered in this class of inheritance which has been appeared first in Flavors language [Taivalsaari, 1996]. More details about mix-ins are in the following sections.
- *Inheritance for inclusion*. In languages which do not offer modules as a container for grouped classes, this sort of inheritance is used to simulate a module. The solution is to create a class and add proper functions in that class. Then, in order to *import* those functions into a new class, it is either possible to instantiate an object from that class and use it into the new class or inherit from that class so the functionalities will be *imported* automatically into the new class [Taivalsaari, 1996].
- *Other uses of inheritance.* Other uses of inheritance are not very common. One possible usage is *inheritance for generalization*. The basis of this inheritance is to create a more general class from a parent rather than a more specific one. Sometimes it is more convenient to create a more general class from a specific class than from other general classes. For instance, one may create a *Stack* class from *Deque* class [Taivalsaari, 1996].

Inheritance has been also used in different forms. Different types of inheritance can be classified into single, multiple and mix-in inheritance [Schärli et al., 2003]. More details about this classification are in the following subsections.

2.2.1. Single and Multiple Inheritance

In single inheritance one class may extend exactly one other class, whereas in multiple inheritance (MI), one class may extend one or more than one other classes. MI is problematic in some ways. One issue is the "diamond problem" [Bracha, 1992] where, for example, classes B and C both have a superclass D (as in Figure 2) and A extends both those B and C. There is a method M in D which both B and C have implemented but it has not been overridden by A. Now, the problem is that if A emits a call to M then which method will be called eventually, the implemented one in B or in C.

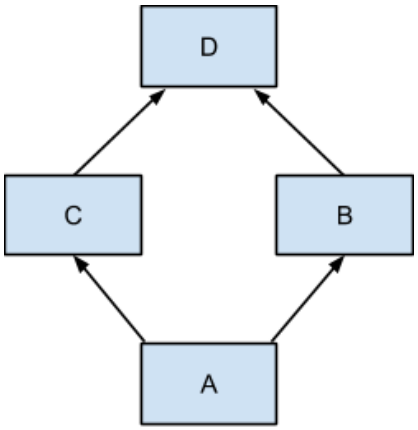

Figure 2- Diamond problem

C# and Java do not allow MI, where one class may inherit from multiple other classes [Ducasse et al., 2006]. These languages, however, provide a different mean for supporting MI. It is like this: class A may extend class B and may extend one or more interfaces. This method does not cause the diamond problem as in the inheritance

hierarchy there is at most one implementation of a particular method although there may be more than one declarations of a single method.

2.2.2. Mix-ins

Mix-ins are non-instantiate pieces of behavior which are added to other existing classes in order to attach a property [Bracha, 1990]. Mix-ins are syntactically like ordinary classes but they have different usages. In mix-in inheritance, one or more properties are attached to a class just like it happens using MI, but in mix-ins the MI problems do not exist anymore. The reason is that mix-ins are not included in the inheritance hierarchy [Bracha, 1990]. So, for example, the *diamond* problem never happens. In other words, mix-in classes do not have ancestors or subclasses [Bracha, 1990].

One issue in mix-ins appears when there are, for instance, two inherited mix-ins that have a property or method with an identical name. In that case, if a class inherits from both, then one property or method will override the other one. This happens implicitly by the compiler as the mix-in composition is linear [Ducasse et al., 2006]. That means, there will be particular precedence for inherited mix-in methods in a particular class.

Mix-ins have become very popular recently. In C#, for instance, there is only one way for class composition, that is, that is single inheritance. However, as a mean for more flexibility in class compositions, there is an open source project which developed mix-in, re-mix, for C# to support mix-in inheritance [remix at CodePlex].

2.2.3. Traits

Traits are used for code reuse just like mix-ins, but they are believed to be more appropriate than mix-ins for code reuse [Ducasse et al., 2006]. Some of the differences between traits and mix-ins are the following:

- Multiple traits can be applied to a class in one operation whereas this happens in mix-ins incrementally [Ducasse et al., 2006].
- Composition order in traits is irrelevant whereas that in mix-ins is linear [Ducasse et al., 2006] which causes the problem of implicit overriding of identical methods as mentioned earlier. In traits identical methods give rise to an error so the programmer is responsible for fixing the error while in mix-ins the compiler choses one method automatically which may not be the one that the developer means.
- Traits contain only methods whereas mix-ins contain states in addition to methods [Ducasse et al., 2006].
- In traits there is an interesting feature to resolve conflicts that arise from identically named methods that come from combining multiple traits. That is to add glue code in the level of the class to override those identical methods, and as the methods in the class have more priority in order to resolve the conflict [Ducasse et al., 2006].

  For example, the specification of traits in Scala is as follows [Odersky et al., 2004]:

  1- May have abstract and concrete methods.

  2- May have states.

  3- A class can extend exactly one other class but it may extend any number of traits.

  For example, both the following declarations are valid in Scala:

  - Class C1 extends C2 with Trait1, Trait2, Trait3 {body}.
  - Class C1 extends Trait1 with Trait2, Trait3 {body}.

2.3. Tools for Changing the Language

In this section tools and technologies which are involved in the implementation part of this work are introduced and briefly discussed.

Coco/R

Coco/R is a tool for generating the parser and the scanner for some languages. When running Coco/R, it gets the source language grammar specification (which is also called an attributed grammar), and two .frame files (scanner.frame and parser.frame). The .frame files include static code, just like a template, which generated code is injected into it based on the input grammar specification file (.atg file). Frame files are available for different languages like C#, Java, C among others [Hanspeter et al.]. After executing Coco/R with appropriate input parameters, it generates a parser and a scanner (lexical analyzer and syntax analyzer). Figure 3 shows how Coco/R employs .frame files and the .atg file in order to generate the parser and the scanner.

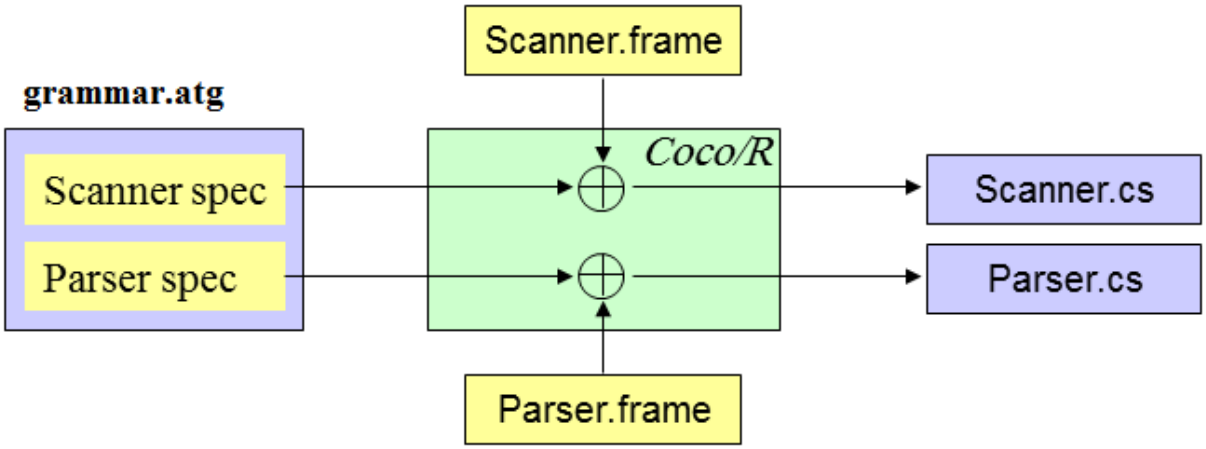

Figure 3- Coco/R inputs and outputs [Hanspeter et al.]

The language grammar is specified in the form of EBNF [Hanspeter et al.]. A simple example of a grammar for simple arithmetic *Add* expressions is in Listing 16.

```
COMPILER AddEx
CHARACTERS
        digit = '0'..'9'.
TOKENS
        number = digit {digit}.
IGNORE '\r' + '\n'
PRODUCTIONS
        AddEx                           (. int n; .)
        = { "add"
                Expr<out n>             (. Console.WriteLine(n); .)
          }.

        Expr<out int n>                 (. int n1; .)
        = Term<out n>
          {'+'
                Term<out n1>            (. n = n + n1; .)
          }.
        Term<out int n>
        = number                        (. n = Convert.ToInt32(t.val); .)
        .
END AddEx.
```

Listing 16: A simple grammar specification in CoCo/R

In Listing 16, expressions between "(." and ".)" are semantic actions. Those codes are executed by the parser when applying every associated rule. For example, when parser is about to apply AddEx rule, it will instantiate a variable by running "int n;".

In order to scan and parse a source program using the generated parser and the scanner, there is a need to have another program which invokes the parser with an input source program. The C# program in Listing 17 takes care of that. That is actually the *compiler* of the language.

```
using System;
class Compile
{
    static void Main(string[] arg)
    {
        Scanner scanner = new Scanner(arg[0]);
        Parser parser = new Parser(scanner);
        parser.Parse();
        Console.Write(parser.errors.count + " errors detected");
    }
}
```

Listing 17: A simple Compiler

Finally, it is possible to compile a source program using the built compiler. Input.txt consists of one line of program as 'add 2+3' (without quotes):

```
compiler.exe input.txt
0 errors detected
```

Note that there must be scanner.frame and parser.frame files available in the root as Coco/R needs those.

For Dafny, the file Dafny.atg specifies Dafny's grammar and is modified accordingly in order to add *trait* support to Dafny.

3 ADDING TRAITS TO DAFNY

In this chapter required changes to Dafny source code[1] in order to add traits to the language are discussed.

3.1. Design

In this section, Dafny's objects interaction is first shown using a sequence diagram. Then, all the changes required for adding *trait* to the language are briefly introduced.

3.1.1. Dafny Classes and Objects

Figure 4 shows Dafny's execution sequence diagram. The diagram shows main components of Dafny language and the required sequence of calls for an execution from starting Dafny.exe to verifying and compiling the input program.

[1] Download Dafny source code from https://dafny.codeplex.com/SourceControl/latest

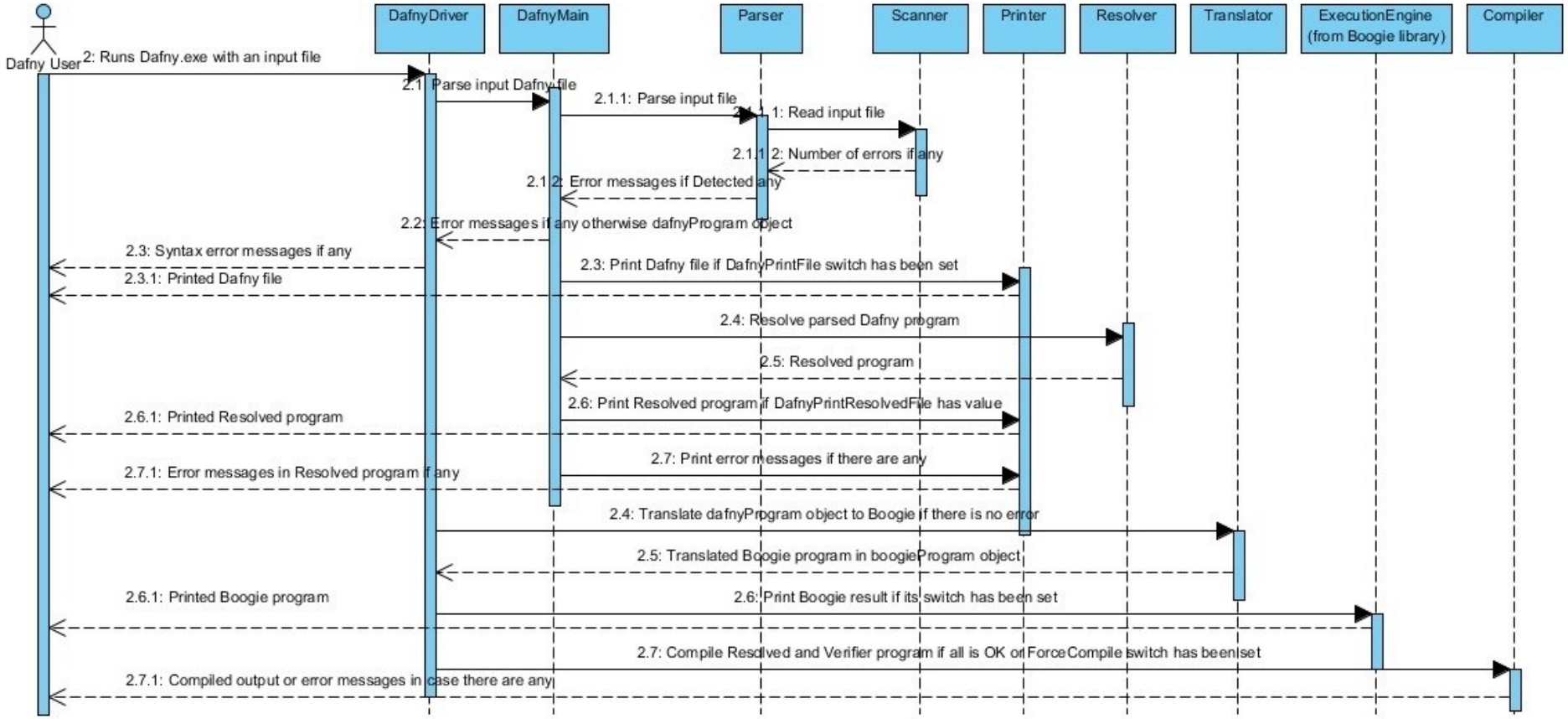

Figure 4- Dafny's execution sequence diagram

A short description of the objects in Figure 4:

- The scanner and the parser are responsible for reading the input program and creating the Abstract Syntax Tree (AST) from the Dafny programs.
- Resolver is responsible for semantic analyzing, type checking and resolution of a program.
- Compiler is responsible for producing executables. If there is a *main* method in the source file then the output will be an ".exe" file. Otherwise the compiler produces C# .dll files.
- The translator is responsible for translating Dafny program into Boogie program for the verification process.
- DafnyMain is the main object that initiates calls to other objects to parse and resolve the input program.
- DafnyDriver is a class in a Console application which communicates with the user and emits calls to start an execution. It also initiates calls to verify and translate the resolved program if no errors are detected during the parsing and the resolving.

3.1.2. Required Changes

The required changes are classified under the following major classes:

- *The scanner and the parser changes* which are required in order to enable Dafny to scan and parse the new keywords, *trait* and *extends*. These changes are done using CoCo/R tool for generating a new *parser* and a new *scanner*.
- *Resolver changes* to adopt facilities like polymorphism to enable dynamic dispatch, and to disallow declarations like *constructor* or *new* keywords in a trait. Also to merge trait members into an associated child class, and to make sure that body-less methods and functions have been implemented in a child class (in case a class extends a trait).

- *Compiler* changes to allow the compiler to compile programs that include traits. These changes are to create proper C# interfaces and classes that the new declarations rise to.
- *Translator changes* to translate Dafny programs with traits to appropriate Boogie programs. These changes are mostly to add new Boogie procedures such as *override specification check* and some axioms to the target generated Boogie program. The recent changes are to make sure that a call to a method or function in the trait will be also properly handled by the overridden method or function in the class. As Boogie does not know anything about inheritance and subclasses, there is a need to add procedures, axioms, and predicates to the target Boogie program to verify such Dafny programs.
- *Other changes* are related to the *printer* and *syntax highlighters* for the new keywords. More details will be given in the next sections.
- *Test* suits have been added for every new feature, from minor changes to the resolver to changes to the translator. For every minor change to the language, the whole old test suits in addition to the new test suits are executed to make sure that the new change does not break anything.

3.1.3. Proposed Traits for Dafny

Dafny Traits are like interfaces with possible implementations for methods and functions. Dafny traits also have fields or states. They are very much like Scala traits. The purpose of Dafny traits is to allow fine-grained reuse as it is in Scala. In Dafny, it is possible to extend a class with only one trait. It is not possible to extend a class with another class. Currently, it is not possible to extend a trait. Extending the current design to also support the mentioned features, however, would not be too different from the current implementation.

3.2. Implementations

In this section all the discussed changes on the previous section are discussed step by step.

3.2.1. Changes to the Parser and the Scanner

The parser and the scanner changes are began by changing the grammar specification, Dafny.atg. Then, CoCo/R tool must be executed using the newly changed grammar specification to generate the new scanner and the new parser. A summary of the changes in Dafny.atg is the following:

- A change to add a new grammar production for the traits, *TraitDecl.*
- A change to make it possible to declare traits in a module.
- A change to allow classes *extend* a trait using *extends* keyword.

The *TraitDecl* grammar production is added using the declarations shown in Listing 18.

```
TraitDecl<ModuleDefinition/*!*/ module, out TraitDecl/*!*/ trait>
 = (. Contract.Requires(module != null);
          Contract.Ensures(Contract.ValueAtReturn(out trait) != null);
          IToken/*!*/ id;
          Attributes attrs = null;
          List<TypeParameter/*!*/> typeArgs = new List<TypeParameter/*!*/>();
          List<MemberDecl/*!*/> members = new List<MemberDecl/*!*/>();
          IToken bodyStart;
    .)
       SYNC
       "trait"
       { Attribute<ref attrs> }
       NoUSIdent<out id>
       [ GenericParameters<typeArgs> ]
       "{"                                              (. bodyStart = t; .)
    { ClassMemberDecl<members, true>
    }
    "}"
       (. trait = new TraitDecl(id, id.val, module, typeArgs, members, attrs);
     trait.BodyStartTok = bodyStart;
     trait.BodyEndTok = t;
    .)
 .
```

Listing 18: Adding one more grammar production to the grammar specification to allow trait declaration

Also optional *extends* modifier is added to *ClassDecl* production so that the parser is able to also parse classes with an *extends* declaration. Listing 19 shows the change.

```
ClassDecl<ModuleDefinition/*!*/ module, out ClassDecl/*!*/ c>
= (. Contract.Requires(module != null);
     Contract.Ensures(Contract.ValueAtReturn(out c) != null);
     IToken/*!*/ id;
     List<IToken>/*!*/ traitId=null;
        Attributes attrs = null;
     List<TypeParameter/*!*/> typeArgs = new List<TypeParameter/*!*/>();
     List<MemberDecl/*!*/> members = new List<MemberDecl/*!*/>();
     IToken bodyStart;
  .)
  SYNC
  "class"
  { Attribute<ref attrs> }
  NoUSIdent<out id>
  [ GenericParameters<typeArgs> ]
  ["extends" QualifiedName<out traitId>]
  "{"                                              (. bodyStart = t; .)
  { ClassMemberDecl<members, true>
  }
  "}"
  (. c = new ClassDecl(id, id.val, module, typeArgs, members, attrs, traitId);
     c.BodyStartTok = bodyStart;
     c.BodyEndTok = t;
  .)
.
```

Listing 19: Adding optional “extends” modifier to the ClassDecl production

For the full changes related to the grammar specification please refer to Dafny.atg in the source code at CodePlex.Com.

At the moment the traits are disallowed to declare type parameters. The restriction is made on the resolver and not on the grammar specification. An error message follows in case a trait is declared with a type parameter. Actually, at the moment, the resolver enforces that restriction and it is not applied on the parser in case one wants to support the feature later.

Changing Dafny.atg followed by generating the new parser and the new scanner using Coco/R.

There are also other changes:

- Changes to the DafnyAst.cs file, which includes Dafny Abstract Syntax Tree classes, so to add TraitDecl. The new class is a subclass of ClassDecl as given in Listing 20.

```
public class TraitDecl : ClassDecl
{
    public bool IsParent { set; get; }
    public TraitDecl(IToken tok, string name, ModuleDefinition module,
      List<TypeParameter> typeArgs,
      [Captured] List<MemberDecl> members, Attributes attributes)
        : base(tok, name, module, typeArgs, members, attributes, null) { }
}
```

Listing 20: Adding TraitDecl subclass to DafnyAST (Dafny Abstract Syntax Tree) file

Since the new class is a subclass of ClassDecl, many things work already. The parser will create an instance of TraitDecl or ClassDecl depending on which keyword it parses.

3.2.2. Changes to the Resolver

The changes on the resolver are the following:

- Some restrictions to disallow *new* and *constructor* keywords for a trait and to disallow type parameters for a trait. The later restriction forces the resolver to raise an error in case the user declares a generic trait.
- A change on the Dafny's *type checking* to allow an object of a class to be assigned to a trait that the class implements it. It will be used for polymorphism in the Dafny's type checking which will be discussed more in the following subsections.
- Inheriting the members of the trait.

Applying the Restrictions

In order for the resolver to disallow *new* keyword, the code snippet in Listing 21 is added to the resolver.

```
Type ResolveTypeRhs(TypeRhs rr, Statement stmt, bool specContextOnly, ICodeContext
codeContext) {
...
   var cl = (ClassDecl)udt.ResolvedClass;
   if (cl is TraitDecl) {
     Error(stmt, "new cannot be applied to a trait");
   }
...
}
```

Listing 21: Disallow new keyword for a trait initialization

Also to disallow constructors for a trait, the code snippet in Listing 22 is used:

```
ModuleSignature RegisterTopLevelDecls(ModuleDefinition moduleDef, bool useImports) {
...
   cl.HasConstructor = hasConstructor;
   if (cl is TraitDecl && cl.HasConstructor)
   {
       Error(cl, "a trait is not allowed to declare a constructor");
   }
...
}
```

Listing 22: Disallow constructor for a trait

Polymorphism in Dafny's Type Checking

The outcome of the change on the type checker is to allow the program in Listing 23. It verifies and compiles fine. Dynamic dispatch happens in the *PolymorphicMethod* method as the compiler does not know what will be the exact type of *j* parameter at the compile time as it can be any type that implements T.

```
trait T {}

class C1 extends T { }
class C2 extends T { }
```

```
method PolymorphicMethod(j: T)
{
  var jj := j;
}

method TestPolymorphicMethod(c1: C1, c2: C2)
{
   PolymorphicMethod (c1); //this is valid
   PolymorphicMethod (c2); //this is valid
}
```

Listing 23: Polymorphism in Dafny

However, the program in Listing 24 is not legal as it is not valid to assign a trait to a class and there is not a dynamic check to allow it.

```
method Bad(j: T) returns (c: C)
{
  c := j; // error: cannot assign a J to a C
}
```

Listing 24: Illegal assignment

The change to the type checker for supporting polymorphism, as in Listing 25, is to add one more case to the existing *if conditions* in *UnifyTypes* method, which is responsible for determining equivalent types.

```
public bool UnifyTypes(Type a, Type b) {
Contract.Requires(a != null);
Contract.Requires(b != null);
…
   var aa = (UserDefinedType)a;
   var bb = (UserDefinedType)b;

   …
   else if ((bb.ResolvedClass is ClassDecl) && (aa.ResolvedClass is TraitDecl))
   {
      return ((ClassDecl)bb.ResolvedClass).Trait.FullCompileName ==
         ((TraitDecl)aa.ResolvedClass).FullCompileName;
   }
   else if ((aa.ResolvedClass is ClassDecl) && (bb.ResolvedClass is TraitDecl))
   {
      return ((ClassDecl)aa.ResolvedClass).Trait.FullCompileName ==
         ((TraitDecl)bb.ResolvedClass).FullCompileName;
   }
…
```

```
}
```

Listing 25: Changing the type checker to support dynamic dispatch

Inheriting Trait Members

The last change on the resolver is to inherit members from a trait by a class that extends the trait. The implementation resides in the *InheritTraitMembers* method in the resolver in Dafny's source code. The inheritance implementation involves the following steps:

- To merge members of the trait with the class that implements that trait.
- To check if the class members have provided their own specifications. In this design the method/function specifications are not inherited by a class from its base; it is mostly due to readability and simplicity of the design. The specification in the class though may be stronger than that in the base trait.
- To check all body-less methods or functions to make sure that they have been implemented in the child class.

For merging the members, we first discuss about the resolver, before going through its implementations.

The resolver registers all class declarations parsed by the parser in a field called *classMembers*. That field is a dictionary object which denotes a mapping from a class declaration to a mapping from string to member declarations:

```
Dictionary<ClassDecl, Dictionary<string, MemberDecl>> classMembers;
```

So, keys in the dictionary are class declarations and the values are a collection of members (the class members). The dictionary type ensures that there are neither duplicated class declarations in a program nor duplicated member declaration in a single class declaration.

After registering the class declarations by the resolver, if any class has inherited a trait then trait members must be merged with the class members. The merge will do the following:

- If the trait includes a member *m*, then the merge will look up *m* in the class and do the following:
  - If there is a member *m* also in the class, then:
    - If *m* in the trait has body then an error is reported by the resolver. The present design does not allow overriding implemented members.
    - If *m* in the trait is body-less then the user has overridden the body-less member. This is allowed and no error is reported.
  - If there is not member *m* in the class then mapping *from the name m to the member in the trait* is also added to the name-member mapping of the class. In previous Dafny implementations, the name-member mapping was just a mapping of names to member of that class, but with this implementation there are also name-member mapping of the inherited members.

Listing 26 shows the implementations of the merge.

```
void InheritTraitMembers(ClassDecl cl)
{
    Contract.Requires(cl != null);
    //merging class members with parent members if any
    if (cl.Trait != null) {
        var clMembers = classMembers[cl];
        var traitMembers = classMembers[cl.Trait];
        foreach (KeyValuePair<string, MemberDecl> traitMem in traitMembers)
        {
            MemberDecl clMember;
            if (clMembers.TryGetValue(traitMem.Key, out clMember)) {
                if (traitMem.Value is Method) {
                    Method traitMethod = (Method)traitMem.Value;
                    Method classMethod = (Method)clMember;
                    if (traitMethod.Body != null
                        && !clMembers[classMethod.CompileName].Inherited)
                        Error(classMethod, "a class cannot override
                                            implemented methods");
                    else {
                        classMethod.OverriddenMethod = traitMethod;
                    //adding a call graph edge from the trait method to that of class
                        cl.Module.CallGraph.AddEdge(traitMethod, classMethod);
                        ...
                    }
                } else if (traitMem.Value is Function) {
                    Function traitFunction = (Function)traitMem.Value;
                    Function classFunction = (Function)clMember;
                    if (traitFunction.Body != null
```

```
                        && !classMembers[cl][classFunction.CompileName].Inherited)
                        Error(classFunction, "a class cannot override
                                              implemented functions");
                    else {
                        classFunction.OverriddenFunction = traitFunction;
                    //adding a call graph edge from the trait method to that of class
                        cl.Module.CallGraph.AddEdge(traitFunction, classFunction);
                        ...
                    }
                } else if (traitMem.Value is Field) {
                    Field traitField = (Field)traitMem.Value;
                    Field classField = (Field)clMember;
                    if (!clMembers[classField.CompileName].Inherited)
                        Error(classField, "member in the class has
                                    been already inherited from its parent trait");
                }
            } else {
                //the member is not already in the class
                // enter the trait member in the symbol table for the class
                clMembers.Add(traitMem.Key, traitMem.Value);
            }
        }//foreach
        ...
    }
}
```

Listing 26: Merging trait members with its child class members

In terms of the class members' specification, class members must provide their own specifications anew possibly strengthened. There is a check in the *InheritTraitMembers* method to make sure if all the overriding methods or functions in the class have provided those specifications, in case the overridden method has provided any. Listing 27 shows a part of that check. Please refer to CodePlex.com for the full code.

```
void InheritTraitMembers(ClassDecl cl) {
  Contract.Requires(cl != null);
   …
   //class method must provide its own specifications
   in case the overriden method has provided any
   if ((classMethod.Req == null || classMethod.Req.Count == 0) &&
      (classMethod.OverriddenMethod.Req != null
      && classMethod.OverriddenMethod.Req.Count > 0))
   {
     Error(classMethod, "Method must provide its own Requires clauses anew");
   }
   …
   //class function must provide its own specifications
   in case the overriden function has provided any
```

```
      if ((classFunction.Req == null || classFunction.Req.Count == 0) &&
         (classFunction.OverriddenFunction.Req != null &&
         classFunction.OverriddenFunction.Req.Count > 0))
      {
        Error(classFunction, "Function must provide its own Requires clauses anew");
      }
      …
}
```

Listing 27: Checking class members specifications

The last check is to make sure if all of the body-less members of the base trait has been implemented by the child class. It is a simple loop that goes through all the members of the trait. If there is a member *m* in the trait which is body-less and there is not a member *m* with the same signature and name in the related class, an error is reported. The code fragment is not shown here. Please refer to CodePlex.com for the full implementation of the *InheritTraitMembers* which implements that check.

3.2.3. Changes to the Compiler

Dafny's compiler translates from the Dafny AST into a string which represents a C# program. The string is passed to the C# compiler in order to produce an executable (.exe) or a .dll library.

As explained in Subsection 3.1.3 traits may include fields and functions and methods with or without bodies. So, traits are richer than interfaces in C# and the introduction of the traits affects the Dafny's compiler which resides in Compiler.cs under the source code.

In the present design of the traits, the idea is to generate one C# *interface* and one C# *companion class* per each Dafny trait. The C# *interface* includes the non-static members of the trait and the C# *companion class* includes the static members of the trait. In the compilation phase, ghost functions, methods and fields are not considered (recall that ghost members are just for verification and are not included in the built output). In

order to see the outcome of the compilation of a Dafny program with trait into C# consider the trait declaration in Dafny in Listing 28 which includes static and non-static members:

```
trait J
{
	var x: T;

	// Here are three functions that are relevant to the compiler:
	function method F(y: Y): T
	function method G(y: Y): T { E }
	static function method K(y: Y): T { E }

	// Here are three methods that are relevant to the compiler:
	method M(y: Y)
	method N(y: Y) { }
	static method Q(y: Y) { }
}
```

Listing 28: A trait declaration in Dafny

By compiling the program in Listing 28, the result in C# will be the following interface and companion class.

```
public interface J
{
    // An interface in C# cannot have fields, so we instead use a
    // property getter and setter
    T x { get; set; }
    // The two instance functions are declared as members of the C# interface
    T F(Y y);
    T G(Y y);
    // The two instance methods are declared as members of the C# interface
    void M(Y y);
    void N(Y y);
}

public class _Companion_J
{
    // The static function in the Dafny interface is declared as a
    // static member in this companion class
    public static T K(Y y) { return E; }
    // The static method in the Dafny interface is declared as a
    // static member in this companion class
    public static void Q(Y y) { }
}
```

Listing 29: C# interface and companion class generated by the Dafny compiler

Consider now the design in Listing 29. If there is a call to static member Q of the trait J, then the compiler emits calls to the associated member in the companion class, *_Companion_J.Q*.

In this point, some explanations about the Dafny compiler are needed.

In the compilation phase of a Dafny program, the type of all the registered declarations (from the resolver phase) is checked. Then, for every declaration, appropriate C# program fragment is generated and added to the *wr* object, which is a field in the compiler class and is responsible for saving a long string which actually represents a C# program. Then, as every declaration also has some members, those members must be iterated to generate appropriate C# code fragments also for those members. For example, a ClassDecl has related method and function declarations in it from which proper C# code fragments are generated for the ClassDecl and its members. The *wr* object then is sent to the C# compiler for building the output.

In order to compile traits in a Dafny program, the whole work is to add one more *if* branch to the current *if* branches to check if the current declaration is a TraitDecl. In case it is, the proper C# code fragment is added to *wr* object. This is slightly different from that of ClassDecl which already exists in the previous implementation of the Dafny. Listing 30 shows the implementations.

```
public void Compile(Program program) {
       …
       else if (d is TraitDecl)
       {
           //writing the trait
           var trait = (TraitDecl)d;
           Indent(indent);
           wr.Write("public interface @{0}", trait.CompileName);
           wr.WriteLine(" {");
           CompileClassMembers(trait, indent + IndentAmount);
           Indent(indent); wr.WriteLine("}");

           //writing the _Companion class
           List<MemberDecl> members = new List<MemberDecl>();
           foreach (MemberDecl mem in trait.Members)
           {
               if (mem.IsStatic && !mem.IsGhost)
               {
```

```
                    if (mem is Function)
                    {
                        if (((Function)mem).Body != null)
                            members.Add(mem);
                    }
                    if (mem is Method)
                    {
                        if (((Method)mem).Body != null)
                            members.Add(mem);
                    }
                }
            }
            var cl = new ClassDecl(d.tok, d.Name, d.Module,
               d.TypeArgs, members, d.Attributes, null);
            Indent(indent);
            wr.Write("public class @_Companion_{0}", cl.CompileName);
            wr.WriteLine(" {");
            CompileClassMembers(cl, indent + IndentAmount);
            Indent(indent); wr.WriteLine("}");
        }
        else if (d is ClassDecl) {
        …
}
```

Listing 30: Compiling trait declarations

In Listing 30, call to *CompileClassMembers* method is responsible for generating the C# code fragments for individual members of a trait. Please refer to CodePlex.com for the full detailed implementations.

3.2.4. Changes to the Verifier

Apart from the parser, the scanner and the compiler that are typical components of many other programming languages, Dafny has also a program verifier. As there are off-the-shelf program verifiers and intermediate languages that talk to the program verifiers, Dafny's verifier is actually a translator from Dafny to Boogie. Boogie in this case is the intermediate language. After translating the input program, which is a Boogie program, it is sent to Boogie to produce Verification Conditions (VCs) which are input to the formula solver.

Introducing traits affects the translator. If a class C extends a trait T, then the following changes are needed:

- Adding a predicate as implements$T to the target generated Boogie program.
- Adding proper axioms for the implements$T predicate to tell the verifier the fact that T is the supper class of C.
- If in a class C a member (function or method) *m* overrides *m'* in its base trait, then adding a new Boogie procedure *OverrideSpecificationCheck* to the target Boogie program to check if the specification of the overriding member is indeed equal or strengthened of its base. The "strengthened specifications" will be discussed later when elaborating further these changes.
- Adding a connection between the trait members and the related class members. The connection is created in the target Boogie program using an axiom. More explanations come below.

Adding the implements$T Predicate

In the translation process, the Dafny types are translated into more coarse grained Boogie types [Leino, 2009]. The reason is that, for example, the Boogie does not support classes or object orientation. In Boogie, all reference types are declared using type *ref* [Leino, 2009]. In order to translate the classes, Dafny generates some type predicates also on particular class variables to further distinguish those variables [Leino, 2009]. It means that predicates describe some properties of an object. For instance, if in a Dafny program, a variable *x* is declared of class type *C*, then the corresponding declaration in Boogie is a variable *x* of type *ref* along with a type predicate on *x* [Leino, 2009] as shown in Listing 31.

```
x == null || dtype(x) == Tclass._module.C()
```

Listing 31: A predicate in Boogie to denote x is of type C

The predicate in Listing 31 says that either x is null or it is a dynamic type of the return value of *Tclass._module.C()*, where *Tclass._module.C()* is a function declared in the Boogie program [Leino, 2009].

In the new translation, for every trait T, the translation into Boogie generates a predicate implements$T as shown in Listing 32.

```
function implements$T(Ty) : bool;
```

Listing 32: implements$T predicate

In order to add the mentioned predicate to the Boogie program during the translation, the change should be done in the method *AddClassMembers* under the Translator class. That method is responsible for generating Boogie program snippets for every member of a Dafny class. Listing 33 shows how to add the predicate. The full code is available in CodePlex.com in the file Translator.cs.

```
//this adds: function implements$T(Ty): bool;
if (c is TraitDecl)
{
    var arg_ref = new Bpl.Formal(c.tok, new Bpl.TypedIdent(c.tok,
       Bpl.TypedIdent.NoName, predef.Ty), true);
    var res = new Bpl.Formal(c.tok, new Bpl.TypedIdent(c.tok,
       Bpl.TypedIdent.NoName, Bpl.Type.Bool), false);
    var implement_intr = new Bpl.Function(c.tok, "implements$" +
       c.Name, new List<Variable> { arg_ref }, res);
    sink.TopLevelDeclarations.Add(implement_intr);
}
```

Listing 33: Generating implements$T predicate in the Boogie program

In Listing 33, *sink* is an object which is used to register all Boogie declarations which are the result of translating a Dafny program. Then, for the traits, instead of generating:

```
dtype(x) == Tclass._module.C(),
```

the following is generated:

```
implements$T(dtype(x))
```

So, after the above change Listing 31 will be:

```
x == null || implements$T(dtype(x))
```

In order to see the translated Boogie program using Dafny and the generated Boogie predicate above, there is a switch */print* which can be used when running Dafny.exe with an input Dafny program. For example, the following command verifies and compiles a Dafny program and writes the generated intermediate Boogie program into a file as *d.bpl*:

```
Dafny.exe  d.dfy  /print:d.bpl
```

A Boogie Axiom for a Class that Extends a Trait

When a class extends a trait in a program, and Dafny's resolver merges the members of the trait into the class members, the verifier should already know what to do. But the verifier needs to encode the information that a class extends a trait. For that information, one *axiom* must be added to the generated Boogie program. For a class C, the verifier already has generated the following Boogie constant (except that C will be fully qualified):

```
const unique class.C: ClassName;
```

To encode the fact that a class C implements a trait T, the verifier should also generate the following axiom:

```
axiom implements$J(class.C);
```

The mentioned axiom is generated using Listing 34 in the method *AddClassMembers* in the Translator class. The code checks if a class really extends a trait (c.trait != null) and then generates the axiom for that particular class.

```
//this adds: axiom implements$T(Ty)
else if (c is ClassDecl)
{
    if (c.Trait != null)
    {
        var args = new Bpl.Formal(c.tok, new Bpl.TypedIdent(c.tok,
              Bpl.TypedIdent.NoName, predef.ClassNameType), true);
        var ret_value = new Bpl.Formal(c.tok, new Bpl.TypedIdent(c.tok,
              Bpl.TypedIdent.NoName, Bpl.Type.Bool), false);
        var funCall = new Bpl.FunctionCall(new Bpl.Function(c.tok, "implements$" +
              c.TraitId.val, new List<Variable> { args }, ret_value));
        var expr = new Bpl.NAryExpr(c.tok, funCall, new List<Expr> { new
              Bpl.IdentifierExpr(c.tok, string.Format("class.{0}",
              c.FullSanitizedName), predef.ClassNameType) });
        var implements_axiom = new Bpl.Axiom(c.tok, expr);
        sink.TopLevelDeclarations.Add(implements_axiom);
    }
}
```

Listing 34: Generating implements$T(Ty) axiom in the Boogie program

OverrideSpecificationCheck: A Check for Overridden Functions and Methods

Based on the present design for overriding, it is possible for a class to override body-less functions and methods of the parent trait. Also, if a class wants to provide a body for a body-less function or method, then it repeats the signature of that method or function, including the in-parameters and out-parameters. In terms of the specifications, the class must provide the specifications all anew possibly stronger than its base.

The specification provided by a method or function in a class is strengthened or stronger than its base, if:

- The precondition is *more permissive*. As the caller is responsible for establishing the precondition, it may call the new member with a wider range of possible values. For example, in Listing 35, the values of the parameter *x* in the method *m* in the trait *T*, must be *x>= 0*. Now with a *more permissive* precondition that has been declared in the overriding member in the class *C*, *x* may be *-2000 <= x*.
- The postcondition is *more detailed*. As the callee is responsible for establishing the postcondition, the caller knows a more detailed description of the outcome of the called member. Actually, the postcondition in the class member includes the

postcondition in the trait member. For instance in Listing 35, $2*x < y$ declared in the class is a more detailed postcondition than $x<y$, that was declared in the trait.

- If w is the frame subset of the class and W is the frame subset of the trait, then $w <= W$. That makes sure that any caller that is allowed to read or modify w, is indeed allowed to read and modify W.
- If r is the termination measure of the class and R is the termination measure of the trait, then, $r <= R$. That ensures that any caller whose termination measure is strictly above r will also be strictly above R.

```
trait T
{
	function method F(x: int): int
	requires x < 100;

	method M(x: int) returns (y: int)
	requires 0 <= x;
	ensures x < y;
}
class C extends T
{
	function method F(x: int): int
	requires x < 100;
	{
		E
	}

	method M(x: int) returns (y: int)
	requires -2000 <= x; //a more permissive precondition than in the parent trait
	ensures 2*x < y; // a more detailed postcondition than in the parent trait
	{
		S;
	}
}
```

Listing 35: Extending a trait and overriding the specifications

In Listing 35, E is an expression whose result is appropriate for the result type of F, and S is a list of statements. The class declares its own function and methods with their own parameters and specifications. Then, the resolver checks that the function and the

method signatures in the class adhere to their associated function and method signatures in the trait, and the verifier checks that the provided specifications are really strengthened of those in the trait. In Listing 35, the provided precondition is a *more permissive* precondition than its parent trait. Also the provided postcondition is a *more detailed* one than of its parent trait.

This design increases the readability as it allows anyone to look at the class and immediately figure out what specifications govern the behavior of the members. Its downside is that the specifications must be repeated.

For checking the strengthening of the specifications, the verifier needs to be changed in two ways.

The verifier must provide an *override specification check* which ensures that any call to a member in the trait is also properly handled by its overridden member in the class. This semantic check is stronger than the check that the resolver does.

To describe how to generate the mentioned *override specification check*, consider the trait and class declarations in Listing 36.

```
trait T
{
       method M(x: X) returns (y: Y)
       requires Pre;
       modifies Frame;
       ensures Post;
       decreases Rank;
}
class C extends T
{
       method M(a: X) returns (b: Y)
       requires P;
       modifies W;
       ensures Q;
       decreases R;
}
```

Listing 36: A trait and extending it with a class

The *override specification check* for a method *M* is done using a new Boogie procedure, as in Listing 37.

```
procedure $OverrideCheck_$T_$C.M(a: X) returns (b: Y)
free requires ...; // standard stuff about $ModuleContextHeight and
$FunctionContextHeight
modifies $Heap, $Tick;
implementation $OverrideCheck_$T_$C.M(a: X) returns (b: Y)
{
	assume Pre';
	assert P; // this checks that Pre' implies P
	assert R <= Rank';
	assert W <= Frame'; // check subset
	change the heap at locations W;
	assume Q;
	assert Post'; // this checks that Q implies Post'
}
```

Listing 37: Boogie procedure for override specification check

In Listing 37, *Pre'*, *Post'*, *Rank'*, and *Frame'* are obtained from the specifications in the trait and then substituting their parameters' names with those used in the class. This means that *x* is replaced with *a* and *y* is replaced with *b* in Listing 36.

By assuming Pre' and then checking P, the verifier checks that any caller that has satisfied the precondition *Pre'* will also always satisfy *P*. By checking *R* <= *Rank'*, means that any caller whose termination metric is strictly above *Rank'* will be always also strictly above *R*. By checking that *W* is a subset of *Frame'*, the verifier makes sure that any caller who is allowed to modify frame subset at *Frame'* is indeed allowed to modify also *W*. By changing the heap at *W*, then assuming *Q* and then checking Post', the verifier checks that any caller that expects postcondition *Post'* is always happy also with *Q*.

For generating an override specification check for a function, the instruction is as above with two exceptions. The first one is that functions have *reads* clause instead of *modifies* clause. (Please refer to section related to the Dafny for more information.). The other exception is that, for functions, one more assume expression must be made between *assume Q* and *assert Post'* in Listing 37. That assumption is made to connect the trait

function and its overriding function in the class. If a trait T and class C have a function F with parameter *x*, then the mentioned *assume* expression will be as follows:

```
assume T.F(x) == C.F(x);
```

By applying the above changes, the Dafny program in Listing 35 is verified successfully. Please refer to the Dafny's source code at Codeplex.com for the implementations of the steps provided in this subsection. The code can be found in the translator class under *AddMethodOverrideCheckImpl* method (for generating the override specification check for an overridden method) and *AddFunctionOverrideCheckImpl* (for generating override specification check for an overridden function).

A Connection between the Trait and the Class Members

The last change on the verifier is to create a connection between the trait functions and the class functions. For example, consider the trait and class in Listing 38.

```
trait T
{
   function method F(x: int): int
}

class C extends T
{
   function method F(x: int): int
   {
      E
   }
}
```

Listing 38: Creating connection between the trait function and the class function

For the above functions, Dafny's verifier generates the following functions in the Boogie program:

```
function T.F($heap: HeapType, this: ref, x#0: int) : int;
function C.F($heap: HeapType, this: ref, x#0: int) : int;
```

The verifier also generates the following constant and function somewhere in the Boogie program (has mentioned in the previous subsections):

```
const unique class.C: ClassName;
function implements$T(Ty): bool;
```

But there is not any semantic connection between the mentioned functions. The connection which is required between the mentioned functions is as follows:

```
axiom (forall $heap: HeapType, this: ref, x#0: int ::
{ T.F($heap, this, x#0) }
this != null && dtype(this) == class.C
==>
T.F($heap, this, x#0) == C.F($heap, this, x#0));
```

The connection is created using *AddFunctionOverrideAxiom* method in the translator class.

3.2.5. Termination

Dafny verifies that recursion terminates. In case of defining a trait in one module and a class that extends that trait in another module, then infinite recursion could happen. Therefore, in this design such a declaration is not allowed. So, the resolver raises an error in case it finds a trait and a class that extends that class and they are in two modules. There is also a need to add a call-graph edge from T.F to C.F, where T and C are a trait and a class and F is a function or method, respectively. By adding such a call-graph we indicate that the behavior of T.F is like that of possibly calling C.F.

By making the above two changes, the verifier can do sound verification of termination in intra-module case while forbidding other cases at the same time (multi-module case).

The mentioned change has been done using the code snippet in Listing 39 in the *InheritTraitMembers* method in the resolver class.

```
//adding a call graph edge from the trait method to that of class
cl.Module.CallGraph.AddEdge(traitMethod, classMethod);
```

Listing 39: Adding call-graph edge

Additional Minor Changes

There are also some minor changes to the source language. One is on the *printer* and the other one is on the *syntax highlighter*. The first one, as its name suggests, is to change the printer class in order to be able to print a parsed Dafny program using command prompt switches. The second one is to make Visual Studio highlights the new keywords, *trait* and *extends*.

Changing the Printer

In order to print the parsed program in the console, there is a switch which can be used in the command line. In order to support *trait* keyword in the printed program, printer class must be altered slightly. It is simply done by changing the *PrintClass* method as shown in Listing 40.

```
public void PrintClass(ClassDecl c, int indent) {
  Contract.Requires(c != null);
  Indent(indent);
  PrintClassMethodHelper((c is TraitDecl) ? "trait" : "class", c.Attributes, c.Name,
       c.TypeArgs);
  if (c.TraitId != null) {
    wr.Write(" extends {0}", c.TraitId.val);
  }
…
}
```

Listing 40: Altering printer class

By applying the above change, by calling Dafny.exe using the following command, trait.dfy will be compiled and printed in the standard output:

Dafny.exe /dprint:- c:\Dafny\Test\dafny0\trait.dfy

Changing the Syntax Highlighters

After changing the parser, there is a need to add new keywords to the syntax highlighters accordingly, namely *trait* and *extends* keywords. These changes are made in *TokenTagger* class and in *Util* directory in the source folder.

3.2.6. Adding Associated Test Suits

Dafny is tested using regression testing. One or more test suit is added for every feature which is added to the source code. In addition, after fixing a bug, one more test suit is added to the test collection to make sure that the bug will not be reproduced in the next versions of the language. There are a collection of test suits at the moment in the source under *Test* folder.

New test cases have been created to test *trait* feature. All tests are executed to make sure that all the previous test cases pass successfully in addition to the new test cases. Test suits are executed by *lit* which is a utility written in Python. The utility, *lit*, lets executing all test suites in parallel. Dafny's test suits have been organized in different folders with specific number of tests in each one. That is useful for load balancing when the test suites are executed in parallel. Here is the command which is executed for running the test suites in parallel using the *lit* tool:

```
C:\dafny\Test\dafny0>lit . –v
```

That command will go through all available tests under dafny0 folder and run them in parallel. At the end, the utility will produce a summary of the executed tests and reports the number of failed tests and passed tests along with the name of the failed tests. For traits, there are 12 tests at the moment. The *lit* mechanism for testing is to take one source program, which in this case is a *.dfy* file, which must be decorated with one line of special command (which will be introduced later in this section) and one *.expect*

file. In the *.expect* file, there must be the expecting output which is the outcome of running the mentioned program with *dafny.exe*. For example, for testing the traits overriding feature, the test in Listing 41 has been provided.

```
// RUN: %dafny /compile:0 /print:"%t.print" /dprint:"%t.dprint" "%s" > "%t"
// RUN: %diff "%s.expect" "%t"
trait T1
{
   function method Plus (x:int, y:int) : int
     requires x>y;
   {
      x + y
   }

   function method bb(x:int):int
     requires x>10;
}
class C1 extends T1
{

}
```

Listing 41: TraitOverride.dfy

And the associated .expect file is as shown in Listing 42.

```
TraitOverride.dfy(15,6): Error: class: C1 does not implement trait member: bb
1 resolution/type errors detected in TraitOverride.dfy
```

Listing 42: TraitOverride.dfy.expect

By running *lit*, it will execute the source program and will compare the result with that of the associated *.expect* file, if they match, lit will report a pass otherwise it will report a failure.

For every single step of the implementation, one test is added and all tests are executing after every single change. These tests will guarantee the compatibility of the new changes with the existing implementations.

4 CONCLUSIONS

In the present work, support for inheritance was added in the form of traits to the Dafny programming language. This is a major advantage for programmers that are used to have inheritance in other programming languages. However, this work could be developed more, while traits could be utilized in a much richer way than in the current implementation. One potential future implementation could be to allow traits extend one or more than one traits or allowing a class to extend more than one trait. Implementing such features would enrich the ways traits are utilized. However, the implementations would not be rather different from the present one. One more potential extension would be to allow generic traits.

REFERENCES

[Barnett et al., 2005] Barnett, Mike, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# programming system: An overview." In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 49-69. Springer, 2005.

[Barnett et al., 2006] Barnett, Mike, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A modular reusable verifier for object-oriented programs." In *Formal methods for Components and Objects*, pp. 364-387. Springer, 2006.

[Barnett et al., 2011] Barnett, Mike, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. "Specification and verification: the Spec# experience." *Communications of the ACM* 54, 6 (2011): 81-91.

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

[Bracha, 1990] Bracha, Gilad, and William Cook. "Mixin-based inheritance." *SIGPLAN Notices*, 25, 10 (1990), 303-311.

[Bracha, 1992] Bracha, Gilad. "The programming language jigsaw: mixins, modularity and multiple inheritance." Ph.D. thesis, The University of Utah, 1992.

[Burdy et al., 2005] Burdy, Lilian, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. "An overview of JML tools and applications." *International Journal on Software Tools for Technology Transfer* 7, 3 (2005): 212-232.

[CodeDom at MSDN] http://msdn.microsoft.com/en-us/library/y2k85ax6(v=vs.110).aspx

[De Moura and Bjørner, 2008] De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." *Tools and Algorithms for the Construction and Analysis of Systems* (2008):

337-340.

[Dijkstra, 1976] Edsger W. Dijkstra. *A Discipline of Programming*. Vol. 4. Eng-lewood Cliffs: Prentice-Hall, 1976.

[Ducasse et al., 2006] Ducasse, Stéphane, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. "Traits: A mechanism for fine-grained reuse." *ACM Transactions on Programming Languages and Systems* 28, 2 (2006): 331-388.

[Hanspeter et al.] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The Compiler Generator Coco/R, Web pages at http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

[Herbert et al., 2012] Herbert, Luke, K. Rustan M. Leino, and Jose Quaresma. "Using Dafny, an automatic program verifier." In *Tools for Practical Software Verification*, pp. 156-181. Springer, 2012.

[Koenig and Leino, 2012] Koenig, Jason, and K. Rustan M. Leino. "Getting started with Dafny: A guide."*Software Safety and Security: Tools for Analysis and Verification* 33 (2012): 152-181.

[Leino, 2008] Leino, K. Rustan M. This is boogie 2. Manuscript KRML, 2008, 178: 131.

[Leino, 2009] Leino, K. Rustan M. "Specification and verification of object-oriented software." *Engineering Methods and Tools for Software Safety and Security* 22 (2009): 231-266.

[Leino, 2010] Leino, K. Rustan M. "Dafny: An automatic program verifier for functional correctness." In *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348-370. Springer, 2010.

[Leino et al., 2014] Leino, K. Rustan M., and Valentin Wüstholz. "The Dafny integrated development environment." arXiv preprint arXiv:1404.6602 (2014).

[Odersky et al., 2004] Odersky, Martin, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. "An overview of the Scala programming language." (2004). Technical Report IC/2004/64, École polytechnique fédérale de Lausanne, 2004.

[remix at CodePlex] http://remix.codeplex.com/

[Schärli et al., 2003] Schärli, Nathanael, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. "Traits: Composable units of behaviour." In *ECOOP 2003–Object-Oriented Programming* (2003): 248-274.

[Snyder, 1986] Snyder, Alan. "Encapsulation and inheritance in object-oriented programming languages." *ACM SIGPLAN Notices* 21, 11 (1986): 38-45.

[Taivalsaari, 1996] Taivalsaari, Antero. "On the notion of inheritance." *ACM Computing Surveys* 28, 3 (1996): 438-479.