Automatic Verification of Dafny Programs with Traits

Reza Ahmadi
University of Tampere
reza.ahmadi@uta.fi

K. Rustan M. Leino
Microsoft Research
leino@microsoft.com

Jyrki Nummenmaa
University of Tampere
jyrki@sis.uta.fi

ABSTRACT

This paper describes the design of *traits*, abstract superclasses, in the verification-aware programming language Dafny. Although there is no inheritance among classes in Dafny, the traits make it possible to describe behavior common to several classes and to write code that abstracts over the particular classes involved. The design incorporates behavioral specifications for a trait's methods and functions, just like for classes in Dafny. The design has been implemented in the Dafny tool.

Categories and Subject Descriptors

F.3.1 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Languages, Verification

Keywords

Program Verification, Dafny, Traits, Boogie

0. Introduction

Dafny is a verification-aware programming language with a relatively modest set of features to support classes with dynamically allocated instances [14]. This makes it possible to write and verify a large variety of programs that deal with pointers and mutate the state of objects in the heap. In the original design of the language, there was no inheritance among classes and no dynamic dispatch of methods. This ruled out the possibility of writing object-oriented programs where common functionality can be collected in some superclass and where code that uses the superclass can be polymorphic in the types of the instances that it sees at run time.

In this paper, we remove this limitation by introducing *traits* into the Dafny language. A trait is like a class in that it can declare member fields, functions, and methods, but a trait itself is not instantiable. Instead, a class can be declared to *extend* one or more traits, and the instances of the class can be used where such traits are expected. Object-oriented languages offer a slue of variations of traits and these go under many different names, including *abstract superclasses* (e.g., Eiffel [18]), *opaque types* (e.g., Modula-3 [19]), *interfaces* (e.g., Java [8] and C# [6]), *mix-ins* (e.g., Racket [7]), and, yes, traits (e.g., Scala [20]). Since there seems to be no consensus about what characteristics each of these names implies, we have somewhat arbitrarily chosen the name "trait".

A trait in Dafny can include mutable fields. These are inherited by the classes that extend the trait. Instance functions and methods declared in the trait can be given a body in either the trait itself or in classes that extend the trait. In either case, the trait, just like a class, associates a behavioral specification with each function and method and these must be respected by the classes that extend the trait. A trait can also declare static functions and methods, that is, functions and methods that do not take a receiver parameter. Since there is no dynamic dispatch to static functions and methods, a trait must give these a body.

Figure 0 shows a program snippet that illustrates some of the trait features. We will describe these features in more detail in the paper.

In this paper, we describe the design of traits in Dafny and give some examples. The design is implemented in the Dafny tool and can be tried out online at http://rise4fun.com/dafny. The implementation consists of not just the compiler, but also a static program verifier. We show in this paper that our traits can be specified in the style of *dynamic frames* [12], which is also the standard idiomatic way to specify classes in Dafny. At the end, we discuss future extensions of features we have left out in the current version and compare with related work. The contributions of this paper are:

- A specification-aware design of traits.
- A logical encoding of traits and overriding that is suitable for a verifier.
- Support for trait specifications in the style of dynamic frames.
- A language implementation of traits that includes both a compiler and a verifier.
- Pointing out a specification problem we have not solved regarding termination.

1. Trait Design

In Dafny, there are three kinds of members in classes: *fields* are mutable instance variables, *functions* are mathematical functions

```
trait J {
  var x: int
  function method Combine(y: int): int
    reads this
  method Reset()
    modifies this
    ensures 0 <= x
  method CombineAndReset(y: int) returns (z: int)
    modifies this
  {
    z := Combine(y);
    Reset();
  }
}
class C extends J {
  var w: int
  constructor ()
    modifies this
  { x, w := 0, 0; }
  function method Combine(y: int): int
    reads this
  { 2*x + w + y }
  method Reset()
    modifies this
    ensures x == 0 && w == old(x)
  { x, w := 0, x; }
}
```

Figure 0. The declaration of a trait with a field, a function, and two methods, as well as a class that extends the trait and supplies an implementation of the function and method.

that can take parameters and return a result, and *methods* are imperative code procedures that can take any number of in- and out-parameters. Unless declared `static`, functions and methods take an implicit receiver parameter, denoted `this`. A function may also depend on the program state, but must use a `reads` clause to declare the set of objects on whose state it may depend. A method may read and write the program state, but must use a `modifies` clause to declare the set of objects whose state it may update. The body of a function is an expression (enclosed in curly braces) and the body of a method is a statement sequence (also enclosed in curly braces).

There are some different declarations for special kinds of functions and methods. For example, a `predicate` is a function that returns a boolean and a `lemma` is a special kind of method (see the short tutorial notes [15] for some examples). For the sake of the present paper, these distinctions are not important, except for `constructor` declarations, which are special methods that can only be invoked when an object is allocated.

Like classes, traits can also declare fields, functions, and methods. Non-instance functions and methods are declared with the keyword `static`, just as for classes. However, since a trait itself is not instantiable, a trait may not declare any `constructor`. Figure 0 shows a trait `J` with a field `x`, a function `Combine`, and two methods, `Reset` and `CombineAndReset`. Instance functions and methods in a trait may omit the body, so that different classes get to implement these in their own ways. For example, in Figure 0, `Combine` and `Reset` do not have a body, whereas `CombineAndReset` does.

A class can be declared to `extend` one or more traits. By extending a trait, the class inherits all fields declared in the trait as well as all functions and methods with given bodies in the trait. For any body-less function or method in the trait, the class must also declare such a function or method. For example, in Figure 0, class `C` extends trait `J` and declares `Combine` and `Reset` with bodies.

```
trait Automobile {
  ghost var Repr: set⟨object⟩
  predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  function method Brand(): string
  var position: int
  method Drive()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures old(position) <= position
}
```

Figure 1. A trait specified in the general style of dynamic frames.

A salient feature of functions and methods in Dafny is that they can have behavioral specifications. For example, they can declare preconditions (keyword `requires`), which say under which conditions they are allowed to be invoked. As we alluded to above, functions can have `reads` clauses and methods can have `modifies` clauses. Both can also have postconditions (keyword `ensures`), which say something about what the implementation of the function or method guarantees. For example, trait `J` in Figure 0 declares that method `Reset` is allowed to modify the state of the receiver object (`this`) but must establish that field `x` is non-negative upon termination. The Dafny program verifier makes sure that the program respects the behavioral specifications.

When a class redeclares a function or method that was body-less in the trait, the class must also give a behavioral specification. In other words, the behavioral specification of the function or method is not inherited from the trait. We settled on this design because it means that a programmer only needs to look in one place in the source text to figure out what specification governs a particular call. The redeclaration will be subject to an *override check*, where the program verifier checks that the specification in the class is at least as strong as the one given in the trait. Note, for example, how class `C` gives method `Reset` a stronger postcondition.

2. Traits and Dynamic Frames

Specifying programs that deal with mutable, dynamic data structures can be difficult. A variety of specification idioms and logics have been developed (see [9] for a survey). For this purpose, Dafny uses *dynamic frames* [12]. From the language perspective, Dafny supports dynamic-frame specifications simply by supporting `reads` and `modifies` clauses that can list sets of objects. The basic idiom is for the program to maintain a set `Repr` of all the objects in the data structure and a predicate `Valid` that gives the invariant on that data structure. For more details of this idiom, see [10, 13].

In Figure 1, we show a trait `Automobile` that is equipped with idiomatic dynamic-frame specifications. The field `Repr` is used to hold the set of all objects that represent an automobile; it is declared with `ghost` to say that the field is for specifications only and will be erased by the compiler. Predicate `Valid()` holds whenever the automobile's representation is in a consistent state. The idea is that constructors have `Valid()` as a postcondition and that other methods have `Valid()` as a pre- and postcondition.

In our trait, the body of `Valid()` is elided so that classes that extend the trait can provide their own specific definitions. However,

```
method Main() {
  var auto: Automobile;
  auto := new Fiat(0);
  WorkIt(auto);
  auto := new Volvo();
  WorkIt(auto);
  auto := new Catacar();
  WorkIt(auto);
}
method WorkIt(auto: Automobile)
  requires auto ≠ null && auto.Valid()
  modifies auto.Repr
{
  auto.Drive();
  auto.Drive();
  assert old(auto.position) <= auto.position;
  print auto.Brand(), ": ", auto.position, "\n";
  auto.position := 18;
}
```

Figure 2. A main program that allocates instances of two specific automobile classes. Method `WorkIt` is able to operate on any object whose class extends the `Automobile` trait.

for our example, we have chosen to say that every class extending `Automobile` must define validity to imply `this in Repr`.

The specification of method `Drive()` has a postcondition that says the automobile must not go backwards. The rest of this method's specification contains the usual idiomatic parts of dynamic-frame specifications in Dafny. The postcondition that mentions `fresh` says that `Drive()` is allowed to add newly allocated objects to the automobile's representation set (see [10] for more motivation and details).

In Figure 2, we show a client method that allocates several automobile instances. More precisely, it allocates instances of classes that extend the trait `Automobile`. The constructors of these classes establish validity (Figures 3, 4, and 5) and therefore the precondition of method `WorkIt` is met. Note that the `auto` parameter of `WorkIt` has type `Automobile`, so `WorkIt` operates on any automobile.

Some details in method `WorkIt` are noteworthy. First, by the postcondition of `Drive()` in trait `Automobile`, the assertion in `WorkIt` after the two calls to `Drive()` can be verified to hold. Second, the method updates the field `position` directly. This is possible, since all `Automobile` objects have the field. Third, to update `auto.position`, the verifier will check that `auto` is covered by the `modifies` clause of `WorkIt`. This check comes down to verifying that `auto` is in `Repr`, which follows from the postcondition of `Valid()` that we included in Figure 1. Fourth, after the update to `auto.position`, it is not possible to prove, for any `Automobile`, that `Valid()` still holds. This is because `Valid()`, according to its `reads` clause, is allowed to depend on the value of the `position` field, and some class that extends `Automobile` may never be valid when `position == 18`. Hence, if `WorkIt` had a postcondition `auto.Valid()`, then the verifier would complain (but note that the automobile does remain valid after calling `Drive()`, as can be witnessed by the second call to `Drive()`).

Finally, we show in Figures 3, 4, and 5 classes `Fiat`, `Volvo`, and `Catacar`, each of which extends trait `Automobile`. Each class provides its own validity condition (that is, class invariant); for example, class `Fiat` places an upper bound on `position` and class `Volvo` says its `position` is a multiple of `10`. Note how the specifications

```
class Fiat extends Automobile {
  predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  {
    this in Repr && null !in Repr &&
    position <= 100
  }
  constructor (pos: int)
    requires pos <= 100
    modifies this
    ensures Valid() && fresh(Repr - {this})
    ensures position == pos
  {
    position, Repr := pos, {this};
  }
  function method Brand(): string {
    "Fiat"
  }
  method Drive()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures old(position) <= position
  {
    position := if position < 97
                then position + 3 else 100;
  }
}
```

Figure 3. A class that extends trait `Automobile`. The constructor allows the initial position to be set, and the class invariant says the position will never exceed `100`.

of the methods overridden in the trait are repeated or, in the case of `Volvo.Drive()`, strengthened. Lastly, note a rather subtle point in the implementation of `Fiat.Drive()`. If `position` is initially not less than `97`, then the method sets `position` to `100`. It may seem from this that the method may fail to live up to the postcondition that says `position` is not decreased. However, because of the method precondition `Valid()` and the definition of `Valid()` in class `Fiat`, the verifier is able to see that setting `position` to `100` does indeed satisfy the postcondition.

3. Logical Encoding

Our logical encoding for methods is the usual one. A method implementation is checked to satisfy the specification given to the method. At a call site, the static type of the receiver is used to retrieve the specification governing the call. For example, with reference to Figure 0, a call to `e.Reset()` uses the postcondition `0 <= x` if the static type of expression `e` is `J` and uses the more specific postcondition `x == 0 && w == old(x)` if the static type of `e` is `C`. At run time, a call `e.Reset()` dynamically dispatches to the implementation in the class corresponding to the allocated type of `e`. Since the override check verifies that the specification in the class is at least as strong as the one in the trait, soundness follows from the Liskov-Leavens Substitution Principle [17].

For functions, Dafny generates axioms [13]. For an instance function `F(x: X)` in a class `C`, the logical encoding uses a function with the receiver argument explicated. If the body of `F` is some

```
class Volvo extends Automobile {
  var odometer: Odometer
  predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  {
    this in Repr && null !in Repr &&
    odometer in Repr &&
    position % 10 == 0 &&
    odometer.value == position
  }
  constructor ()
    modifies this
    ensures Valid() && fresh(Repr - {this})
  {
    position, Repr := 0, {this};
    odometer := new Odometer();
    Repr := Repr + {odometer};
  }
  function method Brand(): string {
    "Volvo"
  }
  method Drive()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures old(position) < position
  {
    position := position + 10;
    odometer.Advance(10);
  }
}
class Odometer {
  var value: int
  constructor ()
    modifies this
    ensures value == 0
  {
    value := 0;
  }
  method Advance(d: int)
    requires 0 <= d
    modifies this
    ensures value == old(value) + d
  {
    value := value + d;
  }
}
```

Figure 4. Another class that extends trait Automobile. This automobile makes use of a simple auxiliary Odometer object whose state is updated by the Drive method. The postcondition of the Drive method promises a definite move forward. The class invariant says the odometer reading is kept in synch with the automobile's position, which is always a multiple of 10.

```
class Catacar extends Automobile {
  var f: Fiat
  var v: Volvo
  predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  {
    this in Repr && null !in Repr &&
    f in Repr && this !in f.Repr &&
      f.Repr <= Repr && f.Valid() &&
    v in Repr && this !in v.Repr &&
      v.Repr <= Repr && v.Valid() &&
    f.Repr !! v.Repr &&
    position == f.position + v.position
  }
  constructor ()
    modifies this
    ensures Valid() && fresh(Repr - {this})
  {
    Repr := {this};
    f := new Fiat(0);  Repr := Repr + f.Repr;
    v := new Volvo();  Repr := Repr + v.Repr;
    position := v.position;
  }
  function method Brand(): string {
    "Catacar"
  }
  method Drive()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures old(position) <= position
  {
    f := new Fiat(f.position);
    f.Drive();  v.Drive();
    Repr := Repr + v.Repr + f.Repr;
    position := f.position + v.position;
  }
}
```

Figure 5. A more advanced extension of the Automobile trait, illustrating how aggregate objects are composed in the dynamic-frame style of specifications. Using Dafny's sets-are-disjoint operator, !!, the class invariant says that the representations of the two component objects are separated. In this contrived example, the Drive method abandons the old Fiat, allocates a new one at the same position, and drives it.

expression E, then the axiom has the basic shape:

$$\begin{aligned} &\forall o\colon ref,\ x\colon X \bullet \\ &\quad o \neq null \wedge dtype(o) = C \\ &\quad \Longrightarrow F(o,x) = E \end{aligned} \tag{0}$$

where $dtype$ is a logical function that returns the allocated type (that is, the dynamic type) of object o and C is a term denoting the class C.

If function F is declared in a trait but given a body in a class, then we produce the same axiom. However, if the body is given in the trait, then there is no single value for $dtype(o)$ that can be used in the antecedent of the axiom. Instead, for every trait J, we introduce into the logical encoding a prediate $extends\_J$ and represent the axiom as:

$$\begin{aligned} &\forall o\colon ref,\ x\colon X \bullet \\ &\quad o \neq null \wedge extends\_J(dtype(o)) \\ &\quad \Longrightarrow F(o,x) = E \end{aligned} \tag{1}$$

Furthermore, for every class C that extends J, we generate an axiom:

$$extends\_J(C) \tag{2}$$

The Dafny verifier, and thus our logical encoding, uses Boogie [1] and proof obligations are discharged using an SMT solver.

4. Compilation

The standard Dafny compiler compiles to .NET bytecode (MSIL). Internally, the compilation is done by constructing a C# program and then calling on the C# compiler to generate MSIL. Conveniently, C# *interface* declarations are close to Dafny traits. But not quite. In this section, we describe the differences.

Unlike a Dafny trait, a C# interface cannot contain field declarations. Instead, we compile fields in traits into setter/getter methods in the C# interface. In each class that extends the trait, we introduce a field and arrange to override the setter/getter methods to use this field. C# makes this convenient for us through its *properties*, which are setter/getter method pairs with a field-like syntax for client code.

Despite the fact that MSIL allows some amount of code in interfaces, C# does not. Therefore, for instance functions and methods with a body in a Dafny trait, we omit the body when generating a method into the C# interface and we include a copy of the compiled code in every class that extends the trait. To compile static functions and methods in a Dafny trait J, we generate not just an interface J in the C# code, but also a class _Companion_J. The static members and then compiled into the companion class and calls to these static members are compiled into calls to these methods in the companion class.

5. Future Work

We have omitted some useful features in our current implementation of traits in Dafny. In particular, traits and the classes that implement them are currently not allowed to take type parameters, a feature we intend to add soon. Also, traits currently are not allowed to extend other traits. To add this feature, the language design will face the *diamond problem*, where a class can be an extension of a trait in more than one way. We are optimistic that verification will not add more issues than compilation does for this problem.

Dafny is currently undergoing a redesign of its type system to provide a more uniform treatment of subset types (like what the built-in nat is to the built-in int) and superset types (like what traits are to classes and what the built-in type object is to all classes). The hope is that this redesign will also provide a generalized approach to type inference, including the type inference that is done for traits.

There is an issue in our design that we have not discussed so far, namely the issue of termination. Dafny verifies that recursion and loops terminate, except loops and methods (not functions) that are specially marked as being possibly divergent. Within each module, Dafny constructs a call graph, and any calls within the same strongly connected component of that call graph are checked to decrease some programmer-supplied rank function [14]. To continue this approach for traits, we add a call-graph edge from body-less functions and methods in a trait to the corresponding functions and methods in classes that implement the trait. These edges represent the dynamic dispatch that takes place.

The import relation between modules in Dafny is acyclic, so calls into other modules have in the past had the property that they cannot be part of any recursive cycle. Traits change this. Consider the desirable case where a trait J containing a method or function M is declared in a library module. Consider, further, a method or function P(j: J) in that library module. If a client module declares a class C that extends J, then a call to P(c), passing an instance c of class C, could form another recursive cycle (in particular, if P calls j.M).

One modular solution (that is, a solution that allows verification module by module) to this specification and verification problem is to require programmer-supplied rank functions across module boundaries. This can be achieved in an approach that uses multisets of methods to represent ordinal numbers [11]. We would like to explore how such an approach can be incorporated into Dafny with a minimal impact on specification overhead. In the meantime, in our current design, we take the simple but draconian measure of disallowing a class to extend a trait that is declared in a different module. (A tiny step better would be to allow the class to extend the trait if the trait has no functions and all methods are marked as being possibly divergent.)

6. Related Work

A salient feature of our traits is that functions and methods have behavioral specifications, which the Dafny verifier checks. Object-oriented languages with trait-like features that also support behavioral specifications include Eiffel, Java (with some form of specifications), Spec#, and Racket. Racket provides only dynamic checking of specifications, whereas the others provide both dynamic and static checking and can verify adapted versions of the example programs in this paper.

In Spec#, the specifications of methods in an interface or class C are inherited by classes that extend C [0]. Spec# has a specification methodology that deals with subclasses and method overrides (see [16]). The overall specification language in Spec# is otherwise rather limited, including the limited support for user-defined functions.

Eiffel supports multiple-inheritance classes, which are verified by AutoProof [25]. To specify and reason about inter-object behavior, AutoProof uses the *semantic collaboration* methodology [22], which is more flexible than the Spec# methodology.

Java interfaces and classes can be specified using the Java Modeling Language (JML) [3] and statically checked using tools like ESC/Java and OpenJML [4]. The design of these tools trades sound verification for ease of use, in particular with regard to framing (that is, modifies clauses). The KeY system, based on dynamic logic, gives a full verifier for Java/JML [2].

Java interfaces and classes can also be specified using separation logic in the verifier VeriFast [24]. More precisely, VeriFast uses *ab-*

stract predicate families [21], which allow each subclass to provide its own specifications for overridden methods. Our logical encoding of traits was inspired by abstract predicate families, but since Dafny does not support subclasses, our encoding is but a special case of abstract predicate families.

The vision behind traits is to provide independent and composable units of behavior [23]. Damiani et al. give a proof system that focuses on the flexible composition of traits [5]. The proof system is modular in that each trait is verified separately from its uses, and it also allows some trait specifications to be added incrementally. However, from what we can tell, the proof system lacks features for framing that are necessary to deal with aggregate objects.

7. Conclusions

We have presented a design for traits in Dafny. The design allows specifications to be written in the style of dynamic frames, as we have demonstrated with an example. We have implemented our design, not just in the compiler but also in the program verifier. The most conspicuous issue we have left for future work involves figuring out how best to specify termination in the presence of dynamic dispatch, while keeping the benefits of modular verification.

Acknowledgments

We are grateful to the expert reviewers for their careful readings and good suggestions, and to Nadia Polikarpova for help with examples in AutoProof.

References

[0] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNAI*, pages 49–69. Springer, 2005.

[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. In *FMCO 2005*, volume 4111 of *LNAI*, pages 364–387. Springer, September 2006.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334. Springer, 2007.

[3] Lilian Burdy, 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): 212–232, June 2005.

[4] David R. Cok. Improved usability and performance of SMT solvers for debugging specifications. *Software Tools for Technology Transfer (STTT)*, 12 (6): 467–481, November 2010.

[5] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. Verifying traits: an incremental proof system for fine-grained reuse. *Formal Aspects of Computing*, 26 (4): 761–793, July 2014.

[6] *C# Language Specification*. ECMA International, June 2006.

[7] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *APLAS 2006*, volume 4279 of *LNAI*, pages 270–289. Springer, November 2006.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44 (3), June 2012. Article 16.

[10] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. Using Dafny, an automatic program verifier. In *LASER Summer School 2011*, volume 7682 of *LNAI*, pages 156–181. Springer, 2012.

[11] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification. In *ECOOP 2015*, 2015.

[12] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006*, volume 4085 of *LNAI*, pages 268–283. Springer, August 2006.

[13] K. Rustan M. Leino. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.

[14] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNAI*, pages 348–370. Springer, April 2010.

[15] K. Rustan M. Leino. Developing verified programs with Dafny. In *ICSE '13*, pages 1488–1490. IEEE/ACM, May 2013.

[16] K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *LASER Summer School 2007/2008*, volume 6029 of *LNAI*, pages 91–139. Springer, 2010.

[17] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 (6), 1994.

[18] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[19] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, 1991.

[20] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, EPFL, 2006.

[21] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL 2008*, pages 75–86. ACM, January 2008.

[22] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *FM 2014*, volume 8442 of *LNAI*, pages 514–530. Springer, May 2014.

[23] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003*, volume 2743 of *LNAI*, pages 248–274. Springer, July 2003.

[24] Jan Smans, Bart Jacobs, and Frank Piessens. VeriFast for Java: A tutorial. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNAI*, pages 407–442. Springer, 2013.

[25] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *TACAS 2015*, volume 9035 of *LNAI*, pages 566–580. Springer, April 2015.