Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.

0 Introduction

Applications of program verification technology fall into a spectrum of assurance levels, at one extreme proving that the program lives up to its functional specification (*e.g.*, [8, 23, 28]), at the other extreme just finding some likely bugs (*e.g.*, [19, 24]). Traditionally, program verifiers at the high end of the spectrum have used interactive proof assistants, which require the user to have a high degree of prover expertise, invoking tactics or guiding the tool through its various symbolic manipulations. Because they limit which program properties they reason about, tools at the low end of the spectrum have been able to take advantage of satisfiability-modulo-theories (SMT) solvers, which offer some fixed set of automatic decision procedures [18, 5].

An SMT-based program verifier is automatic in that it requires no user interaction with the solver. This is not to say it is effortless, for it usually requires interaction at the level of the program being analyzed. Used analogously to type checkers, the automatic verifier takes a program with specifications (analogously, type annotations) and produces error messages about where the program may be violating a rule of the language (like an index bounds error) or a programmer-supplied specification (like a failure to establish a declared postcondition). The error messages speak about the program (*e.g.*, "`MyProgram.dfy(212,23): loop invariant might not hold on entry`") and can be computed continuously in the background of an integrated development environment. Compared with the use of an interactive proof assistant, the added automation and the interaction closer to the programmer's domain stand a chance of reducing the effort involved in using the verifier and reducing the amount of expertise needed to use it.

Recently, some functional-correctness verification has been performed using automatic program verifiers based on SMT solvers or other automatic decision procedures [45, 52, 14, 33]. This has been made possible by increased power and speed of state-of-the-art SMT solvers and by carefully crafted program verifiers that make use of the SMT solver. For example, the input language for programs and specifications affects how effective the verifier is. Moreover, SMT solvers obtain an important kind of user extensionality by supporting quantifiers, and these quantifiers are steered by matching triggers. The design of good triggers is a central ingredient in making effective use of SMT solvers (for various issues in using matching triggers, see [34]).

In this paper, I describe the language and verifier Dafny. The language is imperative, sequential, supports generic classes and dynamic allocation, and builds in specification constructs (as in Eiffel [42], JML [29], and Spec# [4]). The specifications include standard pre- and postconditions, framing constructs, and termination metrics. Devoid of convenient but restricting ownership constructs for structuring the heap, the specification style (based on *dynamic frames* [27]) is flexible, if sometimes verbose. To aid in specifications, the language includes user-defined mathematical functions and ghost variables. These features permit programs to be specified for *modular verification*, so that the separate verification of each part of the program implies the correctness of the whole program. Finally, in addition to class types, the language supports sets, sequences, and algebraic datatypes.

Dafny's basic features and statements are presented in Marktoberdorf lectures notes [33]. Those lecture notes give a detailed account of the logical encoding of Dafny, including the modeling of the heap and objects, methods and statements, and user-defined functions. The additional features described in this paper and present in the current implementation of the language and verifier include generic types, algebraic datatypes, ghost constructs, and termination metrics.

Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie 2 [2, 40, 32] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. Thus, the semantics of Dafny are defined in terms of Boogie (a technique applied by many automatic program verifiers, *e.g.*, [14,21]). The Boogie tool is then used to generate first-order verification conditions that are passed to a theorem prover, in particular to the SMT solver Z3 [17].

Dafny has been used to specify and verify some challenging algorithms. The specifications are understandable and verification times are usually low. To showcase the composition of Dafny's features, I describe the Schorr-Waite algorithm in Dafny. In fact, I include its entire program text (including its full functional correctness specifications), which as far as I know is a first in a conference paper.

Feeding this program through the Dafny verifier, the verification takes less than 5 seconds.

Dafny is available as open source at `boogie.codeplex.com`.

1 Dafny Language Features

Dafny is an imperative, class-based language. In this section, I describe some of its more advanced features and sketch how these are encoded in Boogie 2 by the Dafny verifier.

In principle, the language can be compiled to executable code, but the current implementation includes only a verifier, not a compiler. For a verified program, a compiler would not need to generate any run-time representation for specifications and ghost state, which are needed only for the verification itself.

1.0 Types

The types available in Dafny are booleans, mathematical integers, (possibly null) references to instances of user-defined generic classes, sets, sequences, and user-defined algebraic datatypes. There is no subtyping, except that all class types are subtypes of the built-in type **`object`**. Programs are type safe, that is, the static type of an expression accurately describes the run-time values to which the expression can evaluate. Generic type instantiations and types of local variables are inferred. Because of the references and dynamic allocation, Dafny can be used to write interesting pointer algorithms. Sets are especially useful when specifying framing (described below), and sequences and algebraic datatypes are especially useful when writing specifications for functional correctness (more about that below, too).

1.1 Pre- and Postconditions

Methods have standard declarations for preconditions (keyword **`requires`**) and postconditions (keyword **`ensures`**), like those in Eiffel, JML, and Spec#. For example, the following method declaration promises to set the output parameter `r` to the integer square root of the input parameter `n`, provided `n` is non-negative.

```
method ISqrt(n: int) returns (r: int)
  requires 0 ≤ n;
  ensures r*r ≤ n ∧ n < (r+1)*(r+1);
{ /* method body goes here... */ }
```

It is the caller's responsibility to establish the precondition and the implementation's responsibility to establish the postcondition. As usual, failure by either side to live up to its responsibility is reported by the verifier as an error.

1.2 Ghost State

A simple and useful feature of the language is that variables can be marked as **ghost**. This says that the variables are used in the verification of the program but are not needed at run time. Thus, a compiler can omit allocating space and generating code for the ghost variables. For this to work, values of ghost variables are not allowed to flow into non-ghost ("physical") variables, which is enforced by syntactic checks.

Like other variables, ghost variables are updated by assignment statements. For example, the following program snippet maintains the relation `g = 2*x`:

```
class C {
  var x: int; var y: int; ghost var g: int;
  method Update() modifies {this};
  { x := x + 1;  g := g + 2; }
}
```

(I will explain the **modifies** clause in Sec. 1.3.) Dafny follows the standard convention of object-oriented languages to let **this**`.x` be abbreviated as `x`, where **this** denotes the implicit receiver parameter of the method.

As far as the verifier is concerned, there is no difference between ghost variables and physical variables. In particular, their types and the way they undergo change are the same as for physical variables. At the cost of the explicit updates, this makes them much easier to deal with than model variables or pure methods (*e.g.*, [29]), whose values change as a consequence of other variables changing.

1.3 Modifications

An important part of a method specification is to say which pieces of the program state are allowed to be changed. This is called *framing* and is specified in Dafny by a **modifies** clause, like the one in the `Update` example above. For simplicity, all framing in Dafny is done at the object granularity, not the object-field granularity. So, a **modifies** clause indicates a set of *objects*, and that allows the method to modify *any field* of any of those objects.

For example, the `Update` method above is also allowed to modify **this**`.y`. If callers need to know that the method has no effect on `y`, the method specification can be strengthened by a postcondition **ensures** `y =` **old**`(y);`, where **old**`(E)`, which can be used in postconditions, stands for the expression `E` evaluated on entry to the method.

A method's **modifies** clause must account for *all* possible updates of methods. This may seem unwieldy, since the set of objects a method affects can be both large and dynamically determined. There are various solutions to this problem; for example, Spec# makes use of a programmer-specified ownership hierarchy among objects [3, 38], JML uses ownership and data groups [43, 31], and separation logic and permission-based verifiers infer the frame from the given precondition [44,49,37]. Inspired by *dynamic frames* [27], Dafny uses the crude and simple **modifies** clauses just described, which allows the frame to be specified by the value of a ghost variable. The standard idiom is to declare a set-valued ghost field, say `Repr`, to dynamically maintain `Repr` as the set of objects that are part of the receiver's representation, and to use `Repr` in **`modifies`** clauses (see [33]). For example:

```
class MyClass {
  ghost var Repr: set<object>;
  method SomeMethod() modifies Repr; { /* ... */ }
}
```

Recall, this **`modifies`** clause gives the method license to modify any field of any object in `Repr`. If **`this`** is a member of the set `Repr`, then the **`modifies`** clause also gives the method license to modify the field `Repr` itself.

In retrospect, I find that this language design—explicit, set-valued **`modifies`** clauses that specify modifications at the granularity of objects—has contributed greatly to the flexibility and simplicity of Dafny.

1.4 Functions

A class can declare mathematical functions. These are given a signature, which can include type parameters, a function specification, and a body. For illustration, consider the following prototypical function, declared in a class `C`:

```
function F(x: T) requires P; reads R; decreases D; { Body }
```

where `T` is some type, `P` is a boolean expression, `R` is a set-valued expression, `D` is a list of expressions, and `Body` is an expression (not a statement). The **`requires`** clause says in which states the function is allowed to be used; in other words, the function can be partial and its domain is defined by the **`requires`** clause. Just like Dafny checks for division-by-zero errors, it also checks that invocations of a function satisfy the function's precondition. The **`reads`** clause gives a frame for the function, saying which objects the function may depend on. Analogously to **`modifies`** clauses of methods, the **`reads`** clause describes a set of objects and the function is then allowed to depend on any field of any of those objects. Dafny enforces the **`reads`** clause in the function body. The **`decreases`** clause gives a termination metric (also known as a variant function or a ranking function), which specifies a well-founded order among recursive function invocations. Finally, `Body` defines the value of the function.

By default, function are "ghost" and can be used in specifications only. But by declaring the function with **`function method`**, the function definition is checked to be free of specification-only constructs and the function can then be used in compiled code.

The Dafny function is translated into a Boogie function with the name $C.F$ (in Boogie, dot is just another character that can be used in identifier names). In addition to the explicit Dafny parameters of the function, the Boogie function takes as parameters the heap and the receiver parameter. The prototypical function above gives rise to a Boogie axiom of the form:

$$\begin{array}{l}(\forall \mathcal{H}\colon \mathit{HeapType},\ \mathit{this}\colon [\![\mathsf{C}]\!],\ x\colon [\![\mathsf{T}]\!] \bullet \\ \qquad \mathit{GoodHeap}(\mathcal{H}) \land \mathit{this} \neq \mathit{null} \land [\![\mathsf{P}]\!] \ \Rightarrow\ C.F(\mathcal{H}, \mathit{this}, x) = [\![\mathsf{Body}]\!]\ )\end{array}$$

where *HeapType* denotes the type of the heap, $[\![\,\cdot\,]\!]$ denotes the translation function from Dafny types and expressions into Boogie types and expressions, and *GoodHeap* holds of well-formed heaps (see [33]). The Dafny function declaration is allowed to omit the body, in which case this definitional axiom is omitted and the function remains uninterpreted.

When generating axioms, one needs to be concerned about the logical consistency of those axioms. Unless the user-supplied body is restricted, the definitional axiom could easily be inconsistent, in particular if the function is defined (mutually) recursively. To guard against this, Dafny insists that any recursion be well-founded, which is enforced in two phases. First, Dafny builds a call graph from the syntactic declarations of functions. Then, for any function that may be (mutually) recursive, the language makes use of the termination metric supplied by the **decreases** clause. Such a metric is a lexicographic tuple whose components can be expressions of any type. Dafny enforces that any call between mutually recursive functions leads to a strictly smaller metric value. In doing the comparison, it first truncates the caller's tuple and callee's tuple to the longest commonly typed prefix. Integers are ordered as usual, **false** is ordered below **true**, **null** is ordered below all other references, sets are ordered by subset, sequences are ordered by their length, and algebraic datatypes are ordered by their rank (see Sec. 1.9). All of these are finite and naturally bounded from below, except integers, for which a lower bound of `0` is enforced. The lower bound is checked of the caller's **decreases** clause, but the check is performed at the time of a (mutually) recursive call, not on entry to the caller. This makes the specification of some **decreases** clauses more natural.

An omitted **decreases** clause defaults to the set of objects denoted by the **reads** clause.

There is one more detail about the encoding of the definitional axiom. In Boogie, all declared axioms are available when discharging proof obligations. Thus, if the axioms are inconsistent, then all proof obligations can be discharged trivially, even the proof obligations designed to ensure the consistency of the axioms! To avoid such circularities, Dafny adds an antecedent to each definitional axiom; this lets the axiom be activated selectively. Let the *height* of a function be its order in a topological sort of all functions according to the call graph. For example, mutually recursive functions have the same height. The antecedent added to the definitional axiom of a function `F` of height h is $h < \mathit{ContextHeight}$, where *ContextHeight* is an uninterpreted constant. To activate the definitional axioms of non-recursive calls, the consistency check of function `F` is given the assumption $\mathit{ContextHeight} = h$. Proof obligations related to methods get to assume what amounts to $\mathit{ContextHeight} = \infty$, thus activating all definitional axioms.

To further explain the Boogie encoding of Dafny's functions, it is necessary first to give more details of the encoding of the heap [33]. The Dafny verifier models the heap as a map from object references and field names to values. The type of a Dafny field-selection expression `o.f` depends on the type of the field `f`, which is modeled directly using Boogie's polymorphic type system. All object references in Dafny are modeled by a single Boogie type *Ref*. A field `f` of type `T` declared in a class `C` is modeled as a constant $C.f$ of type *Field* $[\![\mathtt{T}]\!]$, where *Field* is a unary type constructor. The type of the heap, *HeapType*, is a polymorphic map type $\langle\alpha\rangle[\mathit{Ref}, \mathit{Field}\ \alpha]\alpha$; in words, for any type α, given a *Ref* and a *Field* α, the map returns an α [40]. For example, if `f` is of type **`int`** and `o` has type `C`, then the Boogie encoding declares a constant $C.f$ of type *Field* **int** and, in a heap $\mathcal{H}$, `o.f` is modeled as $\mathcal{H}[o, C.f]$, which has the Boogie type **int**.

Back to the encoding of functions. The **`reads`** clause of the prototypical function above produces the following frame axiom:

$$\begin{array}{l}
(\forall\, \mathcal{H}, \mathcal{K} \colon \mathit{HeapType},\ \mathit{this} \colon [\![\mathtt{C}]\!],\ x \colon [\![\mathtt{T}]\!] \bullet \\
\quad \mathit{GoodHeap}(\mathcal{H}) \wedge \mathit{GoodHeap}(\mathcal{K}) \wedge \\
\quad (\forall\, \langle\alpha\rangle\ o \colon \mathit{Ref},\ f \colon \mathit{Field}\ \alpha \bullet\ o \neq \mathit{null} \wedge [\![\mathtt{o} \in \mathtt{R}]\!] \Rightarrow \mathcal{H}[o, f] = \mathcal{K}[o, f]\ ) \\
\quad\ \Rightarrow C.F(\mathcal{H}, \mathit{this}, x) = C.F(\mathcal{K}, \mathit{this}, x)\ )
\end{array}$$

where "$\forall\langle\alpha\rangle\ o \colon \mathit{Ref},\ f \colon \mathit{Field}\ \alpha$" quantifies over all types α, all references o, and all fields names f of type *Field* α. The frame axiom says that if two heaps $\mathcal{H}$ and $\mathcal{K}$ agree on the values of all fields of all non-null objects in `R`, then $C.F$ returns the same value in the two heaps.

At first, it may seem odd to have a frame axiom, since the function's definitional axiom is more precise, but the frame axiom serves several purposes. First, if `Body` is omitted, the frame axiom still gives a partial interpretation of the function. Second, the frame axiom opens the possibility of using scope rules where Dafny hides the exact definition, except in certain restricted scopes, for example when verifying the enclosing class. By emitting the definitional axiom only when verifying the program text in the restricted scopes, other scopes then only get to know what the function depends on, not its exact definition. Such scope rules are still under experimentation in the current version of Dafny (but see, *e.g.*, [30, 49]). Third, for certain recursively defined functions, the frame axiom can sometimes keep the underlying SMT solver away from matching loops.

Dafny allows the frame of a function to be given as **`reads`** `*;`, in which case the function is allowed to depend on anything and the frame axiom is not emitted.

The logical consistency of the frame axiom plus the definitional axiom is justified by reads check that are part of the function body's consistency check: each heap dereference is checked to be in the declared **`reads`** clause [33]. Meanwhile, the frame axiom by itself is consistent, so it is always activated.

1.5 Specifying Data-Structure Invariants

When specifying a program built as a layers of modules, it is important to have specifications that let one abstract over the details of each module. Several approaches exist, for example built around ownership systems [13, 12], explicit validity bits [3], separation logic [46], region logic [1], and permissions [9]. Dafny follows an approach inspired by dynamic frames [27], where the idea is to specify frames as dynamically changing sets and where the consistency of data structures (that is, *object invariants* [42,3]) are specified by validity functions [19,39,44]. The sets, functions, ghost variables, and **modifies** clauses of Dafny are all that is needed to provide idiomatic support for this approach.

Let me illustrate the idiom that encodes dynamic frames in Dafny with this program skeleton:

```
class C {
  ghost var Repr: set<object>;
  function Valid(): bool
    reads {this} ∪ Repr;
  { this ∈ Repr ∧ ... }
  method Init()
    modifies {this};
    ensures Valid() ∧ fresh(Repr - {this});
  { ... Repr := {this} ∪ ...; }
  method Update()
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
  { ... }
  ...
}
```

The ghost variable `Repr` is the dynamic frame of the object's representation. The Dafny program needs to explicitly update the variable `Repr` when the object's representation changes. Dafny does not build in any notion of an object invariant. Instead, the body of function `Valid()` is used to define what it means for an object to be in a consistent state.

Dafny does not have any special constructs to support object construction. Instead, one declares a method, named `Init` in the skeleton above, that performs the initialization. A client then typically allocates and initializes an object as follows:

```
var c := new C;  call c.Init();
```

Method `Init` says it may modify the fields of the object being initialized, which includes the ghost field `Repr`. Its postcondition says the method will return in a state where `Valid()` is **`true`**. Since `Init` is allowed to modify `Repr`, it declares a postcondition that says something about how it changes `Repr`. An expression **`fresh`**`(S)`, which is allowed in postconditions, says that all non-null objects in the set `S` have been allocated since the invocation of the method. The postcondition of `Init` says that all objects it adds to `Repr`, except **`this`** itself, have been allocated by `Init`. Thus, the typical client above can conclude that `c.Repr` is disjoint from any previous set of objects in the program.

The program skeleton also shows a typical mutating method, `Update`. It requires and maintains the object invariant, `Valid()`, and it only modifies objects in the object's representation. Since `Update` can modify the ghost variable `Repr`, the postcondition **`fresh`**(...) promises to add only newly allocated objects to `Repr`, which lets clients conclude that the object's representation does not bleed into previous object representations.

More about this idiom and some examples are found in the Marktoberdorf lecture notes [33].

Note that the specifications in the program skeleton above are just idiomatic. It is easy to deviate from this idiom. For example, to specify that the `Append` method of a `List` class will reuse the linked-list representation of the argument, one might use the following specification:

```
method Append(that: List)
  requires Valid() ∧ that.Valid();
  modifies Repr ∪ that.Repr;
  ensures Valid() ∧ fresh(Repr - old(Repr) - old(that.Repr));
```

Here, nothing is said about the final value of `that.Repr`; in particular, the caller cannot assume **`this`** and `that` to have disjoint representations after the call. Moreover, the value of `that.Valid()` is also under-specified on return, so the caller cannot assume `that` to still be in a consistent state. One may need to strengthen the precondition above with `Repr ∩ that.Repr = {}`, which says that the representations of **`this`** and `that` are disjoint, or perhaps with `Repr ∩ that.Repr ⊆ {null}`, which says that they share at most the **`null`** reference.

The code in the skeleton shows only the specifications one needs to talk about the object structure. To also specify functional correctness, one typically adds more ghost variables (for example,

```
ghost var Contents: seq<T>;
```

for a linked list of `T` objects) and uses these variables in method pre- and postconditions.

1.6 Type Parameters

Classes, methods, functions, algebraic datatypes, and datatype constructors are generic, that is, they can take type parameters. Here is an example generic class, where `T` denotes a type parameter of the class:

```
class Pair<T> {
  var a: T; var b: T;
  method Flip() modifies {this}; ensures a = old(b) ∧ b = old(a);
  { var tmp := a; a := b; b := tmp; }
}
```

Uses of generic features require some type instantiation, which can often be inferred. For example, this code fragment allocates an integer pair and invokes the `Flip` method:

```
var p := new Pair<int>; p.b := 7; call p.Flip(); assert p.a = 7;
```

Expressions whose type is a type parameter can only be treated parametrically, that is, Dafny does not provide any type cast or type query operation for such expressions. For example, the body of the `Flip` method cannot use `a` and `b` as integers, but the client code above can, since, there, the type parameter has been instantiated with **`int`**.

Dafny types are translated into Boogie types, which generally are coarser. For example, **`bool`** and **`int`** translate to the same types in Boogie, and all class types translate into one user-defined Boogie type *Ref*, which is used to model all references. Procedures and functions in Boogie can also take type parameters, but the Dafny verifier does not make use of them for Dafny generics. The reason for this primarily has to do with the types of field values in the heap. As explained above in Sec. 1.4, each Dafny field gives rise to one Boogie constant that denotes the field name. This is important, because it allows generic code to be verified just once; for example, the implementation of the `Flip` method is verified without considering any specific instantiation of type parameter `T`. But in contexts where a use of a field like `a` or `b` is known to produce a specific type, like in the client code for `Flip` above, retrieving that field from the heap needs to produce the specific type. That goes beyond what the type constructor *Field* can do. So, rather than using type parameters in Boogie to deal with the generics in Dafny, the Dafny verifier introduces one Boogie type *Box* to stand for all values whose type is a type parameter. It then also introduces conversion functions from each type to *Box* and vice versa. The verifier is careful not to box an already boxed value, which can be ensured by looking at the static types of expressions.

In my personal experience with the Spec# program verifier, I have found the encoding of generic types to be an error prone enterprise. In retrospect, I think the reason has been that boxed entities in Spec# are encoded as references, which is what they look like in the .NET virtual machine. Admittedly, Dafny's generics are simpler, but my feeling is that the decision of encoding generic types using a separate Boogie type has led to a more straightforward encoding.

1.7 Sets

Dafny supports finite sets. A set of `T`-valued elements has type **`set`**`<T>`. Operations on sets are the usual ones, like membership, union, difference, and subset, but not complement and not cardinality. Sets are encoded as maps from values to booleans. The axiomatization defines all operations in terms of set membership; for example, there are no axioms that directly state the distribution of union over intersection. This axiomatization seems to work smoothly in practice—it is fast, simple, and gets the verification done.

In that spirit, set equality is translated into a Boogie function *SetEqual*, which is defined by equality in membership. Because Boogie does not promise extensionality of its maps [32], the reverse does not necessarily hold. For example, if `F` is a function on sets and `s` and `t` are sets, the Dafny verifier may not succeed in proving `F(s) = F(t)` even if it has enough information to establish that `s` and `t` have the same members. The verifier therefore includes the axiom

$$(\forall a, b{:}\, Set \bullet\ SetEqual(a,b) \Rightarrow a = b\ )$$

but because of the way quantifiers are handled in SMT solvers like Z3, this axiom is put into play only if the prover has a ground term $SetEqual(s,t)$. If that term is not available, the Dafny user may need to help the verifier along by supplying the statement **`assert`** `s = t;`, which both introduces the term $SetEqual(s,t)$ and adds it as a proof obligation.

A final remark about sets is that the Dafny encoding only ever uses sets of *boxes*. That is, the translation boxes values before adding them to sets. The reason for this is similar to the reason for introducing boxes for type parameters described in Sec. 1.6.

1.8 Sequences

Dafny also supports sequences, with operations like member selection, concatenation, and length. They are encoded analogously to sets, except that they do not use Boogie's built-in maps, but instead use a user-defined type *Seq* with a separate member-select function and a function for retrieving the length of the sequence. However, my experience with sequences has not been as smooth as with sets.

In particular, quantifying over sequence elements like in `(∀ i • ...s[i]...)`, but where the index into the sequence involves not just the quantified variable `i` but also some arithmetic, does not always lead to useful quantifier triggering. Although this problem has more to do with mixing triggers and interpreted symbols (like +), the problem can become noticeable when specifying properties of sequences. The workaround, once one has a hunch that this is the problem, is either to rewrite the quantifier or to supply an assertion that mentions an appropriate term to be triggered. For example, the list reversal program in the Marktoberdorf lecture notes [33] needs an assertion of the form:

```
assert list[0] = data;
```

1.9 Algebraic Datatypes

An algebraic datatype defines a set of structural values. For example, generic nonempty binaries trees with data stored at the leaves can be defined as follows:

```
datatype Tree<T> { Leaf(T); Branch(Tree<T>, Tree<T>); }
```

This declaration defines two constructors for `Tree` values, `Leaf` and `Branch`. A use of a constructor is written like `#Tree.Leaf(5)`, an expression whose type is `Tree<`**`int`**`>`.

The most useful feature of a datatype is provided by the **`match`** expression, which indicates one case per constructor of the datatype. For example,

```
function LeafCount<T>(d: Tree<T>): int  decreases d;
{
  match d
  case Leaf(t) ⇒ 1
  case Branch(u,v) ⇒ LeafCount(u) + LeafCount(v)
}
```

is a function that returns the number of leaves of a given tree.

All datatypes are modeled using a single user-defined Boogie type, *Datatype*, and constructors are modeled as Boogie functions. There are five properties of interest in the axiomatization of such functions:

0. each constructor is injective in each of its arguments,
1. different constructors produce different values,
2. every datatype value is produced from some constructor of its type,
3. datatype values are (partially) ordered, and
4. the ordering is well-founded.

Dafny emits Boogie axioms for three of these properties.

Properties (0) and (1) are axiomatized in the usual way, by giving the inverse functions and providing a category code, respectively. Property (2) is currently not encoded by the Dafny verifier, because it can give rise to enormously expensive disjunctions. Luckily, the property is usually not needed, because the only case-split facility that Dafny provides on datatypes is the **match** expression and Dafny insists, through a simple syntactic check, that all cases are covered (which means there is usually no need to prove in the logic that all cases are handled).

Property (3) is encoded using an integer function *rank* and axioms that postulate datatype arguments of a constructor to have a smaller rank than the value constructed. For example, Dafny emits the following axiom for `Branch`:

$$(\,\forall\, a0, a1{:}\, \mathit{Datatype} \bullet\;\; \mathit{rank}(a0) < \mathit{rank}(\mathit{Tree.Branch}(a0, a1))\,)$$

Property (4) is of interest when one wants to do induction on the structure of datatypes. It holds if the datatypes in a program can be stratified so that every datatype includes some constructor all of whose datatype arguments come from a lower stratum. Dafny enforces such a stratification, but it does not actually emit an axiom for this property (which otherwise would simply have postulated *rank* to return non-negative integers only), because the SMT solver never sees any proof obligation that requires it.

If the underlying SMT solver provides native support for algebraic datatypes (which Z3 actually does, as does CVC-3 [6]), then Dafny could tap into that support (which presumably provides all five properties) instead of rolling its own axioms. However, that would also require the intermediate verification language to support algebraic datatypes (which Boogie currently does not).

One final, important thing remains to be said about the encoding of datatypes, and it concerns the definitional axioms generated for Dafny functions. Recursive functions are delicate to define in an SMT solver, because of the possibility that axioms will be endlessly instantiated (a phenomenon known as a *matching loop* [18], see also [34]). The problem can be mitigated by specifying triggers that are structurally larger than the new terms produced by the instantiations. Following VeriFast [26], Dafny emits, for any function whose body is a **`match`** expression on one of the function's arguments, a series of definitional axioms, one corresponding to each **`case`**. The crucial point is that the trigger of each axiom discriminates according to the form of the function's argument used in the **`match`**.

For example, one of the definitional axioms for function `LeafCount` above is:

$$\begin{array}{l}(\forall\, u, v\colon \mathit{Datatype} \bullet \\ \qquad \mathit{LeafCount}(\mathit{Tree.Branch}(u,v)) \;=\; \mathit{LeafCount}(u) + \mathit{LeafCount}(v)\ )\end{array}$$

where the trigger is specified to be the left-hand side of the equality (for brevity, I omitted the $\mathcal{H}$ and *this* arguments to *LeafCount*). Note that the new *LeafCount* terms introduced by instantiations of this axiom would cause further instantiations only if the SMT solver has already equated u or v with some *Tree.Branch* term. In contrast, consider the following axiom, where I write $b0$ and $b1$ for the inverse functions of *Tree.Branch*:

$$(\forall\, d\colon \mathit{Datatype} \bullet\ \mathit{LeafCount}(d) \;=\; \mathit{LeafCount}(b0(d)) + \mathit{LeafCount}(b1(d))\ )$$

If triggered on the term *LeafCount*(d), this axiom is likely to lead to a matching loop, since each instantiation gives rise to new terms that also match the trigger.

1.10 Termination Metrics

As a final language-feature topic, let me say more about termination, and in particular about the termination of loops. Loops can be declared with loop invariants and a termination metric, the latter being supplied with a **`decreases`** clause that takes a lexicographic tuple, just as for functions. Dafny verifies that, each time the loop's back edge is taken (that is, each time control reaches the end of the body of the **`while`** statement and proceeds to the top of a new iteration where it will evaluate the loop guard), the ("post-iteration") metric value is strictly smaller than the ("pre-iteration") metric value at the top of the current iteration.

As I mentioned in Sec. 1.4, each Dafny type has an ordering, has finite values only, and, except for integers, is bounded from below. If the decrement of the metric involves decreasing an integer-valued component of the lexicographic tuple, then Dafny checks, at the time of the back edge, that the pre-iteration value of that component had been at least `0`. The complicated form of this rule (compared to, say, the simpler rule of enforcing as a loop invariant that every integer-valued component of the metric is non-negative) gives more freedom in choosing the termination metric. For example, the following loop verifies with the simple **`decreases`** clause given, despite the fact that `n` will be negative after the loop:

```
while (0 ≤ n) decreases n; { ... n := n - 1; }
```

To support applications where it is not desirable to insist on loop termination, Dafny lets a loop be declared with `decreases *;`, which skips the termination check for the loop.

Dafny also supports `decreases` clauses for methods, giving protection against infinite recursion. As for functions, Dafny proceeds in two phases, in the first phase building a call graph and in the second phase checking the `decreases` clauses of (mutually) recursive calls. Use of the two phases reduces the number of `decreases` clauses that programs need to contain.

2 Case Study: Schorr-Waite Algorithm

The famous Schorr-Waite algorithm marks all nodes reachable in a graph from a given root node [47]. What makes the algorithm attractive, and challenging for verification, is that it keeps track of most of the state of its depth-first search by reversing edges in the graph itself. This can be appropriate in the marking phase of a garbage collector, which is run at a time when space is low. The functional correctness of the algorithm has four parts: (C0) all reachable nodes are marked, (C1) only reachable nodes are marked, (C2) there is no net effect on the structure of the graph, and (C3) the algorithm terminates.

In this section, I present the entire Dafny program text for the Schorr-Waite algorithm. Using this 120-line program as input, Dafny (using Boogie 2 and Z3 version 2.4) verifies its correctness in less than 5 seconds. A large number of proofs have been constructed for the algorithm through both pen-and-paper proofs and mechanical verifications (see, *e.g.*, [10, 0, 41, 25, 11]). The previous shortest mechanical-verifier input for this algorithm appears to be 400 lines of Isabelle proof scripts by Mehta and Nipkow [41], whereas many other attempts have been far longer. To my knowledge, no previous Schorr-Waite proof has been carried out solely by an SMT solver, and never before has all necessary specifications and loop invariants been presented in one conference paper.

Here is a description of highlights of the program:

Class `Node` (lines 0–5 in the program below) represents the nodes in the graph, each with some arbitrary out-degree as represented by field `children`. The Schorr-Waite algorithm adds the `childrenVisited` field for bookkeeping, and the ghost field `pathFromRoot` is used only for the verification.

Datatype `Path` (lines 7–10) represents lists of `Node`'s and is used by function `Reachable` (line 13) to describe the list of intermediate nodes between `from` and `to`. Function `ReachableVia` (line 18) is defined recursively according to that list of intermediate nodes. Reachability predicates are notoriously difficult for first-order SMT solvers, but the trigger-aware encoding of the definitional axiom (explained in Sec. 1.9) makes it work honorably for this program.

The method itself is defined starting at line 28 and its specification is given on lines 29–41. The preconditions say that the method parameters describe a proper graph with marks and auxiliary fields cleared. Correctness property (C0) is specified by the postconditions on lines 36–37, (C1) on line 39, and (C2) on line 41. Noteworthy about this specification is that (C0) is specified as a closure property, using quantifiers, whereas (C1) uses the reachability predicate. Alternatively, one could have also specified (C0) with the reachability predicate, as the dual of (C1), but that would have led to a more complicated proof (using a loop invariant like Hubert and Marché's *I4c* in [25], which they describe as "the trickiest annotation"). Correctness property (C3) is implicit, since Dafny checks that loops and methods terminate.

The heart of the algorithm is implemented by a loop with 3 cases (lines 85–112), one for going deeper into the graph ("push"), one for considering the next child of the current node, and one for backtracking to the previous node on the stack ("pop"). The program maintains a ghost variable `stackNodes` that stores the visitation stack of the depth-first traversal. This ghost variable, which is updated explicitly (lines 48, 95, and 106), plays a vital part in the proof. It is used in most of the loop invariants. Lines 74–76 declare the relation between `stackNodes` and the Schorr-Waite reversed edges.

The loop invariant helps establish the postconditions as follows: Correctness property (C0) is maintained as a loop invariant (lines 51 and 61–62), except for those nodes that are on the stack. Ditto for (C2) (lines 63–65). Correctness property (C1) is also maintained as a loop invariant (line 72). It uses function `Reachable`, which is defined in terms of an existential quantifier. The prover needs help in establishing this existential quantifier when a node is marked (lines 46 and 109), so the program supplies a witness by using a local ghost variable `path` and an associated loop invariant (line 70). To maintain that invariant in the pop case, intermediate reachability paths are recorded in the ghost field `pathFromRoot`, see the loop invariant on line 71.

The termination metric for the loop is given as a lexicographic triple on line 83. The first component of the triple is a set, the second a sequence, and the third an integer. Dafny verifies that each iteration of the loop decreases this triple and that its integer component is bounded from below. Since sets are finite in Dafny, this establishes a well-founded order and thus implies that the loop terminates. In comparison, proving termination of the Schorr-Waite algorithm using Caduceus [21], a verifier equipped to use SMT solvers when possible, involved using a second-order formula even just to express the well-foundedness of the termination metric used, which necessitated the use of an interactive proof assistant to complete the proof [25].

```
 0  class Node {
 1    var children: seq<Node>;
 2    var marked: bool;
 3    var childrenVisited: int;
 4    ghost var pathFromRoot: Path;
 5  }
 6
 7  datatype Path {
 8    Empty;
 9    Extend(Path, Node);
10  }
11
12  class Main {
13    function Reachable(from: Node, to: Node, S: set<Node>): bool
```

```
    requires null ∉ S;
    reads S;
  { (∃ via: Path • ReachableVia(from, via, to, S)) }

  function ReachableVia(from: Node, via: Path, to: Node, S: set<Node>): bool
    requires null ∉ S;
    reads S;
    decreases via;
  {
    match via
    case Empty ⇒ from = to
    case Extend(prefix, n) ⇒ n ∈ S ∧ to ∈ n.children ∧ ReachableVia(from, prefix, n, S)
  }

  method SchorrWaite(root: Node, ghost S: set<Node>)
    requires root ∈ S;
    // S is closed under 'children':
    requires (∀ n • n ∈ S ⟹ n ≠ null ∧ (∀ ch • ch ∈ n.children ⟹ ch = null ∨ ch ∈ S));
    // graph starts with nothing marked and nothing being indicated as currently being visited:
    requires (∀ n • n ∈ S ⟹ ¬n.marked ∧ n.childrenVisited = 0);
    modifies S;
    // nodes reachable from 'root' are marked:
    ensures root.marked;
    ensures (∀ n • n ∈ S ∧ n.marked ⟹ (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
    // every marked node was reachable from 'root' in the pre-state:
    ensures (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
    // the structure of the graph has not changed:
    ensures (∀ n • n ∈ S ⟹ n.childrenVisited = old(n.childrenVisited) ∧
                             n.children = old(n.children));
  {
    var t := root;
    var p: Node := null;  // parent of t in original graph
    ghost var path := #Path.Empty;
    t.marked := true;
    t.pathFromRoot := path;
    ghost var stackNodes := [];
    ghost var unmarkedNodes := S - {t};
    while (true)
      invariant root.marked ∧ t ≠ null ∧ t ∈ S ∧ t ∉ stackNodes;
      invariant |stackNodes| = 0 ⟺ p = null;
      invariant 0 < |stackNodes| ⟹ p = stackNodes[|stackNodes|-1];
      // stackNodes has no duplicates:
      invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |stackNodes| ⟹ stackNodes[i] ≠ stackNodes[j]);
      invariant (∀ n • n ∈ stackNodes ⟹ n ∈ S);
      invariant (∀ n • n ∈ stackNodes ∨ n = t ⟹
        n.marked ∧ 0 ≤ n.childrenVisited ∧ n.childrenVisited ≤ |n.children| ∧
        (∀ j • 0 ≤ j ∧ j < n.childrenVisited ⟹ n.children[j] = null ∨ n.children[j].marked));
      invariant (∀ n • n ∈ stackNodes ⟹ n.childrenVisited < |n.children|);
      invariant (∀ n • n ∈ S ∧ n.marked ∧ n ∉ stackNodes ∧ n ≠ t ⟹
        (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
      invariant (∀ n • n ∈ S ∧ n ∉ stackNodes ∧ n ≠ t ⟹
        n.childrenVisited = old(n.childrenVisited));
      invariant (∀ n • n ∈ S ⟹ n ∈ stackNodes ∨ n.children = old(n.children));
      invariant (∀ n • n ∈ stackNodes ⟹
        |n.children| = old(|n.children|) ∧
        (∀ j • 0 ≤ j ∧ j < |n.children| ⟹
                j = n.childrenVisited ∨ n.children[j] = old(n.children[j])));
      // every marked node is reachable:
      invariant old(ReachableVia(root, path, t, S));
      invariant (∀ n, pth • n ∈ S ∧ n.marked ∧ pth = n.pathFromRoot ⟹
                             old(ReachableVia(root, pth, n, S)));
      invariant (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
      // the current values of m.children[m.childrenVisited] for m's on the stack:
      invariant 0 < |stackNodes| ⟹
                stackNodes[0].children[stackNodes[0].childrenVisited] = null;
      invariant (∀ k • 0 < k ∧ k < |stackNodes| ⟹
        stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k-1]);
      // the original values of m.children[m.childrenVisited] for m's on the stack:
      invariant (∀ k • 0 ≤ k ∧ k+1 < |stackNodes| ⟹
 79          old(stackNodes[k].children)[stackNodes[k].childrenVisited] = stackNodes[k+1]);
 80        invariant 0 < |stackNodes| ⟹
 81          old(stackNodes[|stackNodes|-1].children)[stackNodes[|stackNodes|-1].childrenVisited] =
             t;
 82        invariant (∀ n • n ∈ S ∧ ¬n.marked ⟹ n ∈ unmarkedNodes);
 83        decreases unmarkedNodes, stackNodes, |t.children| - t.childrenVisited;
 84      {
 85        if (t.childrenVisited = |t.children|) {
 86          // pop
 87          t.childrenVisited := 0;
 88          if (p = null) {
 89            return;
 90          }
 91          var oldP := p.children[p.childrenVisited];
 92          p.children := p.children[..p.childrenVisited] + [t] +
                           p.children[p.childrenVisited + 1..];
 93          t := p;
 94          p := oldP;
 95          stackNodes := stackNodes[..|stackNodes| - 1];
 96          t.childrenVisited := t.childrenVisited + 1;
 97          path := t.pathFromRoot;
 98        } else if (t.children[t.childrenVisited] = null ∨ t.children[t.childrenVisited].marked) {
 99          // just advance to next child
100          t.childrenVisited := t.childrenVisited + 1;
101        } else {
102          // push
103          var newT := t.children[t.childrenVisited];
104          t.children := t.children[..t.childrenVisited] + [p] +
                           t.children[t.childrenVisited + 1..];
105          p := t;
106          stackNodes := stackNodes + [t];
107          path := #Path.Extend(path, t);
108          t := newT;
109          t.marked := true;
110          t.pathFromRoot := path;
111          unmarkedNodes := unmarkedNodes - {t};
112        }
113      }
114    }
115  }
116
```

Here is a breakdown of the effort involved in constructing this Dafny program. I spent 5 hours one night, writing the algorithm (starting from a standard depth-first traversal with an explicit stack) and specifications (C0) and (C2), along with the loop invariants necessary for verification. The next day, I implemented **decreases** clauses for loops in Dafny, which let me write the lexicographic triple on line 83 to prove (C3). I then spent 2–3 days trying to define `ReachableVia` using a **seq**`<Node>`, after which I gave up and hand-coded algebraic datatypes into the Dafny-generated Boogie program. That seemed to lead to a proof. After a many-month hiatus from Dafny, I then added datatypes to Dafny and, within a few more hours, completed the full proof.

In conclusion, while the specification of the algorithm is clear and reading any one line of the loop invariants is likely to receive nods from a programmer, the 32 lines of quantifier-filled loop invariants can be a mouthful. The hardest thing in writing the program is deciphering the verifier's error messages so that one can figure out what loop invariant to add or change. That task is not yet for non-experts. Although I am pleased to have done the proof, I find the loop invariants to be complicated because they are so concrete, and think I would prefer a refinement approach like that used by Abrial [0].

3 Related Work

To a large extent, the language, specification constructs, and logical encoding of Dafny borrow from other languages and verifiers. The particular combination adds up to a flexible and powerful system. In this section, I give a more detailed comparison with some of the most closely related tools.

The Java Modeling Language (JML) [29] is a rich specification language for Java. It has many of the same specification features as Dafny. The biggest difference with Dafny lies on the tool side, where JML lacks an automatic verifier. The KeY tool [7] accepts JML specifications, but the tool uses an interactive verifier. ESC/Java [22, 15] uses JML specifications and uses an underlying SMT solver, but it performs *extended static checking* (that is, it intentionally misses some program errors in order to reduce the cost of using the tool [19]), not full verification.

In the specification language itself, JML supports behavioral subtyping for subclasses in Java. It has advanced support for ghosts, including model classes and model code. It does not build in algebraic datatypes, and termination metrics are more developed in Dafny. A larger difference is the way modular verification is done: JML uses object invariants and data groups [43, 31] whereas Dafny uses dynamic frames.

Spec# [4, 38] is an object-oriented language with specifications, defined as a superset of C# 2.0. It has an SMT-based automatic verifier [2] and provides modular verification by enforcing an ownership discipline among objects in the heap. For programs that fit this discipline, the necessary specifications are concise and natural. Spec# also has rich support for subclasses and immutable classes. However, Spec# lacks the mathematical specification constructs needed to carry out full functional correctness verification. For example, it has no ghost variables, no built-in sets or sequences, no algebraic datatypes, no termination metrics, and quantifiers are restricted to be ones that are executable. As a further comparison with Dafny, its support for verifying generic classes is not nearly as developed.

JML and Spec# use *pure methods* instead of mathematical functions. A pure method is a side-effect free method, written using statements in the programming language. The advantage of pure methods is that they leverage an existing language feature, and programs often contain query methods that return the value a mathematical function would have. However, pure methods are surprisingly complicated to get right. A major problem is that pure methods *do* have effects; for example, a pure method may allocate a hashtable that it uses during its computation. Another problem is that pure methods often are not deterministic, because they may return a newly allocated object (perhaps a non-interned string or an object representing a set of integers). These problems make it tricky to provide the programming logic with the desirable illusion that pure methods are functions (see, *e.g.*, [16, 36]). In contrast, the treatment of mathematical functions in Dafny, and the logic functions in Caduceus [20] from which they were first inspired, is simple (as is the treatment of functions in other logics that only provide functions, not statements and other programming constructs). Dafny's **`function method`** declaration achieves the advantage of using one language mechanism for both (restricted) specifications and code, but does so by letting the function be used in code rather than letting code be used as a function. I conclude with a slogan: pure methods are hard, functions are easy.

VeriCool 1 [50] has provided much inspiration for Dafny. It is also based on dynamic frames, but the prevailing style is to use VeriCool's pure methods (which are mostly like functions) instead of ghost variables. When such frames are defined recursively, they can bring about problems with matching loops in the SMT solver. Whereas Dafny does framing at the granularity of objects, VeriCool uses the more detailed object-field granularity. VeriCool has been extended to concurrency [48]. It does not support generic classes, algebraic datatypes, or termination metrics.

The idea in Dafny of using set-valued expressions for framing comes from the original work on dynamic frames by Kassios [27]. The difference lies in how the dynamic frames are represented, which impacts automation. The original dynamic frames are represented by specification variables, which are functions of other variables in the program. They are therefore more like Dafny's functions than Dafny's ghost variables. Because ghost variables are independent coordinates in the heap, they avoid the problems with recursively defined functions that, like in VeriCool 1, can be an issue for the SMT solver.

As for the notation, Kassios's preservation operator Ξ can be written using quantifiers and **`old`** in Dafny, the modification operator Δ corresponds to **`modifies`** clauses in Dafny, and Kassios's operator Λ for the *swinging pivots requirement* [39] corresponds to the idiom that uses **`fresh`** in Dafny (Sec. 1.5). Specification variables correspond to functions in Dafny, and the **frames** predicate corresponds to **`reads`** clauses in Dafny.

A recent trend seems to be to add more specification features to programming languages. For example, the Jahob verification system admits specifications written in higher-order logic [52]. This differs from interactive higher-order proof assistants in that the input mostly looks like a program, not a series of proof steps. Jahob also includes some features that can be used to write proofs, as does, to a lesser extent, Boogie [32]. VeriFast [26] integrates into C features for writing and proving lemmas.

Others are using SMT solvers for functional correctness verification. Régis-Gianas and Pottier used an SMT solver in their proof of Kaplar and Tarjan's algorithm for functional double-ended queues [45]. VCC [14] is being used to verify the Microsoft Hyper-V hypervisor. As part of that project, VCC has been used to prove the functional correctness (sans termination) of several challenging data structures.

From the other direction, various interactive proof assistants are using SMT solvers as part of their *grind* tactics.

4 Conclusions

In this paper, I have shown the design of Dafny, a language and verifier. Although it does not support all functional-correctness verification tasks—to do so is likely to require more data types and perhaps some higher-order features—it has already demonstrated its use in automatic functional-correctness verification. Rosemary Monahan and I have also used Dafny to complete the 8 verification benchmarks proposed by Weide *et al.* [51], except for one aspect of one benchmark, which requires a form of lambda closure [35].

Dafny had started as an experiment to encode dynamic frames, but it has grown to become more of a general-purpose specification language and verifier (where modular verification is achieved via dynamic frames). As part of future work on Dafny, I intend to build a compiler that generates executable code.

I expect that more full functional correctness verifications will be done by SMT-based verifiers in the future.

Acknowledgments. I wish to thank Daan Leijen, Peter Müller, and Wolfram Schulte for encouraging me to write this paper. Michał Moskal provided helpful comments on a draft of this paper. Jochen Hoenicke, Andreas Podelski, and the participants at the IFIP WG 2.3 meeting in Lachen were helpful in discussing termination specifications. Finally, I thank the reviewers for their helpful comments.

References

0. Abrial, J.-R.: Event based sequential program development: Application to constructing a pointer program. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 51–74. Springer, Heidelberg (2003)
1. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library, SMT-LIB (2008), `www.SMT-LIB.org`
6. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
7. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
8. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: Special issue on system verification. Journal of Automated Reasoning 5(4), 409–530 (1989)
9. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)

10. Broy, M., Pepper, P.: Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. ACM TOPLAS 4(3), 362–381 (1982)
11. Bubel, R.: The Schorr-Waite-algorithm. In: [7], ch. 15
12. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002, pp. 292–310. ACM, New York (2002)
13. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64. ACM, New York (1998)
14. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
15. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
16. Darvas, Á.P.: Reasoning About Data Abstraction in Contract Languages. PhD thesis, ETH Zurich, Diss. ETH No. 18622 (2009)
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
18. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
19. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (1998)
20. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
21. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
22. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM, New York (2002)
23. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 462–465. Springer, Heidelberg (1996)
24. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)
25. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: SEFM 2005, pp. 190–199. IEEE, Los Alamitos (2005)
26. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
27. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
28. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: SOSP 2009, pp. 207–220. ACM, New York (2009)

29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
30. Leino, K.R.M.: Toward Reliable Modular Programs. PhD thesis, California Institute of Technology, Technical Report Caltech-CS-TR-95-03 (1995)
31. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: OOPSLA 1998, pp. 144–153. ACM, New York (1998)
32. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178 (2008), `http://research.microsoft.com/~leino/papers.html`
33. Leino, K.R.M.: Specification and verification of object-oriented software. In: Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press, Amsterdam (2009); Summer School Marktoberdorf 2008 lecture notes
34. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009. ACM, NewYork (2009)
35. Leino, K.R.M., Monahan, R.: Dafny meets the Verification Benchmarks Challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010)
36. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008)
37. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, Springer, Heidelberg (2009)
38. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: Müller, P. (ed.) Advanced Lectures on Software Engineering. LNCS, vol. 6029, pp. 91–139. Springer, Heidelberg (2010)
39. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM TOPLAS 24(5), 491–553 (2002)
40. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
41. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation 199(1-2), 200–227 (2005); 19th International Conference on Automated Deduction (CADE-19)
42. Meyer, B.: Object-oriented Software Construction. Series in Computer Science.International. Prentice-Hall International, Englewood Cliffs (1988)
43. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62, 253–286 (2006)
44. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL 2005, pp. 247–258. ACM, New York (2005)
45. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008)
46. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE, Los Alamitos (2002)
47. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM 10(8), 501–506 (1967)

48. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An automatic verifier for a concurrent object-oriented language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008)
49. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
50. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: Automatic verifier for Java-like programs based on dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
51. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 84–98. Springer, Heidelberg (2008)
52. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: PLDI 2008, pp. 349–361. ACM, New York (2008)