

Dafny: An Automatic Program Verifier for Functional Correctness

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

Manuscript KRML 203, 10 February 2010.

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 verification of the Schorr-Waite algorithm in Dafny.

0 Introduction

The applications of program verification technology fall in a spectrum of assurance levels, at one extreme proving that the program lives up to its functional specification (*e.g.*, [8, 20, 25]), at the other extreme just finding some likely bugs (*e.g.*, [18, 21]). 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 what 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 [17, 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. The added automation and the interaction closer to the programmer’s domain stand a chance of reducing the effort involved in using the tool 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 [41, 47, 14, 30]. 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 for making effective use of SMT solvers (for various issues in using matching triggers, see [31]).

Dafny [30] is a language and verifier. The language is imperative, sequential, supports generic classes and dynamic allocation, and builds in specification constructs (as in Eiffel [38], JML [26], and Spec# [4], for example). The specifications include standard pre- and postconditions, framing constructs, and termination metrics for loops. Devoid of convenient but restricting ownership constructs for structuring the heap, the specification style (based on *dynamic frames* [24]) is flexible, if sometimes verbose. To aid in

specifications, the language includes mathematical functions and ghost variables. Finally, in addition to classes, the language supports sets, sequences, and algebraic datatypes.

Dafny has been used to specify and verify some challenging algorithms. The specifications are understandable and verification times are usually low. In this paper, I describe the language features in more detail, give examples of their use, and sketch how, via the intermediate verification language Boogie 2 [2, 36, 29], they are encoded for the SMT solver. 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 I know is a first in a conference paper. Feeding this program through the Dafny verifier, using Z3 (version 2.1) [16] as the underlying SMT solver, the verification takes less than 3 seconds.

1 Dafny Language Features

Dafny is an imperative, class-based language. Its basic features and statements are described in Marktoberdorf lectures notes [30]. Its program verifier works by translating a given Dafny program into the intermediate verification language Boogie 2 [2, 36, 29] 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, 19]). The Boogie tool is then used to generate first-order verification conditions that are passed to a theorem prover. For all experiments reported on in this paper, the underlying theorem prover used is Z3 [16]. The Marktoberdorf lecture notes describe the translation into Boogie for the main features of Dafny, including the modeling of the heap and objects, methods and statements, and user-defined functions. In this section, I describe the more advanced features, which are found in the current implementation of the language and 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.

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

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 expressions accurately describe the run-time values to which the expressions 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 algorithm. 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; however, the current implementation leaves this discipline to the user.

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 x stand for **this.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., [26]), whose values change as a consequence of other variables changing.

1.3 Modifications

An important part of a method specification is to say what 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 y . If callers need to know that the method has no effect on y , the method specification can be strengthened by a postcondition **ensures** $y = \mathbf{old}(y)$;, 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, 34], JML uses ownership and data groups [39, 28], and separation logic and permission-based verifiers infer the frame from the given precondition [40, 44, 33]. Inspired by *dynamic frames* [24], 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 R , to dynamically maintain R as the set of objects that are part of the receiver’s representation, and to use R in **modifies** clauses (see [30]).

```

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

```

Recall, this **modifies** clause gives the method license to modify any field of any object in R . If **this** is a member of the set R , then the **modifies** clause also gives the method license to modify R 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. A function specification has three components. To describe these, consider the following example 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 Dafny function is translated into a Boogie function with the name $C.F$; in addition to the explicit Dafny parameters of the function, the Boogie function takes as parameters the heap and the receiver parameter.

The **requires** clause says in which states the function is allowed to be used. Just like Dafny checks for division-by-zero errors, it also checks function preconditions. The example function above gives rise to a Boogie axiom of the form:

$$(\forall \mathcal{H} : HeapType, this : \llbracket C \rrbracket, x : \llbracket T \rrbracket \bullet \\
 GoodHeap(\mathcal{H}) \wedge this \neq null \wedge \llbracket P \rrbracket \Rightarrow C.F(\mathcal{H}, this, x) = \llbracket Body \rrbracket)$$

where $HeapType$ denotes the type of the heap, $\llbracket \cdot \rrbracket$ denotes the translation function from Dafny types and expressions into Boogie types and expressions, and $GoodHeap$ holds of well-formed heaps (see [30]). The Dafny function declaration is allowed to omit the body, in which case this definitional axiom is omitted and the function remains uninterpreted.

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 (see [30]) and produces the following frame axiom:

$$(\forall \mathcal{H}, \mathcal{K} : HeapType, this : \llbracket C \rrbracket, x : \llbracket T \rrbracket \bullet \\
 GoodHeap(\mathcal{H}) \wedge GoodHeap(\mathcal{K}) \wedge \\
 (\forall \langle \alpha \rangle o : Ref, f : Field \alpha \bullet o \neq null \wedge \llbracket o \in R \rrbracket \Rightarrow \mathcal{H}[o, f] = \mathcal{K}[o, f]) \\
 \Rightarrow C.F(\mathcal{H}, this, x) = C.F(\mathcal{K}, this, x))$$

It 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. This axiom shows some details of how Dafny represents the heap, namely as a two-dimensional polymorphic map (see [30]).

At first, it may seem odd to have a frame axiom since the function's definitional axiom (in terms of `Body`) 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. Dafny currently does not implement such scope rules (but see, *e.g.*, [27, 44]). 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 can depend on anything and the frame axiom is not emitted.

When generating axioms, one needs to be concerned about the logical consistency of those axioms. The consistency of the frame axiom follows from the fact that Dafny checks the function body to read only those objects indicated by the `reads` clause. To avoid circularities, the frame axiom and definitional axiom are disabled during this checking itself. This is easily accomplished in Boogie by adding an antecedent `CanAssumeFunctionDefs` to these axioms and adding the assumption that this constant is `true` only when verifying method bodies. (One could imagine a more precise approach where there is one such constant per function, or even that the constant would be replaced by a parameterized function. In my experience with Dafny, I have not yet found the need for such precision.)

But what about the consistency of the definitional axiom? Unless the user-supplied body is restricted, that 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. To that end, Dafny provides a `decreases` clause that is used to give a termination metric. Such a metric is a lexicographic tuple whose (comma delimited) components can be expressions of any type. Dafny enforces that any use of a function within a function body 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.

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

Other languages, like JML, Spec#, and VeriCool [45], 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 method *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 [15, 32] to mention but a couple of sources). In contrast, the treatment of mathematical functions in Dafny (and other logics that only provide functions, not statements and other programming constructs) is simple. I conclude with a slogan: pure methods are hard, functions are easy.

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 approach exist, for example built around ownership

systems [13, 12], explicit validity bits [3], separation logic [42], region logic [1], and permissions [9]. Dafny follows an approach inspired by dynamic frames [24], where the idea is to specify frames as dynamically changing sets and where the consistency of data structures (that is, *object invariants* [38, 3]) are specified by validity functions [18, 35, 40]. 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 R: set<object>;
  function Valid(): bool
    reads {this} ∪ R;
    { this ∈ R ∧ ... }
  method Init()
    modifies {this};
    ensures Valid() ∧ fresh(R - {this});
    { R := {this}; ... }
  method Update()
    requires Valid();
    modifies R;
    ensures Valid() ∧ fresh(R - old(R));
    { ... }
  ...
}

```

The ghost variable `R` is the dynamic frame of the object's representation. The Dafny program needs to explicitly update the variable `R` 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 would then typically allocate and initialize an object as follows:

```

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

```

Method `Init` says it will modify the fields of the object being initialized, which includes the ghost field `R`. Its postcondition says the method will return in a state where `Valid()` is **true**. Since `Init` is allowed to modify `R`, it declares a postcondition that says something about how it changes `R`. 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 `R`, except **this** itself, have been allocated by `Init`. Thus, the typical client above can conclude that `c.R` 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 `R`, the postcondition **fresh**(`..`) promises to add only newly allocated objects to `R`, 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 [30].

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()  $\wedge$  that.Valid();
  modifies R  $\cup$  that.R;
  ensures Valid()  $\wedge$  fresh(R - old(R) - old(that.R));

```

Here, nothing is said about the final value of `that.R`, but the caller cannot assume **this** and `that` to have disjoint representations after the call. Moreover, the value of `that.Valid()` is also underspecified on return, so the caller cannot assume `that` to still be in a consistent state. One may need to strengthen the precondition above with $R \not\cap \text{that.R}$, which says that the representations of **this** and `that` are disjoint, or perhaps with $R \cap \text{that.R} \subseteq \{\text{null}\}$, 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 this 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)  $\wedge$  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 here. The reason for this primarily has to do with values in the heap. Let me explain.

Every field in Dafny translates into a unique constant that is used when the field is selected. For example, the Dafny expression `p.a` is translated into the Boogie map-select expression $\mathcal{H}[p, \text{Pair}.a]$, where \mathcal{H} denotes the heap and *Pair.a* is the name of the constant corresponding to the field `a` in class *Pair* (in Boogie, dot is just another character that can be used in identifier names). A field `x` of a type `T` translates into a Boogie constant of type *Field* $\llbracket T \rrbracket$, where *Field* is a user-defined unary type constructor [30]. The point is that there is just one constant and it has a single type. This is important, because it allows generic code to be verified just once; for example, the implementation of the Flip method is verified once and for all.

So, 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 very 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 [29], 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

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

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 $(\forall i \bullet \dots s[i] \dots)$, 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 [30] 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 is 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. a 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 in Dafny, 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 never a 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`:

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

(For brevity, I omitted the \mathcal{H} and *this* arguments to `Tree.Branch`.)

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 stratosphere. Dafny does not actually emit an axiom for this property (which otherwise would simply have postulated that *rank* returns 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 Boogie to support algebraic datatypes (which it 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* [17], see also [31]). The problem can be mitigated by specifying triggers that are structurally larger than the new terms the instantiations produce. Following VeriFast [23], 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:

$$(\forall u, v: \text{Datatype} \bullet \text{LeafCount}(\text{Tree.Branch}(u, v)) = \text{LeafCount}(u) + \text{LeafCount}(v))$$

where the trigger is specified to be the left-hand side of the equality. 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*:

$$(\forall d: \text{Datatype} \bullet \text{LeafCount}(d) = \text{LeafCount}(b0(d)) + \text{LeafCount}(b1(d)))$$

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; }
```

From this lower-bound check on integers, the stratified datatypes, and the fact that sets are finite in Dafny, well-foundedness follows. In comparison, proving termination of the Schorr-Waite algorithm using Caduceus [19], 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 [22].

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

Dafny currently does not support **decreases** clauses for methods, so there is no static protection against infinite recursion. Adding them will look very much like those for functions. This would be interesting to do, because it is not at all clear to me how one writes termination metrics suitable for modular verification, that is, writing the termination metric of a method in such a way that future callers can also write suitable termination metrics.

2 Case Study: Schorr-Waite Algorithm

In this section, I present the entire Dafny program text for the famous Schorr-Waite algorithm [43], for which many have constructed proofs (see, *e.g.*, [10, 0, 37, 22, 11]). Using this 120-line program as input, Dafny (using Boogie and Z3) verifies its correctness in less than 3 seconds. To my knowledge, the previous shortest mechanical-verifier input for this algorithm was the 400 lines of Isabelle proof scripts by Mehta and Nipkow [37], 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.

The Schorr-Waite algorithm marks all nodes reachable in a graph from a given root node. 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. 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 graph, and (C3) the algorithm terminates. Let me now give a tour of the program.

Class `Node` (line 0 ff. 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` (line 7 ff.) represents lists of `Node`'s and is used by function `Reachable` (line 13) to describe the list of intermediate nodes between `from` and `to`. Recursive function `ReachableVia` (line 19) is defined recursively according to that list of intermediate nodes. Reachability predicates are notoriously difficult for first-order SMT solvers, but since 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 29 and its specification is given on lines 30–42. 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 37–38, (C1) on line 40, and (C2) on line 42. One of the noteworthy things 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 [22], which they describe as “the trickiest annotation”). Correctness property (C3) is implicit, since Dafny checks that all loops terminate (and since there are no recursive methods in this program).

The algorithm is implemented by a loop with 3 cases, 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 on lines 49, 96, and 107, plays a vital part in the proof. It is used in most of the loop invariants. Lines 75–77 declare the relation between `stackNodes` and the Schorr-Waite reversed edges.

The loop invariant goes to established the postconditions as follows: Correctness property (C0) is maintained as a loop invariant (lines 62–63), except for those nodes that are on the stack. Ditto for (C2)

(lines 64–65). Correctness property (C1) is also maintained as a loop invariant (line 72), but using the paths stored in the ghost field `pathFromRoot` instead of existentially quantifying over the intermediate nodes on the path. This loop invariant is maintained by making sure that the current node, `t`, is always reachable from the root (line 71), which is always the case for a depth-first traversal.

```

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
14     requires null ∉ S;
15     reads S;
16     decreases 1;
17   { (∃ via: Path • ReachableVia(from, via, to, S)) }
18
19   function ReachableVia(from: Node, via: Path, to: Node, S: set<Node>): bool
20     requires null ∉ S;
21     reads S;
22     decreases 0, via;
23   {
24     match via
25     case Empty ⇒ from = to
26     case Extend(prefix, n) ⇒ n ∈ S ∧ to ∈ n.children ∧ ReachableVia(from, prefix, n, S)
27   }
28
29   method SchorrWaite(root: Node, S: set<Node>)
30     requires root ∈ S;
31     // S is closed under 'children':
32     requires (∀ n • n ∈ S ⇒ n ≠ null ∧ (∀ ch • ch ∈ n.children ⇒ ch = null ∨ ch ∈ S));
33     // the graph starts off with nothing marked and nothing being indicated as currently being visited
34     requires (∀ n • n ∈ S ⇒ ¬n.marked ∧ n.childrenVisited = 0);
35     modifies S;
36     // nodes reachable from 'root' are marked:
37     ensures root.marked;
38     ensures (∀ n • n ∈ S ∧ n.marked ⇒ (∀ ch • ch ∈ n.children ∧ ch ≠ null ⇒ ch ∈ S ∧ ch.marked));
39     // every marked node was reachable from 'root' in the pre-state
40     ensures (∀ n • n ∈ S ∧ n.marked ⇒ old(Reachable(root, n, S)));
41     // the graph has not changed
42     ensures (∀ n • n ∈ S ⇒ n.childrenVisited = old(n.childrenVisited) ∧ n.children = old(n.children));
43   {
44     var t := root;
45     var p: Node := null; // parent of t in original graph
46     ghost var path := #Path.Empty;
47     t.marked := true;
48     t.pathFromRoot := path;
49     ghost var stackNodes := [];
50     ghost var unmarkedNodes := S - {t};
51     while (true)
52       invariant root.marked ∧ t ≠ null ∧ t ∈ S ∧ t ∉ stackNodes;
53       invariant |stackNodes| = 0 ⇔ p = null;
54       invariant 0 < |stackNodes| ⇒ p = stackNodes[|stackNodes|-1];
55       // stackNodes has no duplicates:
56       invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |stackNodes| ⇒ stackNodes[i] ≠ stackNodes[j]);
57       invariant (∀ n • n ∈ stackNodes ⇒ n ∈ S);
58       invariant (∀ n • n ∈ stackNodes ∨ n = t ⇒
59         n.marked ∧ 0 ≤ n.childrenVisited ∧ n.childrenVisited ≤ |n.children| ∧
60         (∀ j • 0 ≤ j ∧ j < n.childrenVisited ⇒ n.children[j] = null ∨ n.children[j].marked));
61       invariant (∀ n • n ∈ stackNodes ⇒ n.childrenVisited < |n.children|);

```

```

62   invariant (∀ n • n ∈ S ∧ n.marked ∧ n ∉ stackNodes ∧ n ≠ t ⇒
63     (∀ ch • ch ∈ n.children ∧ ch ≠ null ⇒ ch ∈ S ∧ ch.marked));
64   invariant (∀ n • n ∈ S ∧ n ∉ stackNodes ∧ n ≠ t ⇒
65     n.childrenVisited = old(n.childrenVisited));
66   invariant (∀ n • n ∈ stackNodes ∨ n.children = old(n.children));
67   invariant (∀ n • n ∈ stackNodes ⇒
68     |n.children| = old(|n.children|) ∧
69     (∀ j • 0 ≤ j ∧ j < |n.children| ⇒ j = n.childrenVisited ∨ n.children[j] = old(n.children[j])));
70   // every marked node is reachable
71   invariant old(ReachableVia(root, path, t, S));
72   invariant (∀ n, pth • n ∈ S ∧ n.marked ∧ pth = n.pathFromRoot ⇒ old(ReachableVia(root, pth, n, S)));
73   invariant (∀ n • n ∈ S ∧ n.marked ⇒ old(Reachable(root, n, S)));
74   // the current values of m.children[m.childrenVisited] for m's on the stack
75   invariant 0 < |stackNodes| ⇒ stackNodes[0].children[stackNodes[0].childrenVisited] = null;
76   invariant (∀ k • 0 < k ∧ k < |stackNodes| ⇒
77     stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k-1]);
78   // the original values of m.children[m.childrenVisited] for m's on the stack
79   invariant (∀ k • 0 ≤ k ∧ k+1 < |stackNodes| ⇒
80     old(stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k+1]);
81   invariant 0 < |stackNodes| ⇒
82     old(stackNodes[|stackNodes|-1].children[stackNodes[|stackNodes|-1].childrenVisited] = t;
83   invariant (∀ n • n ∈ S ∧ ¬n.marked ⇒ n ∈ unmarkedNodes);
84   decreases unmarkedNodes, stackNodes, |t.children| - t.childrenVisited;
85   {
86     if (t.childrenVisited = |t.children|) {
87       // pop
88       t.childrenVisited := 0;
89       if (p = null) {
90         return;
91       }
92       var oldP := p.children[p.childrenVisited];
93       p.children := p.children[..p.childrenVisited] + [t] + p.children[p.childrenVisited + 1..];
94       t := p;
95       p := oldP;
96       stackNodes := stackNodes[..|stackNodes| - 1];
97       t.childrenVisited := t.childrenVisited + 1;
98       path := t.pathFromRoot;
99     } else if (t.children[t.childrenVisited] = null ∨ t.children[t.childrenVisited].marked) {
100      // just advance to next child
101      t.childrenVisited := t.childrenVisited + 1;
102    } else {
103      // push
104      var newT := t.children[t.childrenVisited];
105      t.children := t.children[..t.childrenVisited] + [p] + t.children[t.childrenVisited + 1..];
106      p := t;
107      stackNodes := stackNodes + [t];
108      path := #Path.Extend(path, t);
109      t := newT;
110      t.marked := true;
111      t.pathFromRoot := path;
112      unmarkedNodes := unmarkedNodes - {t};
113    }
114  }
115 }
116 }
117

```

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, which let me write line 84 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

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 [47]. 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 [29]. VeriFast [23] integrates into C features for writing and proving lemmas.

Others are using SMT solvers for functional correctness verification. Pottier and Régis-Gianas used an SMT solver in their proof of Kaplar and Tarjan’s algorithm for functional double-ended queues [41]. 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 answered the 8 verification challenges proposed by Weide *et al.* [46] in Dafny, except for one aspect of one benchmark, which requires a form of lambda closure (see the Test/VSI-Benchmarks directory at boogie.codeplex.com).

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.

References

- [0] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, September 2003.
- [1] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, July 2008.
- [2] 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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

- [3] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
- [5] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2008.
- [6] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007.
- [7] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The Key Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.
- [8] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, December 1989.
- [9] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
- [10] Manfred Broy and Peter Pepper. Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. *ACM Transactions on Programming Languages and Systems*, 4(3):362–381, July 1982.
- [11] Richard Bubel. The schorr-waite-algorithm. In *Verification of Object-Oriented Software: The Key Approach* [7], chapter 15.
- [12] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 292–310. ACM, November 2002.
- [13] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, pages 48–64. ACM, October 1998.
- [14] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLS 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, August 2009.
- [15] Ádám Péter Darvas. *Reasoning About Data Abstraction in Contract Languages*. PhD thesis, ETH Zurich, 2009. Diss. ETH No. 18622.
- [16] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
- [17] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
- [18] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
- [19] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.
- [20] Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 462–465. Springer, July–August 1996.
- [21] Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäfer, and Thomas Wies. It’s doomed; we can prove it. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress*, volume 5850 of *Lecture Notes in Computer Science*, pages 338–353. Springer, November 2009.

- [22] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 190–199. IEEE Computer Society, September 2005.
- [23] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
- [24] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.
- [25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*, pages 207–220. ACM, October 2009.
- [26] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.
- [27] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.
- [28] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 144–153. ACM, October 1998.
- [29] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at <http://research.microsoft.com/~leino/papers.html>.
- [30] K. Rustan M. Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *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.
- [31] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 615–622. ACM, March 2009.
- [32] K. Rustan M. Leino and Peter Müller. Verification of equivalent-results methods. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 307–321. Springer, March–April 2008.
- [33] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*. Springer, 2009.
- [34] K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *LASER summer school lecture notes*, Lecture Notes in Computer Science. Springer, 2009. To appear.
- [35] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
- [36] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS 2010*, 2010. To appear.
- [37] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1–2):200–227, May–June 2005. 19th International Conference on Automated Deduction (CADE-19).
- [38] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
- [39] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
- [40] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd*

- ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 247–258. ACM, January 2005.
- [41] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, July 2008.
 - [42] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, July 2002.
 - [43] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.
 - [44] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, volume 5653 of *Lecture Notes in Computer Science*. Springer, 2009.
 - [45] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verifier for java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.
 - [46] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 84–98. Springer, October 2008.
 - [47] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 349–361. ACM, June 2008.