Dafny meets the Verification Benchmarks Challenge

K. Rustan M. Leino[0] and Rosemary Monahan[1]

[0] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[1] National University of Ireland, Maynooth, Co.Kildare, Ireland
Rosemary.Monahan@nuim.ie

Abstract. A suite of verification benchmarks for software verification tools and techniques, presented at VSTTE 2008 [11], provides an initial catalogue of benchmark challenges for the Verified Software Initiative. This paper presents solutions to these eight benchmarks using the language and verifier Dafny. A Dafny program includes specifications, code, inductive invariants, and termination metrics. Each of the eight programs is fed to the Dafny verifier, which without further user interaction automatically performs the verification in a few seconds.

0 The Challenge

The motivation from this work comes from the Verified Software Initiative [3] and the suite of eight purposefully designed, incremental benchmarks for tools and techniques to prove total correctness of sequential object-based and object-oriented software, as presented by Weide *et al.* at VSTTE 2008 [11]. A solution to part of the first benchmark is provided, while the main contribution of their paper is the provision of a benchmark suite that aims to support the assessment of verification tools and the assessment of techniques to prove total correctness of the functionality of software. The benchmark suite also aims to provide for the evaluation of the state-of-the-art and to provide a medium that allows researchers to illustrate and explain how proposed tools and techniques deal with known pitfalls and well-understood issues, as well as how they can be used to discover and attack new ones.

In this paper, we contribute to this assessment and evaluation by presenting our solutions to the benchmark problems using the language and verifier Dafny [7, 6]. The full programs are available online at boogie.codeplex.com, located by browsing the source code to access the folder Test/VSI-Benchmarks.

The benchmarks include several requirements of potential solutions, and we believe we meet these:

- Our Dafny programs contain all formal specifications relevant to the benchmarks, including user-defined mathematical functions where needed. Externally to the programs themselves, they rely only on constructs that are part of the Dafny language, which include sets and sequences.
- The Dafny verifier produces verification conditions via the intermediate verification language Boogie 2 [9, 5]. The Boogie program and the resulting verification conditions can be viewed by using, respectively, the `/print` and `/proverLog` switches

of the Dafny verifier. For example, for the benchmark in the file `b3.dfy`, the command `dafny b3.dfy /print:b3.bpl /proverLog:b3.sx` writes out the intermediate Boogie program as `b3.bpl` and the theorem-prover input, in S-expression form, as `b3.sx`.
- The Dafny verification system is described in a conference paper [7] and some lecture notes [6].
- The Dafny verifier checks for total correctness (that is, including termination). It is (designed to be) sound, which means it only proves correct programs to be correct. The Dafny regression test suite (at boogie.codeplex.com) includes many examples of erroneous programs that, as expected, do not verify. Similarly, any change to any part of our Dafny solution programs that renders them incorrect will cause the verifier to no longer verify the programs.
- Dafny uses modular specifications and modular verification. That is, our solutions need not be re-verified when they are incorporated as components in larger programs.
- Finally, we have alerted the VSI repository of our solution programs (although we have not seen them formally enter the repository yet).

1 Dafny

Dafny is an imperative, sequential language that supports user-defined generic classes and algebraic datatypes [7, 6]. The language includes specification constructs, like pre- and postconditions à la Eiffel [10], JML [4], and Spec#[1, 8]. In addition to instance variables and methods, a class can define mathematical functions, which can be used in specifications. Also, the language allows variables to be defined as **`ghost`**, meaning they are used by the verifier but need not be present at run time.

The types supported by Dafny are booleans, mathematical integers, sets, sequences, as well as user-defined classes and algebraic datatypes. Classes can be instantiated, giving rise to object references (*i.e.*, pointers), which makes the language useful for many common applications. However, as an object-oriented language, Dafny does not support subclasses and inheritance, except that the built-in type **`object`** is a supertype of all class types. As such, the language Dafny could perhaps be construed as a more modern version of Pascal or Euclid, or as a safe version of C.

The specification of a method consists of preconditions (introduced by the keyword **`requires`**) and postconditions (keyword **`ensures`**), as well as a modification frame (keyword **`modifies`**), which specifies which objects the method may modify, and a termination metric (keyword **`decreases`**), which gives a well-founded ranking function (also known as a variant function) for proving termination. The specification of a function consists of preconditions, which specify the domain of the function, a dependency frame (keyword **`reads`**), which indicates on which objects the function's value may depend, and a termination metric.

The body of a method consists of an imperative statement, which includes standard constructs like assignments, field updates, object allocation, conditional statements, loops, and method calls. A loop can indicate an inductive loop invariant as well as a termination metric. The body of a function is an expression that defines the value of the function.

The Dafny verifier, whose basic operation is described in detail in Marktoberdorf lecture notes [6] and whose source code is available at boogie.codeplex.com, follows the standard approach of first translating the input program into a program in an intermediate verification language. In particular, it translates Dafny programs into Boogie 2 [9, 5], from which the Boogie tool [0,9] generates first-order verification conditions. These verification conditions, in turn, are fed to an automatic satisfiability-modulo-theories (SMT) solver, namely Z3 [2]. If the SMT solver proves the given formula to be valid, then the Dafny program is correct; if it reports a counterexample, then the Dafny verifier will report an error message about the Dafny program.

2 Benchmarks Solutions in Dafny

In this section, we present our approach to verifying each of the eight benchmarks in Dafny. The problem requirements for each benchmark challenge and our solutions to each follow. In each benchmark, the program text with specifications and other annotations are fed to the Dafny verifier, which then verifies them automatically with no further user guidance.

2.0 Benchmark #1: Adding and Multiplying Numbers

Problem Requirements: *Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.*

We present an iterative solution to addition and a recursive solution to multiplication in Fig. 0. The standard procedural programming constructs arise in these two modular pieces of code which add and multiply two mathematical integers respectively. In both methods, the postcondition is verified, as is termination. The iterative method also requires the verification of a loop invariant.

Our iterative solution to addition is similar to the solution presented in [11], which uses the RESOLVE language and the SplitDecision simplifier.

When we started answering the benchmarks, Dafny did not verify the absence of infinite method recursion. When this was implemented, we were both surprised and embarrassed, at a reported error that showed that we did not do the recursion properly in our initial recursive solution. The **`decreases`** clause in Fig. 0, which uses a lexicographic pair, is used by Dafny to prove termination.

Also, we found that many termination metrics for loops are boring, so we extended the Dafny verifier with a few simple heuristics for guessing a termination metric from the loop guard, in case the loop has no explicit **`decreases`** clause. This simple trick let us simplify the program text for several loops. For example, the termination metrics `-n` and `n`, respectively, for the two loops in method `Add` are found by the verifier.

2.1 Benchmark #2: Binary Search in an Array

Problem Requirements: *Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.*

```
method Add(x: int, y: int) returns (r: int)
  ensures r = x+y;
{
  r := x;
  if (y < 0) {
    var n := y;
    while (n ≠ 0)
      invariant r = x+y-n ∧ 0 ≤ -n;
    {
      r := r - 1; n := n + 1;
    }
  } else {
    var n := y;
    while (n ≠ 0)
      invariant r = x+y-n ∧ 0 ≤ n;
    {
      r := r + 1; n := n - 1;
    }
  }
}
method Mul(x: int, y: int) returns (r: int)
  ensures r = x*y;
  decreases x < 0, x;
{
  if (x = 0) {
    r := 0;
  } else if (x < 0) {
    call r := Mul(-x, y); r := -r;
  } else {
    call r := Mul(x-1, y); call r := Add(r, y);
  }
}
```

Fig. 0. Benchmark #1: Adding and multiplying numbers.

The binary search algorithm is straightforward to specify, implement, and verify. We include the verified binary search method specification and the verified loop conditions in Fig. 1 and Fig. 2, respectively.

Improvements were made to Dafny, as a result of this exercise, as an error in the well-formedness of functions (in particular, the **`requires`** clause) was detected and corrected.

```
method BinarySearch(a: array<int>, key: int) returns (result: int)
  requires a ≠ null;
  requires (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |a| ⟹ a[i] ≤ a[j]);
  ensures -1 ≤ result ∧ result < |a|;
  ensures 0 ≤ result ⟹ a[result] = key;
  ensures result = -1 ⟹ (∀ i • 0 ≤ i ∧ i < |a| ⟹ a[i] ≠ key);
```

Fig. 1. Benchmark #2: Binary search specification.

```
invariant 0 ≤ low ∧ low ≤ high ∧ high ≤ |a|;
invariant (∀ i • 0 ≤ i ∧ i < low ⟹ a[i] < key);
invariant (∀ i • high ≤ i ∧ i < |a| ⟹ key < a[i]);
```

Fig. 2. Benchmark #2: Loop invariants supplied in the binary search method.

2.2 Benchmark #3: Sorting a Queue

Problem Requirements: *Specify a user-defined FIFO queue ADT that is generic (*i.e.*, parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.*

The specification of an integer FIFO queue ADT with standard methods is straightforward in Dafny. However, specifying the queue as a generic type (*i.e.*, `Queue<T>`) highlighted errors in the translation process from Dafny programs to its representation as a Boogie program. These errors were corrected and a generic queue was successfully specified as in Fig. 3.

```
class Queue<T> {
  var contents: seq<T>;

  method Init();
    modifies this;
    ensures |contents| = 0;

  method Enqueue(x: T);
    modifies this;
    ensures contents = old(contents) + [x];

  method Dequeue() returns (x: T);
    requires 0 < |contents|;
    modifies this;
    ensures contents = old(contents)[1..] ∧ x = old(contents)[0];

  function method Head(): T
    requires 0 < |contents|;
    reads this;
  { contents[0] }

  function method Get(i: int): T
    requires 0 ≤ i ∧ i < |contents|;
    reads this;
  { contents[i] }
}
```

Fig. 3. Benchmark #3: Queue specification.

A method to sort the queue, by removing the minimum element in the input queue and inserting it in the output queue, was easily verified. In our solution in Fig. 4, the ghost out-parameter `perm` is used to specify that the `Sort` method's output queue is a permutation of its input queue. When we first attempted this benchmark, ghost variables were not supported in Dafny. The complicated postconditions for the `Sort` method say

that the output queue is sorted, that perm is a permutation of the input queue, and that perm describes the relationship between the input and output queues. Corresponding invariants are required to verify the postcondition and while this leads to a high amount of annotations, Dafny verifies these automatically. We could have existentially quantified over perm, but chose simply to return the witness of that existential quantification.

```
method Sort(q: Queue<int>) returns (r: Queue<int>, ghost perm: seq<int>)
  requires q ≠ null;
  modifies q;
  ensures r ≠ null ∧ fresh(r);  // return a newly allocated Queue object
  ensures |r.contents| = |old(q.contents)|;
  ensures (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |r.contents| ⟹ r.Get(i) ≤ r.Get(j));
  // perm is a permutation
  ensures |perm| = |r.contents|;
  ensures (∀ i • 0 ≤ i ∧ i < |perm|⟹ 0 ≤ perm[i] ∧ perm[i] < |perm| );
  ensures (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |perm| ⟹ perm[i] ≠ perm[j]);
  // the final Queue is a permutation of the input Queue
  ensures (∀ i • 0 ≤ i ∧ i < |perm| ⟹ r.contents[i] = old(q.contents)[perm[i]]);
{
  r := new Queue<int>;
  call r.Init();
  ghost var p := [];
  var n := 0;
  while (n < |q.contents|)
    invariant n ≤ |q.contents| ∧ n = |p|;
    invariant (∀ i • 0 ≤ i ∧ i < n ⟹ p[i] = i);
  {
    p := p + [n]; n := n + 1;
  }
  perm := [];
  ghost var pperm := p + perm;
  while (|q.contents| ≠ 0)
    invariant |r.contents| = |old(q.contents)| - |q.contents|;
    invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |r.contents| ⟹
                r.contents[i] ≤ r.contents[j]);
    invariant (∀ i, j • 0 ≤ i ∧ i < |r.contents| ∧ 0 ≤ j ∧ j < |q.contents| ⟹
                r.contents[i] ≤ q.contents[j]);
    // pperm is a permutation
    invariant pperm = p + perm ∧ |p| = |q.contents| ∧ |perm| = |r.contents|;
    invariant (∀ i • 0 ≤ i ∧ i < |perm| ⟹ 0 ≤ perm[i] ∧ perm[i] < |pperm|);
    invariant (∀ i • 0 ≤ i ∧ i < |p| ⟹ 0 ≤ p[i] ∧ p[i] < |pperm|);
    invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |pperm| ⟹ pperm[i] ≠ pperm[j]);
    // the current array is that permutation of the input array
    invariant (∀ i • 0 ≤ i ∧ i < |perm| ⟹ r.contents[i] = old(q.contents)[perm[i]]);
    invariant (∀ i • 0 ≤ i ∧ i < |p| ⟹ q.contents[i] = old(q.contents)[p[i]]);
  {
    call m,k := RemoveMin(q);
    perm := perm + [p[k]]; // adds index of min to perm
    p := p[k+1..] + p[..k]; // remove index of min from p
    call r.Enqueue(m);
    pperm := pperm[k+1..|p|+1] + pperm[..k] + pperm[|p|+1..] + [pperm[k]];
  }
  assert (∀ i • 0 ≤ i ∧ i < |perm| ⟹ perm[i] = pperm[i]); // lemma needed to trigger axiom
}
```

Fig. 4. Benchmark #3: Implementation of a method to sort the elements of a queue. For space reasons, we have omitted method RemoveMin.

When we first attempted the benchmarks, Dafny had (immutable) mathematical sequences, but no (mutable) arrays. In our initial versions of Benchmarks #2 and #3, we therefore coded up an `Array` class, which we specified in terms of a sequence. When arrays were subsequently added to Dafny, we changed our code to use them directly, as shown here.

Unfortunately, we were unable to sort a generically typed queue. The problem is that a generic comparable type could not be used, as Dafny has no way of specifying that the comparable type's `AtMost` function (as in Fig. 5) is total and transitive. An alternative solution would be to make the `Sort` operation generic, to pass the instantiation type and its comparison operator in as parameters, and to use a precondition to specify the transitive and reflective properties of the comparison operator. However, this is not possible in Dafny either, so we simply sort the generic queue that is instantiated with integers into increasing order.

```
class Comparable {
 function AtMost(c: Comparable): bool;
   reads this, c;
}
```

Fig. 5. Benchmark #3: Comparable class.

2.3 Benchmark #4: Layered Implementation of a Map ADT

Problem Requirements: *Verify an implementation of a generic map ADT, where the data representation is layered on other built-in types and/or ADTs.*

A generic map ADT may be specified using sequences of keys and values where a key stored at position `i` in the sequence of keys has its corresponding value stored at position `i` in the sequence of values. These are defined as **`ghost`** fields as illustrated in Fig. 6, which means they are part of the specification but not part of the compiled program. Built-in equality is used to compare keys. It would be nice to use a user-supplied comparison operator, but Dafny does not have support for that (*cf.* the discussion about operation `AtMost` in Benchmark #2).

An important part of the 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. Frames in Dafny are at the granularity of objects with a dynamic frame (called `Repr` in Fig. 6) maintained as the set of objects that are part of the receiver's representation.

The consistency of an object is often specified using a class invariant [10], but Dafny does not have a class-invariant construct. Instead, consistency is specified using a function (called `Valid` in Fig. 6) that is systematically incorporated in method pre- and postconditions (see also [6, 7]).

We implement the generic map ADT using a linked-list of nodes, where each node contains a key, its value, and a reference to the next node in the linked list as shown in

Fig. 7. Verification of methods to initialise the mapping, find a value given a key, add a key/value pair, remove a key, and find the index at which a key is stored, are provided. We show two of the methods in Fig. 6.

```
class Map<Key,Value> {
  ghost var Keys: seq<Key>;
  ghost var Values: seq<Value>;
  ghost var Repr: set<object>;

  var head: Node<Key,Value>;
  ghost var nodes: seq<Node<Key,Value>>;

  function Valid(): bool
    reads this, Repr;
  { this in Repr ∧
    |Keys| = |Values| ∧ |nodes| = |Keys| + 1 ∧ head = nodes[0] ∧
    (∀ i • 0 ≤ i ∧ i < |Keys| ⟹
        nodes[i] ≠ null ∧ nodes[i] in Repr ∧
        nodes[i].key = Keys[i] ∧ nodes[i].key ∉ Keys[i+1..] ∧
        nodes[i].val = Values[i] ∧ nodes[i].next = nodes[i+1]) ∧
                  nodes[|nodes|-1] = null
  }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this}) ∧ |Keys| = 0;
  {
    Keys := [];
    Values := [];
    Repr := {this};
    head := null;
    nodes := [null];
  }

 method Add(key: Key, val: Value)
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
    ensures (∀ i • 0 ≤ i ∧ i < |old(Keys)| ∧ old(Keys)[i] = key ⟹
                |Keys| = |old(Keys)| ∧ Keys[i] = key ∧ Values[i] = val ∧
                (∀ j • 0 ≤ j ∧ j < |Values| ∧ i ≠ j ⟹
                    Keys[j] = old(Keys)[j] ∧ Values[j] = old(Values)[j]));
    ensures key ∉ old(Keys) ⟹ Keys = [key] + old(Keys);
         ensures Values = [val] + old(Values);
  {
    call p, n, prev := FindIndex(key);
    if (p = null) {
      var h := new Node<Key,Value>;
      h.key := key;  h.val := val;  h.next := head;
      head := h;
      Keys := [key] + Keys;  Values := [val] + Values;
      nodes := [h] + nodes;
      Repr := Repr + {h};
    } else {
      p.val := val;
      Values := Values[n := val];
    }
  }
 //...
}
```

Fig. 6. Benchmark #4: Extract of the map ADT implementation.

```
class Node<Key,Value> {
  var key: Key;
  var val: Value;
  var next: Node<Key,Value>;
}
```

Fig. 7. Benchmark #4: Node class used in the implementation of the map ADT.

2.4 Benchmark #5: Linked Implementation of a Queue ADT

Problem Requirements: *Verify an implementation of the queue type specified for Benchmark #3, using a linked data structure for the representation.*

An implementation of the queue specified in Benchmark #3 is provided in terms of a set of nodes, references to the head and the tail node, and a sequence of values recording the queue contents.

In our solution, which we adapted from an example in the Marktoberdorf lecture notes on Dafny [6], the queue elements stored in nodes are generically typed. Elements are added to the queue by dynamically creating a new node, storing the element to be added in that node, and linking the queue's tail node to the new node that contains the element to be added. Other queue operations are implemented in a similar manner with Dafny's built-in set and sequence data types providing the underlying data structures that allow our solution to be automatically verified.

2.5 Benchmark #6: Iterators

Problem Requirements: *Verify a client program that uses an iterator for some collection type, as well as an implementation of the iterator.*

The implementation of both a collection class and an iterator class was necessary to meet this benchmark in Dafny. A generic collection class was implemented as a sequence of generic types with methods to initialise the collection, return an item, add an item, and get an iterator on collections. See Fig. 8 for an extract of the code. While we acknowledge that our solution does not meet the original challenge problem[0], which requires that an iterator be invalidated whenever its associated collection is updated, our solution does address the challenge as presented in the Weide *et al.* benchmark suite.

Our client program uses an iterator over this collection type. This program stores the elements of the collection in a sequence and we verify that the iterator returns the correct elements. We also verify that the iterator does not destroy the collection that it iterates over.

2.6 Benchmark #7: Input and Output Streams

Problem Requirements: *Specify simple input and output capabilities such as character input streams and output streams. Verify an application program that uses them in conjunction with one of the components from the earlier benchmarks.*

[0] Issued at SAVCBS 2006

```
class Collection<T> {
  var Repr: set<object>;
  var elements: seq<T>;

  function Valid():bool
    reads this, Repr;
  { this in Repr }

  method GetCount() returns (c:int)
    requires Valid();
    ensures 0 ≤ c;
  { c := |elements|; }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this});
  {
    elements := []; Repr := {this};
  }

  method GetIterator() returns (iter:Iterator<T>)
    requires Valid();
    ensures iter ≠ null ∧ iter.Valid();
    ensures fresh(iter.Repr) ∧ iter.pos = -1;
    ensures iter.c = this;
  {
    iter := new Iterator<T>;
    call iter.Init(this);
  }

  //...
}
```

Fig. 8. Sample collection class for use in Benchmark #6.

Our specification of streams is implemented as a stream of integers with methods to create a stream for writing, open a stream, write an integer to a stream, read an integer from a stream, check for the end of a stream, and close a stream. Sample method implementations are presented in Fig. 10.

A client program that reads integers, stores them in a Queue (specified in Benchmark #3), sorts them (using the algorithm from Benchmark #3) and writes them to a stream, is verified using Dafny. Note that we assume finite streams and that if we were required to prove termination, then we would need some way to signal the end of a stream.

2.7 Benchmark #8: An Integrated Application

Problem Requirements: *Verify an application program with a concisely stated set of requirements, where the particular solution relies on the integration of at least a few of the previous benchmarks.*

The application program that we verify implements a dictionary as a mapping between words and sequences of words. To set up the dictionary, we read a stream of words (adapting code from Benchmark #7) and put them into a mapping where the first word in the stream is the term to be defined. The subsequent words on the same line in the stream form that terms definition. Reading the stream again provides the next

```
class Iterator<T> {
  var c: Collection<T>;
  var pos: int;
  var Repr: set<object>;

  function Valid():bool
    reads this, Repr;
  { this in Repr ∧ c ≠ null ∧ -1 ≤ pos ∧ null ∉ Repr }

  method Init(coll:Collection<T>)
    requires coll ≠ null;
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this}) ∧ pos = -1;
    ensures c = coll;
  {
    c := coll;
    pos := -1;
    Repr := {this};
  }

  method MoveNext() returns (b:bool)
    requires Valid();
    modifies Repr;
    ensures fresh(Repr - old(Repr)) ∧ Valid();
    ensures pos = old(pos) + 1 ∧ b = HasCurrent() ∧ c = old(c);
  {
    pos := pos+1;
    b := pos < |c.elements|;
  }

  //...
}
```

Fig. 9. Sample implementation of the iterator class for benchmark #6.

term followed by its definition and so on until nothing remains to be read. The map ADT used in this application is that verified in Benchmark #4. The sort method and the queue data type verified in Benchmark #3, are used to sort the words into alphabetical order. Some sample code is provided in Fig. 11.

For this benchmark, we were unsure about how much would be reasonable to specify. For example, would one want to go as far as to say the output is a stream of characters that, interpreted as HTML and rendered by a standard web browser, provides a list of words with clickable hyperlinks that lead to other words on the page, and all words and definitions shown are those from the input? We chose a specification closer to the other end of the spectrum, namely to specify just enough to prove the absence of run-time errors in the code.

3 Dafny Success

The performance measures of the proofs, all of which are completed automatically, are presented in Fig. 12. Evidence that the tool automatically detects an incorrect solution can be generated by simple edits to the Dafny implementations. As noted in the presentation of our solutions, some errors in Dafny were detected as a result of attempting to

```
class Stream {
  var Repr: set<object>;
  var stream: seq<int>;
  var isOpen: bool;

  function Valid(): bool
    reads this, Repr;
  { null ∉ Repr ∧ this in Repr ∧ isOpen }

  method GetCount() returns (c:int)
    requires Valid();
    ensures 0 ≤ c;
  { c := |stream|; }

  method Create() //writing
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this});
    ensures stream = [];
  {
    stream := [];
    Repr := {this};
    isOpen:= true;
  }

  method Open() //reading
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this});
  {
    Repr := {this};
    isOpen :=true;
  }

  method PutChar(x: int)
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
    ensures stream = old(stream) + [x];
  { stream:= stream + [x]; }
}
```

Fig. 10. Benchmark #7: Sample implementation of input and output streams.

verify our benchmark solutions. These errors were corrected while other Dafny features were added or improved.

Notable enhancements to Dafny, since our first attempt at these benchmarks, include support for verifying recursion termination, heuristics that eliminate the need for many **`decreases`** clauses, the addition of ghost variables, the addition of arrays, and the redesign of the encoding of generics. This redesign includes improvements to the encoding of built-in types **`set`** and **`seq`** (see Boogie/Binaries/DafnyPrelude.bpl on codeplex.boogie.com). The syntax of Dafny has improved with additions such as the $\notin$ operator, introduced on sets and sequences, and the Dafny **`call`** statement which now automatically declares left-hand sides as local variables, if they were not already local variables.

In early versions of our benchmark solutions, we often needed to supply lemmas to assist the automatic verification of sequence properties. Such a lemma is usually supplied as an **`assert`** statement, as shown in Fig. 4. The asserted expression, which is proved by the verifier, mentions terms that are relevant to the program's correctness,

```
var rs := new ReaderStream;
call rs.Open();
var glossary := new Map<Word,seq<Word>>;
call glossary.Init();
var q := new Queue<Word>;
call q.Init();

while (true)
  invariant rs.Valid() ∧ fresh(rs.Repr);
  invariant glossary.Valid();
  invariant glossary ∉ rs.Repr;
  invariant null ∉ glossary.keys;
  invariant (∀ d • d in glossary.values ⟹ null ∉ d);
  invariant q ∉ rs.Repr;
  invariant q.contents = glossary.keys;
  decreases *;  // we leave out the decreases clause - unbounded stream
{
  call term,definition := readDefinition(rs);
  if (term = null) {
    break;
  }
  call present, d := glossary.Find(term);
  if (¬present) {
    call glossary.Add(term,definition);
    call q.Enqueue(term);
  }
}
call rs.Close();
call q,p := Sort(q);
```

Fig. 11. Sample code verified in the solution to Benchmark #8.

Benchmark #	# of Verifications	Time (in seconds)
1	10	3.5
2	6	3.7
3	10	8.0
4	11	4.9
5	22	7.8
6	21	3.9
7	23	3.9
8	42	5.1

Fig. 12. Performance measurements of program verifications. Times shown take the average of 3 runs. For some of the benchmarks, these times also include some client code we added to the program files.

and the mention of these terms prompts the SMT solver to instantiate related axioms and other quantifications, as may be required in the proof. The treatment of sequences in Dafny has since been improved, allowing the verification to be completed mostly without the use of lemmas to trigger the proof.

In early versions of our benchmark solutions, we often needed to supply lemmas to assist the verifier in triggering the correct axioms to verify sequence properties. The treatment of sequences in Dafny has since been improved, allowing the verification to be completed mostly without the use of lemmas to trigger the proof.

We are aware that there are some limitation to our solutions. For example, Dafny uses mathematical integers, not the bounded integers found in most popular programming languages. This means that there is no issue of overflow in Dafny, which (is really a feature but) may be considered a limitation of what is verified. A more obvious shortcoming is in meeting Benchmark #3, where we couldn't sort a generic `Queue` into some client-defined order. However, this too is informative and will help us to compare Dafny with other tools that take up the benchmark challenges.

4 Conclusions

We have attempted to meet the given verification benchmarks using the language and verifier Dafny and are pleased with the results. We were not concerned about how the benchmarks themselves were designed, but tried to meet their original specifications. Our experience was that it is often tempting to make the benchmark fit your tool rather than to try to solve the benchmark itself. We hope that ongoing discussions with the benchmark authors, and discussions with others who will write solutions for these benchmarks, will help shape future revisions of these and other benchmarks. In particular, we would like to see that the next revision of the benchmarks would make it easier to constrain solutions so that it is easier to compare language features and verification techniques. A more detailed framework for comparing languages and verifiers, to be used in conjunction with the benchmarks, is also desirable. Further benchmarks that we hope to explore include the original iterator and the composite problems[1] which would help compare Dafny to other verifiers based on the dynamic framing paradigm.

The exercise allowed us to explore the strengths and weaknesses of Dafny and contributed towards improving both the syntax of the language and the verification process. We strongly encourage others to take up the verification challenge as this will help the community to compare and improve existing languages and tools. Having a number of problems solved in different languages will also assist researchers in learning about another's favourite language and how it compares to their own.

Acknowledgments We thank the authors of [11] and their research students for feedback on our initial attempts at these verification benchmarks. We also thank the anonymous referees, for their thoughtful and helpful comments.

References

0. 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.
1. 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*

[1] Presented at SAVCBS 2006 and SAVCBS 2008, respectively

Smart Devices, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

2. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
3. C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Computing Surveys*, 41(4):22:1–22:8, October 2009.
4. 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.
5. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/ leino/papers.html.
6. 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.
7. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, 2010. To appear.
8. K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In Peter Müller, editor, *LASER Summer School 2007/2008*, volume 6029 of *Lecture Notes in Computer Science*, pages 91–139. Springer, 2010.
9. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, March 2010.
10. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
11. 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.