In: Proceedings of VSTTE 2010 (Verified Software: Theories, Tools, and Experiments) Workshops, August 2010.

A Review of Verification Benchmark Solutions Using Dafny

Derek Bronish and Bruce W. Weide

The Ohio State University, Columbus OH 43210, USA
{bronish,weide}@cse.ohio-state.edu

Abstract. Proposed solutions to a collection of software verification "challenge problems" have been undertaken by a group using Dafny. The techniques employed to solve these problems present insights into the Dafny specification and verification process. Solutions to key problems including binary search of an array and proof of correctness of data representation are reviewed, with observations about language design and modularity, among other issues.

1 Introduction

In an effort to focus and unify the efforts of the software verification community, the authors of [1] (including the two of us) have proposed a number of incremental "benchmarks." These challenge problems present a wide assortment of potential impediments to automated verification in a manner that progressively increases in difficulty. The benchmarks also elaborate on the data that should be provided in each solution. This includes code, specifications, verification conditions, buggy programs that the system can identify as incorrect, and so on. Each properly documented solution should be, in essence, a self-contained software verification case study.

Dafny [2] is a programming language whose semantics is defined by translation into Boogie [3], thus allowing verification conditions to be generated and then proven using Z3 [4]. The language features Java-like reference semantics and a unique selection of primitives, including generic `set` and `seq` collection types. Dafny-based solutions to all eight of the benchmark problems have recently been claimed [5] and made publicly available [6]. We describe three of these benchmark solutions and discuss the approach used in them.

The benchmark problems, and the specific guidelines for how solutions should be structured, are intended to establish a detailed and rigorous basis on which to compare and contrast different approaches to software verification. We do not seek to position ourselves as *de facto* official reviewers of proposed benchmark solutions simply on the grounds that we suggested the benchmarks. Instead, we hope that this paper will serve as a model for future benchmark solution submissions and analyses, giving a general idea of how a review might be reported and encouraging others to participate in this conversation. Other fields of study

thrive on reviews and responses as effective means for disseminating ideas[1]; benchmarks and challenge problems provide a similar opportunity in software engineering.

For each of the reviewed benchmarks, we quote the original problem statement from [1], present the Dafny solution, and discuss its merits. We conclude with general remarks about the Dafny approach and the lessons learned in undertaking this analysis.

2 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.

Dafny does not provide arrays as a primitive in the programming language. Therefore, Benchmark 2 is where we first see Dafny's approach to user-defined data types. The benchmark does not require that the data type itself be verified, only that client code using it can be verified relative to the correctness of the data type. Issues of data representation are covered in later benchmarks; the issue for now is simply the *specification* of the array.

The Dafny `Array` component is shown in Figure 1. Note that for simplicity, the Dafny solution fixes the contents of an array to be `int`s. This is permissible; generics are not explicitly addressed until the fourth benchmark problem.[2] An array of integers is treated as a `seq` of `int`. This is indicative of a surprising characteristic of Dafny data types: it does not seem possible to specify the behavior of an array without making a commitment to its representation. A `seq` is not an abstract mathematical entity that can be used to describe the behavior of `Array` operations; instead it is a real programmatic object that can only be used in specifications if it is an actual field of a particular `Array` representation.[3]

Given this state of affairs, one may wonder how a client could possibly write specifications (e.g., loop invariants) that involve `Array`s in a modular manner, without committing to any particular `Array` implementation. Dafny does not provide any special new mechanism for this, but rather suggests as a good practice that classes provide a sufficiently robust collection of "pure" functions to facilitate client specification writing.

`Get` is an example of one of these "mathematical" functions, whose bodies are expressions rather than statements in the programming language. Although there is currently no compiler for Dafny, these functions are intended to be executable. One interesting question is how exactly these will be implemented by a Dafny compiler. One may also wonder what performance characteristics `Array` will offer when represented as a `seq`.

---

[1] One famous example from cognitive science is Chomsky's review of Skinner [7].

[2] Actually, Benchmark 3 calls for a generic datatype, but the Dafny solution does not meet this requirement.

[3] Dafny does allow "ghost" variables, but this amounts to the same problem for purposes of present discussion; a ghost variable is treated identically to a non-ghost variable (i.e., via explicit updates).

```
class Array {
  var contents: seq<int>;
  method Init(n: int);
    requires 0 <= n;
    modifies this;
    ensures |contents| == n;
  function Length(): int
    reads this;
  { |contents| }
  function Get(i: int): int
    requires 0 <= i && i < |contents|;
    reads this;
  { contents[i] }
  method Set(i: int, x: int);
    requires 0 <= i && i < |contents|;
    modifies this;
    ensures |contents| == |old(contents)|;
    ensures contents[..i] == old(contents[..i]);
    ensures contents[i] == x;
    ensures contents[i+1..] == old(contents[i+1..]);
}
```

Fig. 1. Dafny `Array` component.

The binary search implementation is shown in Figure 2. We see that the loop can exit unnaturally (i.e., without the loop guard evaluating to false), and so the verification conditions must account for this. Unfortunately, the publicly-available Dafny solutions do not include verification conditions for inspection. Also of note is the arithmetic used to compute `mid`. It is carefully formulated to avoid overflow, which is a subtle bug that is specifically enunciated in the benchmark statement. It would have been nice to see this bug manifest in the code, and then be discovered by a failed verification, but built-in `int`s are unbounded in Dafny.

3 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.

In this benchmark, the issue of parameterizing the sorting operation by an ordering is deemed to be "central," and the invariants involved for even straightforward sorting algorithms are non-trivial.

In Dafny, one can define a function only by writing an explicit function body. This means it is not possible to specify a generic comparison function. So, the proposed solution proceeds by fixing the queue contents to be `int`s and

```
method BinarySearch(a: Array, key: int) returns (result: int)
  requires a != null;
  requires (forall i, j :: 0 <= i && i < j && j < a.Length() ==>
                a.Get(i) <= a.Get(j));
  ensures -1 <= result && result < a.Length();
  ensures 0 <= result ==> a.Get(result) == key;
  ensures result == -1 ==>
            (forall i :: 0 <= i && i < a.Length() ==> a.Get(i) != key);
{
  var low := 0;
  var high := a.Length();

  while (low < high)
    invariant 0 <= low && high <= a.Length();
    invariant (forall i :: 0 <= i && i < low ==> a.Get(i) < key);
    invariant (forall i :: high <= i && i < a.Length() ==>
                  key < a.Get(i));
    decreases high - low;
  {
    var mid := low + (high - low) / 2;
    var midVal := a.Get(mid);

    if (midVal < key) {
      low := mid + 1;
    } else if (key < midVal) {
      high := mid;
    } else {
      result := mid; // key found
      return;
    }
  }
  result := -1;  // key not present
}
```

Fig. 2. Binary search implemented in Dafny.

the ordering to be standard numeric "less than or equal to." This skirts the central issue of parameterization by an operation mentioned above, but leaves a challenging benchmark nonetheless.

Selection sort, implemented with a `RemoveMin` helper operation, is shown in Figures 3 and 4. Clearly, the solution is annotation-heavy. This is primarily due to the need to specify properties about permutations, e.g., that the outgoing (sorted) value of the queue is a permutation of the incoming value.

One can imagine that certain properties, e.g., two strings being permutations of each other, are so fundamental and useful that one may wish to isolate their mathematical definitions inside some sort of atomic unit that can be concisely referred to in specifications. This could be viewed as an analog of procedural abstraction, and indeed one could propose using Dafny's `function`s to serve in this role. However, it is difficult to see exactly where such a function should reside. `Queue` would be too limited a scope, because of course we may wish to use it with other user-defined data types that are implemented with a `seq` — for example a stack, or even the `Array` seen in Benchmark 2. A language system that uses pure mathematical modeling seems better suited to this kind of design, because it would allow such definitions to reside with the rest of the mathematical theory used in the modeling, rather than in some particular program component, where it is hard to reuse.

4 Benchmark 4: Layered Implementation of 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.

The Dafny solution shown in Figure 5 uses two `seq`s to represent the map: one to hold the keys and one to hold the associated values, with the association being established by matching indices.

At this point, issues of modularity and abstraction are brought to center stage. Consider, for example, the `FindIndex` method. Without even fully understanding the contract, an intuitive gloss of the method's name indicates that implementation details have been exposed here. From an abstract point of view, a map has no notion of "index," and so `FindIndex` is in principle surprising.

One could respond that `FindIndex` is intended to be "private." Dafny currently has no enforced access restriction; otherwise the authors would surely use it here. However, even if there were such a mechanism, there are further violations of representation encapsulation in the `Map` class. One example is the ensures clause of `Find`, which exposes the fact that the class contains fields `keys` and `values` of type `seq`. Imagine that one might prefer to implement `Map` using two `Queue`s in the representation. To describe the exact same behavior, the specifications would have to change so that all mentions of `keys` and `values` instead refer to `keys.contents` and `values.contents`, respectively.[4]

[4] In fact the situation becomes worse when one considers that there may be multiple implementations of `Queue`, each exposing its own representation in different ways and with different field names.

```
  method RemoveMin(q: Queue<int>) returns (m: int, k:int)
    requires q != null && |q.contents| != 0;
    modifies q;
    ensures |old(q.contents)| == |q.contents| + 1;
    ensures  0 <= k && k < |old(q.contents)| && old(q.contents[k]) == m;
    ensures (forall i :: 0 <= i && i < |q.contents| ==> m <= q.contents[i]);
    ensures q.contents == old(q.contents)[k+1..] + old(q.contents)[..k];
  {
    var n := |q.contents|;
    k := 0;
    m := q.Head();
    var j := 0;

    while (j < n)
      invariant j <= n;
      invariant q.contents == old(q.contents)[j..] + old(q.contents)[..j];
      invariant 0 <= k && k < |old(q.contents)| && old(q.contents)[k] == m;
      invariant (forall i ::0<= i && i < j ==> m <= old(q.contents)[i]);
      decreases n-j;
    {
      call x:= q.Dequeue();
      call q.Enqueue(x);
      if ( x < m) {k := j; m := x;}
      j:= j+1;
    }

    j := 0;
    while (j < k)
      invariant j <= k;
      invariant q.contents == old(q.contents)[j..] + old(q.contents)[..j];
      decreases k-j;
    {
      call x := q.Dequeue();
      call q.Enqueue(x);
      j:= j +1;
    }

     call m:= q.Dequeue();
   }
}
```

Fig. 3. Dafny `RemoveMin` implementation.

```
method Sort(q: Queue<int>) returns (r: Queue<int>, perm: seq<int>)
  requires q != null;
  modifies q;
  ensures r != null && fresh(r) && |r.contents| == |old(q.contents)|;
  ensures (forall i, j :: 0 <= i && i < j && j < |r.contents| ==>
            r.Get(i) <= r.Get(j));
  ensures |perm| == |r.contents|; // ==|pperm|
  ensures (forall i: int :: 0 <= i && i < |perm| ==>
            0 <= perm[i] && perm[i] < |perm|);
  ensures (forall i, j: int :: 0 <= i && i < j && j < |perm| ==>
            perm[i] != perm[j]);
  ensures (forall i: int :: 0 <= i && i < |perm| ==>
            r.contents[i] == old(q.contents)[perm[i]]);
{ r := new Queue<int>;
  call r.Init();
  // initialize ghostvar p so p=<0,...,|q.contents|-1> (code omitted)
  perm:= []; ghost var pperm := p + perm;
  while (|q.contents| != 0)
    invariant |r.contents| == |old(q.contents)| - |q.contents|;
    invariant (forall i, j :: 0 <= i && i < j && j < |r.contents| ==>
                r.contents[i] <= r.contents[j]);
    invariant (forall i, j :: 0 <= i && i < |r.contents| && 0 <= j &&
                j < |q.contents| ==> r.contents[i] <= q.contents[j]);
    invariant pperm==p+perm&& |p|==|q.contents|&& |perm|==|r.contents|;
    invariant (forall i: int :: 0 <= i && i < |perm| ==>
                0 <= perm[i] && perm[i] < |pperm|);
    invariant (forall i:int::0<=i && i<|p|==> 0<=p[i] && p[i]<|pperm|);
    invariant (forall i, j: int :: 0 <= i && i < j && j < |pperm| ==>
                pperm[i] != pperm[j]);
    invariant (forall i: int :: 0 <= i && i < |perm| ==>
                r.contents[i] == old(q.contents)[perm[i]]);
    invariant (forall i: int :: 0 <= i && i < |p| ==>
                q.contents[i] == old(q.contents)[p[i]]);
    decreases |q.contents|;
  {
    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]];
  }
  //lemma needed to trigger axiom
  assert (forall i:int :: 0<=i && i < |perm| ==> perm[i] == pperm[i]);
}
```

Fig. 4. Dafny `Sort` implementation. A trivial portion of code that properly initializes the ghost `seq p` is omitted for presentation purposes, but is correctly annotated and verified in the full solution.

```
class Map<Key,Value> {
  var keys: seq<Key>;
  var values: seq<Value>;

  function Valid(): bool
    reads this;
  {
    |keys| == |values| &&
    (forall i, j :: 0 <= i && i < j && j < |keys| ==> keys[i] != keys[j])
  }
  method Find(key: Key) returns (present: bool, val: Value)
    requires Valid();
    ensures !present ==> key !in keys;
    ensures present ==> (exists i :: 0 <= i && i < |keys| &&
                                     keys[i] == key && values[i] == val);
  {
    call j := FindIndex(key);
    if (j == -1) {
      present := false;
    } else {
      present := true;
      val := values[j];
    }
  }
  method FindIndex(key: Key) returns (idx: int)
    requires Valid();
    ensures -1 <= idx && idx < |keys|;
    ensures idx == -1 ==> key !in keys;
    ensures 0 <= idx ==> keys[idx] == key;
  {
    var j := 0;
    while (j < |keys|)
      invariant j <= |keys|;
      invariant key !in keys[..j];
      decreases |keys| -j;
    {
      if (keys[j] == key) {
        idx := j;
        return;
      }
      j := j + 1;
    }
    idx := -1;
  }
}
```

Fig. 5. An implementation of a map component using two parallel `seq`s. Methods for initializing the map, adding values, and removing values have been omitted solely for conciseness of presentation.

This is symptomatic of the aforementioned coupling between abstract specifications and concrete realizations. In the comments accompanying their Benchmark 4 solution, the authors note that this problem would be easier to solve if "Dafny had a built-in map type that could be used in specifications," but of course one cannot expect such an entity to exist for every kind of data type one may need to implement. Pure mathematical modeling is a convenient way of avoiding these issues and decoupling code from specifications, but it introduces complexities — such as establishing a correspondence between the programmatic entities participating in the data representation and the model used to express the desired behavior [8, 9] — that Dafny is not yet engineered to handle.

5 Conclusion

The most important issue brought to light by these proposed benchmark solutions has to do with Dafny's treatment of mathematical modeling. Dafny components tightly couple specifications with their implementations — indeed there seems to be no mechanism for writing specifications in a separate code unit.

In Dafny, some "mathematical" entities such as sets and sequences are actually primitives in the programming language. They serve a dual role: as descriptors for component behavior, and as actual fields of the class implementing that behavior. For example, both `Array` and `Queue` are specified and represented with a Dafny `seq`. One advantage of this approach is that in situations where a particular property is needed both for specification and for implementation purposes, a single entity (a Dafny `function`) can be used to define it. One example of this is the `Get` function seen in Benchmark 2.

On the other hand, this conflation of code and mathematics also poses some threats to modularity. For example, consider the issue of multiple implementations. In Benchmark 3, we see that class fields (e.g., `contents`) are allowed to appear in client specifications. Now imagine that there were two implementations of `Queue`: one which has a `seq` called `contents`, and another which uses a `set` called `contents`, likely in conjunction with some other fields. Exchanging one implementation with the other would inadvertently change the meaning (or even the well-formedness) of client specifications. The recommended idiom is of course to decouple the implementation from the specifications by only mentioning functions rather than fields on the client side, but the proposed solution to Benchmark 3 provides evidence that this is not currently possible in all cases.[5]

As authors of [1], we are pleased that the verification community has begun to take on the challenges proposed therein. Dafny's ability to address all eight benchmarks is impressive and appreciated; unfortunately space does not permit a presentation or review of all solutions here.

The collection, cataloging, and analysis of multiple solutions to the verification benchmark problems provide a valuable opportunity for detailed analyses and comparison. This work presents an initial discussion; we hope others follow.

[5] The authors note in the comments of their solution to Benchmark 3 that attempting to use `Get` instead of referring to `contents` does not work.

6 Acknowledgements

First and foremost we thank K. Rustan M. Leino and Rosemary Monahan for their impressive work on the Dafny benchmark solutions and their willingness to share their results, insights, and answers to questions with us. The authors are also grateful for the constructive feedback from Bruce Adcock, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Jason Kirschenbaum, Bill Ogden, Murali Sitaraman, Hampton Smith, Aditi Tagore, and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. ECCS-0931669, DMS-0701260 and CCF-0811737. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B.M., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Verified Software: Theories, Tools, and Experiments (VSTTE). (2008) 84–98
2. Leino, K.R.M.: Specification and verification of object-oriented software. In Broy, M., Sitou, W., Hoare, T., eds.: Engineering Methods and Tools for Software Safety and Security. Volume 22 of NATO Science for Peace and Security Series D: Information and Communication Security. IOS Press (2009) 231–266 Summer School Marktoberdorf 2008 lecture notes.
3. Barnett, M., Chang, B.Y.E., Deline, R., Jacobs, B., Leino, K.R.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science, Springer (2006) 364–387
4. Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2008)
5. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). (2010)
6. Microsoft Research: The Boogie Project Codeplex Page. In: http://boogie.codeplex.com/SourceControl/list/changesets. (last accessed March 29th 2010)
7. Chomsky, N.: A review of B.F. Skinner's Verbal Behavior. Language **35**(1) (1959) 26–58
8. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**(4) (1972) 271–281
9. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Trans. Softw. Eng. **23**(3) (1997) 157–170