# Array Programming in Whiley

David J. Pearce

School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

djp@ecs.vuw.ac.nz

## Abstract

Arrays are a fundamental mechanism for developing and reasoning about programs. Using them, one can easily encode a range of important algorithms from various domains, such as for sorting, graph traversal, heap manipulation and more. However, the encoding of such problems in traditional languages is relatively opaque. That is, such programming languages do not allow those properties important for the given problem to be encoded within the language itself and, instead, rely up on programmer-supplied comments.

This paper explores how array-based programming is enhanced by programming languages which support specifications and invariants over arrays. Examples of such systems include Dafny, Why3, Whiley, Spec# and more. For this paper, we choose Whiley as this language provides good support for array-based programming. Whiley is a programming language designed for verification and employs a *verifying compiler* to ensure that programs meet their specifications. A number of features make Whiley particularly suitable for array-based programming, such as type invariants and abstract properties. We explore this through a series of worked examples.

## 1. Introduction

Arrays provide a powerful and oft-overlooked primitive for use in programming. Arrays can be used to describe fundamental algorithms from a wide range of areas and disciplines. Sorting algorithms are, of course, one such example. With multi-dimensional arrays and integer types we can easily encode more complex data structures, such as *sets*, *trees* and *graphs*, etc. Even *linked structures*, typically implemented with references, can be implemented using arrays (i.e. where array indices replace references, etc). Unfortunately, modern languages make array-based programming feel very "low level" and, instead, promote the use of linked structures by making `struct` and/or `class` constructs feel more "sophisticated". The essential argument of this paper is that, with proper programming language support, array-based programming can be made to feel just as sophisticated.

Numerous programming languages have been developed with first-class support for specification in the form of pre- and post-conditions and which typically emphasise the use of automated theorem provers, such as Simplify [1] or Z3 [2], for static checking. Good examples include ESC/Java [3], Spec# [4], Dafny [5], Why3 [6], VeriFast [7], SPARK/Ada [8], and Whiley [9]. The development of such tools has proved something of a boon for array-based programming. This is because the specification languages they employ, as well as the underlying automated theorem provers, typically have limited support for dealing with linked data structures [10]. To work-around this, tool developers and practitioners have placed great emphasis on the use of arrays for encoding problems of interest.

***Contribution.*** This paper explores how array-based programming is enhanced by programming languages which support specifications and invariants over arrays. This is achieved through a series of worked examples written in the Whiley programming language.

## 2. Background

In this section, we introduce the Whiley language in the context of software verification through a series of examples. We do not provide an exhaustive examination of the language and, instead, the interested reader may find more detailed introductions elsewhere [9].

### 2.1 Overview

The Whiley programming language has been developed from the ground up to enable compile-time verification of programs [9]. The Whiley Compiler (WyC) attempts to ensure that every function in a program meets its specification. When it succeeds in this endeavour, we know that: 1) all function post-conditions are met (assuming their pre-conditions held on entry); 2) all invocations meet their respective function's pre-condition; 3) runtime errors such as divide-by-zero, out-of-bounds accesses and null-pointer dereferences are impossible. Note, however, that such pro-

grams may still loop indefinitely and/or exhaust available resources (e.g. RAM).

### 2.2 Example 1 — Preconditions and Postconditions

Whiley allows explicit *pre-* and *post-conditions* to be given for functions. For example, the following function accepts a positive integer and returns a natural number:

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
    //
    return x - 1
```

Here, `decrement()` includes **`requires`** and **`ensures`** clauses which correspond (respectively) to its *precondition* and *postcondition*. In this context, `y` represents the return value and may be used only within the **`ensures`** clause. The Whiley compiler statically verifies this function meets its specification (note, integers are unbounded and cannot underflow).

The Whiley compiler reasons about functions by exploring their control-flow paths. As it learns more about the variables encountered, it takes this into account. For example:

```
function max(int x, int y) -> (int z)
// Must return either x or y
ensures x == z || y == z
// Return must be as large as x and y
ensures x <= z && y <= z:
    //
    if x > y:
        return x
    else:
        return y
```

Here, multiple **`ensures`** clauses are given which are conjoined to form the function's postcondition. We find that allowing multiple **`ensures`** clauses helps readability, and note that JML [11], Spec# [4] and Dafny [5] also permit this. Furthermore, multiple **`requires`** clauses are permitted in the same manner.

### 2.3 Example 2 — Data Type Invariants

Type invariants over data can also be explicitly defined:

```
// A natural number is an integer greater-than-or-equal-to zero
type nat is (int n) where n >= 0
// A positive number is an integer greater-than zero
type pos is (int p) where p > 0
```

Here, the **`type`** declaration includes a **`where`** clause constraining the permitted values. The declared variable (e. g., `n` or `p`) represents an arbitrary value of the given type. Thus, `nat` defines the type of natural numbers. Likewise, `pos` gives the type of positive integers. Constrained types are helpful for ensuring specifications remain as readable as possible. For example, we can update `decrement()` as follows:

```
function decrement(pos x) -> (nat n):
    //
    return x - 1
```

Types in Whiley are more fluid than in typical languages as variables can move seamlessly between them. If two types $\texttt{T}_1$ and $\texttt{T}_2$ have the same *underlying* type, then $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ iff the constraint on $\texttt{T}_1$ implies that of $\texttt{T}_2$.

### 2.4 Example 3 — Loop Invariants

Whiley supports loop invariants which are necessary for proving properties about loops. The following illustrates:

```
function sum(int[] xs) -> (int r)
// Every item in xs is greater or equal to zero
requires all { i in 0..|xs| | xs[i] >= 0 }
// Return must be greater or equal to zero
ensures r >= 0:
    //
    int s = 0
    int i = 0
    while i < |xs| where s >= 0 && i >= 0:
        s = s + xs[i]
        i = i + 1
    return s
```

Here, a bounded quantifier enforces that `sum()` accepts an array of natural numbers (which could equally have been expressed as type `nat[]`). A key constraint is that summing an array of natural numbers yields a natural number (recall arithmetic is unbounded and does not overflow). The Whiley compiler statically verifies that `sum()` does meets its specification. The loop invariant is necessary to help the compiler generate a sufficiently powerful verification condition to prove the function meets the post condition.

### 2.5 Example 4 — Properties

Properties are provided as a convenient mechanism for expressing common parts of a specification/invariant:

```
property contains(int[] xs, int x, int n)
where some { k in 0..n | xs[k] == x }
```

Here, the property `contains` captures the notion that an item is contained in an array. Properties provide a mechanism for defining the "language" in which a function's specification can be written.[1]

[1] One may wonder why properties are needed at all. However, properties have special treatment within the verifier, compared with functions which are "uninterpreted".

## 3. Case Studies

We now examine array programming in the context of a language supporting specification and invariants. Whilst our presentation does focus on the use of Whiley, it should be noted that it applies equally to other similar languages (e.g. Dafny [5], Spec# [4], etc).

### 3.1 Maximum Element

Our first example is the well-known problem of finding the maximum element of an array. This is an oft-used example in the context of verification [12, 13]. The problem definition is simple:

> *"Given an array, return the largest value contained therein."*

To begin the process, we must first state this more precisely as a specification. To do this, we define the concept of *"largest"* as follows:

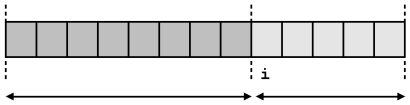
```
property largest(int[] xs, int x, int n)
where all { k in 0..n | xs[k] <= x }
```

This states that no element in the array from zero upto (but not including) `n` is larger than `x` (the reason for including the upper bound `n` will be apparent shortly). This is not itself sufficient to specify our problem but, using `contains()` from above, we can now do so:

```
function max(int[] items) -> (int r)
// Input array must have at least one item
requires |items| > 0
// Item returned must be largest of any in array
ensures largest(items,r,|items|)
// Item returned must be contained in array
ensures contains(items,r,|items|):
```

We can immediately see how the language facilitates a more expressive description of the function, compared with mainstream languages. In particular, the use of properties and quantification over arrays is critical here.

At this point, it remains to implement `max()` and we simply take the "most obvious" approach. Key to this is a loop invariant which, at each point, maintains the largest value seen. This can be viewed diagrammatically as follows:




This represents our implementation operating on a given array which, at any given point, has determined the maximum element (as determined by our specification) for all elements upto (but not including) `i`.

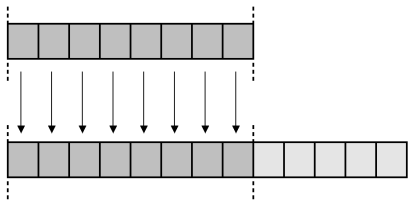
Our implementation of the `max()` function is as follows (where the specification is omitted as it matches above):

```
function max(int[] items) -> (int r)
...
    nat i = 1
    int m = items[0]
    //
    while i < |items|
    where i <= |items|
    where largest(items,m,i)
    where contains(items,m,i):
        if items[i] > m:
            m = items[i]
        i = i + 1
    //
    return m
```

The reason that both `contains()` and `largest()` were defined with an upper bound `n` is now apparent. Doing so allows us to reuse them for both specification and loop invariant. The latter is necessary for the Whiley compiler to statically verify that this function meets its specification.

### 3.2 Resize

Our next example is a simple function for resizing an array. If the array size is reduced, all items up to that point are unchanged. If the array size increases, the new portion of the array is filled with a default. The following illustrates:



To give the specification for `resize()`, we define the notion of "unchanged" as follows:

```
property unchanged(int[] xs,int n,int[] ys)
// All elements upto n unchanged
where all { j in 0..|xs| |
    j < n ==> xs[j] == ys[j]
}
```

This states that everything up to a given size `n` is the same between `xs` and `ys`. Using this, we can specify `resize()`:

```
function resize(int[] xs, nat n, int v)
                                -> (int[] ys)
// Returned array is of specified size n
ensures |ys| == n
// All elements retained from old array (up to new n)
ensures unchanged(xs,n,ys)
// All new elements match default value
ensures all {i in |xs|..n | ys[i] == v}:
```

The specification restricts the possible implementations considerably but still leaves open, for example, the order of

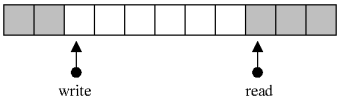
iteration through the arrays. As before, our implementation takes the straightforward approach:

```
function resize(int[] xs, nat n, int v)
...
    int[] rs = [v; n]
    nat i = 0
    while i < n && i < |xs|
    // Ensure array size remains unchanged
    where |rs| == n
    // All elements up to i match as before
    where unchanged(xs,i,rs)
    // All elements about n match element
    where all {j in |xs|..n | rs[j] == v}:
        //
        rs[i] = xs[i]
        i = i + 1
    //
    return rs
```

One curious aspect of our implementation is the loop invariant clause "`|rs| == n`". This clause seems unnecessary as no statement in the loop affects the size of `rs`. The need for it is an artefact of the verification system (which is based on Hoare logic). Specifically, within the body of a loop all knowledge about variables modified in that loop is lost, except that given in the loop invariant and condition.

### 3.3 Cyclic Buffer

A *cyclic buffer* consists of a fixed-sized array with *read* and *write* pointers that "wrap around". The key is that we should be able to write when the buffer is not full, and read when it is not empty. The following gives a diagrammatic view:




Here, we see a buffer holding five items (marked in gray) where `write` has wrapped around and is below `read`. We can also see that `write` identifies the first *unused* space, whilst `read` identifies the first *used* space (if one exists). The buffer is *empty* when "`read == write`" and *full* when there is exactly one unused space (i.e. roughly when "`write+1 == read`"). We can encode the general concept of a cyclic buffer as follows:

```
type Buffer is {
    int[] data,
    nat read,    // read pointer
    nat write    // write pointer
}
//
// Read / write pointers within bounds
```

Our definition above is not very restrictive and simply requires both pointers are within bounds. We can now easily define the notion of an empty buffer:

```
// same position.
type EmptyBuffer is (Buffer b)
```

The type `EmptyBuffer` further constrains `Buffer` to the case where both pointers are the same. Using this we can give a suitable constructor for `Buffer`:

```
// Create a buffer with a given number of slots.
function Buffer(int n) -> EmptyBuffer
// Cannot create buffer with zero size!
requires n > 0:
   //
```

Here, "`[v;n]`" generates an array of size `n` with each element initialised to `v`. We see a precondition is necessary on the constructor to ensure "`n > 0`" as, otherwise, one cannot construct a valid buffer (i.e. since this requires e.g. "`0 <= read < n`"). We now proceed to define the concept of a "non-full" buffer as follows:

```
// NonFull buffer has at least one writeable space.
type NonFull is (Buffer b)
// Write cannot be immediately behind read
```

This simply states that `write` cannot be one space behind `read` (accounting for wrap around). Using this definition, we can give the function for writing into the buffer:

```
// Write an item into a buffer which is not full
function write(NonFull b,int v)->(Buffer r)
// All items unchanged except at (old) write position
ensures unchanged(b.data, b.write, r.data)
// Item v now at old write pos
ensures r.data[b.write] == v
// Read pointer is unchanged
ensures b.read == r.read
// Write pointer has advanced one position
ensures (b.write+1) % |b.data| == r.write:
    //
    b.data[b.write] = v
    b.write = (b.write + 1) % |b.data|
```

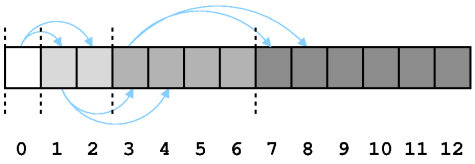
The post-condition here is perhaps surprisingly involved and, indeed, is longer than the implementation itself! In fact, it provides a fairly exact description of the what the function does (though, in general, we find that specifications leave some "wriggle" room for the implementation). The property `unchanged()` simply states that everything in the array is unchanged, except for the item being written:

```
property unchanged(int[] xs,int i,int[] ys)
// Size of arrays remains unchanged
where |xs| == |ys|
// All elements except i are unchanged
where all {
 j in 0..|xs| | (j != i) ==> (xs[j]==ys[j])
```

At this point, we leave the remainder of the implementation to the reader's imagination and simply remark that it follows from the above as expected.

### 3.4 Binary Heap

The binary heap is a common data-structure for implementing a priority queue. From the perspective of array programming, it is convenient as it implemented using an array with relative ease. Diagramatically, we can view it as thus:




The key is that, for a given node at index `i`, the position of its children is calculated as "`(2*i)+1`" and "`(2*i)+2`". Furthermore, the value of each element is larger (i.e. has higher priority) than either of its children. We can formalise this quite nicely as follows:

```
property valid(int[] H, int n, int i,int c)
// For valid child, value is less than parent
where (2*i)+c < n ==> H[(2*i)+c] < H[i]

property validChildren(int[] H, int n)
// Items on left branch are below their parent's item
where all{ i in 0..n | valid(H,n,i,1)}
// Items on right branch are below their parent's item
where all{ i in 0..n | valid(H,n,i,2)}
```

Property `validChildren()` gives the invariant which holds for any valid heap where `n` represents the number of items currently in the heap (and for which we will additionally require `n <= |H|`). Using this property we can define a suitable data type for representing heaps:

```
type Heap is {int[] data, nat length}
// All children within the heap are valid
where validChildren(data,length)
// Never more allocated items than space
where length <= |data|
```

Thus, a `Heap` defines a region of `data` within which all items are stored and which limits the heap's maximum possible size. At any given moment *some*, *all* or *none* of the items in that region maybe contained in the heap proper (as determined by `length`).

For completness, we now consider one operation for manipulating heaps. Figure 1 gives the `insert()` function. This puts the item `v` being inserted into the next available slot, which may break the heap invariant (i.e. because `v` is larger than its parent). To restore the invariant, the function proceeds to swap `v` up the tree until it find the correct spot.

One interesting aspect of `insert()` is the use of variable "`oh`". This is commonly referred to as a *ghost variable*,

```
function insert(Heap h, int v) -> (Heap r)
// Heap cannot be full
requires h.length < |h.data|
// Item v is added to heap
ensures contains(r.data,v,r.length)
// All items in heap remain in heap
ensures containsAll(r.data,h.data)
// All items now in heap were in heap (except perhaps v)
ensures containsEx(h.data,r.data,v):
    //
    nat i = h.length
    // Add v to end of heap
    h.data[i] = v
    // Create (ghost) copy of heap
    Heap oh = h
    //
    int parent = (i-1)/2
    // Swap v up to restore invariant
    while parent > 0 && h.data[parent] < v
    where parent < h.length
    where contains(h.data,v,|h.data|)
    where containsAll(h.data,oh.data)
    where containsEx(oh.data,h.data,v):
        // perform a swap
        h.data[i] = h.data[parent]
        h.data[parent] = v
        // update indices
        i = parent
        parent = (i-1)/2
    //
    return h

property containsAll(int[] a, int[] b)
// Every item in b is in a
where all { i in 0..|b| |
    contains(a,b[i],|b|)
}
property containsEx(int[] a,int[] b,int v)
// Every item in b is in a, except perhaps v
where all { i in 0..|b| |
    b[i] == v || contains(a,b[i],|b|)
}
```

**Figure 1.** The function for inserting an item into a heap

since its only purpose is to aid verification [3, 14, 15]. Using this the loop invariant can refer to `h` as it was before the loop. Finally, `insert()` also employs two properties which extend the property `contains()` from before.

## 4. Discussion

The above case studies highlight how the use of specifications gives life to simple array-based implementations. That is, the ability to express invariants explicitly can offset the "low-level" nature of the array implementation. Such invariants are typically otherwise hidden (though one can opti-

mistically hope they are at least mentioned in the docs). We now provide some further commentary arising from this.

***Properties and Type Invariants.*** Our case studies make extensive use of properties to ensure specifications are as readable as possible. Typically, we also use them as the language for defining the problem. We can observe some reuse of properties between examples. For example, `contains()` is used in the `max()` and binary heap examples. Likewise, a similar notion of `unchanged()` is used in `resize()` and the cyclic buffer. We imagine common properties will be combined into libraries for describing related problems. And, most likely, that a core set of properties will emerge as a primary language for describing array problems (which, ideally, would be incorporated into the standard library).

Type invariants are another mechanism we employ to help characterise our problems. In the cyclic buffer problem, defining different states of the buffer (e.g. `EmptyBuffer`, `NonFull`, etc) proved useful. We expect type invariants to be reused less than properties as, by their nature, they are somehow more "concrete" (i.e. they range over fixed types).

***Other Tools.*** Whilst this paper focuses primarily on Whiley, the argument put forward (i.e. that specifications/invariants enhance array programming) extends to other similar tools. We now consider one such tool, namely Dafny, which is quite comparable to Whiley [5, 14]. Dafny is an imperative language with support for objects and classes without inheritance. Like Whiley, Dafny employs unbound arithmetic, pure functions and provides immutable collection types with value semantics. Dafny also employs *dynamic frames* [16] as a simple mechanism for reasoning about pointer-based programs. Finally, Dafny has been used successfully in many verification challenges [13, 17–20].

Figure 2 provides a Dafny implementation of the `max()` function from §3.1. Generally speaking, this is largely similar to the Whiley implementation. However, there are some differences between the two. For example, Dafny treats arrays as references which can be **`null`** and, hence, requires it be established they are *not*. Likewise, **`reads`** clauses are required to declare any reference variables which may be accessed within a method. Finally, predicate methods (analogous to properties in Whiley) must declare preconditions to ensure, for example, arrays are not null and array accesses are within bounds.

***Loop Invariants.*** The need for loop invariants remains something of a barrier-to-adoption for tools like Whiley and Dafny. Flanagan and Qadeer commented that "*While method specifications also function as useful documentation and may be helpful for code maintenance, loop invariants do not provide comparable benefits*" [21]. Their experiences using ESC/Java lead them believe that "*the burden of specifying loop invariants is substantial*". Likewise, Beckert *et al.* note that, in general, "*loop invariants are polluted by formulas stating what the loop does not do.*" [22]. Numerous techniques exist for automatically inferring loop invariants and we hope such work will ease the burden [23–27].

```
predicate method
largest(xs: array<int>, x: int, n: int)
requires xs != null && n <= xs.Length;
reads xs;{
 (forall k :: 0 <= k < n ==> xs[k] <= x)
}
predicate method
contains(xs: array<int>, x: int, n: int)
requires xs != null && n <= xs.Length;
reads xs;{
 (exists k :: 0 <= k < n && xs[k] == x)
}

method max(items:array<int>) returns(r:int)
// Input array must have at least one item
requires items != null && items.Length > 0;
// Item returned must be largest of any in array
ensures largest(items,r,items.Length);
// Item returned must be contained in array
ensures contains(items,r,items.Length);
{
  r := items[0];
  var i := 1;

  while i < items.Length
  invariant i <= items.Length;
  invariant largest(items,r,i);
  invariant contains(items,r,i);
  {
    if items[i] > r { r := items[i]; }
    i := i + 1;
  }
}
```

**Figure 2.** Dafny implementation of the `max()` function from §3.1.

## 5. Related Work

***Verification.*** Small array-oriented programs have long been used in the context of verification. Dijkstra's *Dutch National Flag* problem, although somewhat contrived, is perhaps one of the most widely used for introducing loop invariants [28]. The essential problem is to sort an array of three colours (i.e. those making up the Dutch National Flag) into the correct order. The solution requires only a single loop with a rather nice loop invariant. The introductory books of Broda *et al.* [29], Kourie and Watson [12] and Backhouse [30] all provide worked solutions to this problem to illustrate loop invariants, whilst Gries leaves it as an exercise for the reader [31].

The *Correctness-by-Construction (CbC)* approach pioneered by Dijkstra, Hoare and others promotes the develop-

ment of programs by progressive refinement from specifications [32, 33]. Kourie and Watson applied this to a range of simple array-based programs, such as finding the maximum element of an array, performing a linear search through an array, finding the longest matching array segment, etc [12]. Dony and Le Charlier developed a tool for teaching students the CbC method [34]. Their stated ambition was to restrict themselves to *"a very simple programming language with simple types (limited to finite domains) and arrays, and to address (relatively) small examples such as searching and sorting algorithms"*. This, they claimed, allowed problems to be specified clearly and formally.

In works on verification, especially those related to teaching, the presentation of loop invariants almost always coincides with the use of array-based examples (which is perhaps not entirely surprising). Back emphasises the value of *"drawing good figures to illustrate the way the algorithm is intended to work"* where his examples primarily correspond with array diagrams similar to those seen in this (and many other) papers [35]. He goes on to claim clear benefit from focusing on *"well understood application domains (mostly array manipulation programs)"* when teaching. In a similar vein, Astrachan argues that *"invariants are especially useful in introductory courses"* but are often avoided because of the mathematical notation involved [36]. Instead, a preference in this context is for the use of invariant diagrams which, again, are typically (though not exclusively) over arrays.

Finally, we observe that a large portion of the problems set in verification challenges are array problems of the kind illustrated here. For example, in VSCOMP'10 three out of five were array problems (with the remainder on linked lists) [18]. Likewise, in COST'11 two out of three were array problems (with the others on trees) [13]. However, in VerifyThis'16 only one out of three was an array problem [20].

***Languages.*** Fortran, for a long time, has been the undeniable king of high-performance computing in part, at least, because of its strong emphasis on arrays. Early success with vectorising compilers for Fortran77 lead to numerous efforts to support parallelisation across different architectures, such as the High-Performance Fortran (HPF) standard [37], Vienna Fortran [38] and Co-Array Fortran [39]. Fortran arrays are declared via the `DIMENSION` attribute which allows one to specify the *rank* (number of dimensions), *shape* (size of each dimension) and *extent* (ranges for each dimension).

Pascal, despite the lack of dynamic arrays (described as a "defect" by Wirth [40]), also has surprisingly expressive support for static arrays. In particular, since array types include the exact range of permitted indices, interesting subtype relationships can be enforced. For example, a value of type "`ARRAY[1..100] of T`" is not permitted to flow into a location of type "`ARRAY[0..99] of T`". Indeed, any ordinal type can be used to restrict the range of permitted indices. For example, a variable of type "`ARRAY[`**`byte`**`] of T`" has an index for every possible **`byte`** value.

The Ada programming language provides a similar mechanism to Pascal for defining and accessing bounded arrays, though additionally supports explicit subtypes [41]. The SPARK/Ada subset [8] also provides syntax for expressing specifications and invariants (and support for static checking), some of which migrated into Ada 2012 [42].

The X10 language from IBM was designed from the beginning with an aim to *"include a rich array sub-language that supports dense and sparse distributed multi-dimensional arrays"* for the purposes of high performance computing [43]. However, the language designers struggled to achieve *"acceptable levels of performance for a single unified array sub-language"* [44]. To resolve this, they instead identified a set of core mechanisms which allowed powerful array primitives to be provided via libraries. X10 also provided, like Whiley, support for constrained types [45]. This meant their libraries, for example, could *"use constrained types to enforce that the number of dimensions used in indexing operations ... matches the rank"* [44].

The Java-based language Lime, also from IBM, bears some similarity to X10, though it focused on exploiting GPU and FPGAs as accelerators [46]. Key differences over Java include the introduction of *deeply immutable* arrays and also *bounded* (i.e. fixed-sized) arrays. The latter plays an important role in Lime for expressing inputs and outputs for stream operators. Furthermore, bounded arrays are always accessed by index expressions which are statically guaranteed to be within bounds which, like Pascal, is achieved through ordinal ranges.

## 6. Conclusion

Through a series of short case studies, we have explored how specifications and invariants enhance array programming. We believe much is gained from expressing specifications and invariants as these are typically otherwise hidden. This is especially true for array programs which often involve subtle and complex invariants. We also find the specification languages used in systems like Whiley and Dafny are ideally suited for describing array problems. Of course, their real power comes from statically verifying programs meet their specifications, and we hope further advances will see them used more widely.

## References


[1] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *JACM*, 52(3):365–473, 2005.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[3] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[4] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *CACM*, 54(6):81–91, 2011.

[5] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

[6] J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proc. ESOP*, pages 125–128, 2013.

[7] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proc. NFM*, pages 41–55. Springer-Verlag, 2011.

[8] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., Reading, 1997.

[9] D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *SCP*, pages 191–220, 2015.

[10] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proc. VMCAI*, pages 41–62, 2016.

[11] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proc. CASSIS*, pages 108–128, 2005.

[12] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[13] T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The COST IC0701 verification competition 2011. In *Proc. FoVeOOS*, pages 3–21, 2011.

[14] K. R. M. Leino. Developing verified programs with Dafny. In *Proc. VSTTE*, pages 82–82, 2012.

[15] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. TPHOL*, pages 23–42, 2009.

[16] I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proc. FM*, pages 268–283, 2006.

[17] K.R.M. Leino and R Monahan. Dafny meets the verification benchmarks challenge. In *Proc. VSTTE*, pages 112–126, 2010.

[18] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K.R.M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st verified software competition: Experience report (VSComp). In *Proc. FM*, 2011.

[19] M. Huisman, V. Klebanov, and R. Monahan. Verifythis verification competition 2012 - organizer's report, 2013.

[20] M. Huisman, R. Monahan, P. Müller, and E. Poll. Verifythis 2016 — a program verification competition. *STTT*, 2016.

[21] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, pages 191–202, 2002.

[22] B. Beckert, S. Schlager, and P. Schmitt. An improved rule for while loops in deductive program verification. In *Proc. ICFEM*, pages 315–329, 2005.

[23] A. Ireland, B. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In *Proc. MICAI*, pages 190–201, 2004.

[24] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proc. APLAS*, pages 119–134, 2005.

[25] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *CoRR*, abs/0909.0884, 2009.

[26] M. Aponte, P. Courtieu, Y. Moy, and M. Sango. Maximal and compositional pattern-based loop invariants. In *Proc. FM*, pages 37–51, 2012.

[27] D. Cachera, T. P. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In *Proc. SAS*, pages 58–74, 2012.

[28] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[29] K. Broda, S. Eisenbach, H. Khoshnevisan, and Steven Vickers. *Reasoned Programming*. Prentice Hall, 1994.

[30] Roland Backhouse. *Program Construction*. Wiley, 2003.

[31] D. Gries. *The science of programming*. Springer-Verlag, 1981.

[32] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[33] C. A. R. Hoare. Proof of a program: FIND. *CACM*, 14:39–45, 1971.

[34] I. Dony and B. Le Charlier. A tool for helping teach a programming method. In *Proc. SIGCSE*, pages 212–216, 2006.

[35] R. Back. Invariant based programming: basic approach and teaching experiences. *FAC*, 21(3):227–244, 2009.

[36] Owen Astrachan. Pictures as invariants. In *Proc. SIGCSE*, pages 112–118. ACM, 1991.

[37] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran. *CACM*, 54(11):74–82, 2011.

[38] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in vienna fortran. *SCP*, 1(1):31–50, 1992.

[39] R. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[40] N. Wirth. Recollections about the development of pascal. In *Proc. HOPL*, pages 333–342. ACM Press, 1993.

[41] J. Barnes. *Programming in Ada 2012*. Addison Wesley, 2014.

[42] R. Dewar. Ada2012: Ada with contracts. *Dr. Dobbs's Journal*, 2013.

[43] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA*, pages 519–538, 2005.

[44] D. Grove, J. Milthorpe, and O. Tardieu. Supporting array programming in X10. In *ARRAY*, pages 38–43, 2014.

[45] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Proc. OOPSLA*, pages 457–474, 2008.

[46] J. Auerbach, D. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proc. OOPSLA*, pages 89–108, 2010.