Journal of Automated Reasoning (2022) 66:747–803
https://doi.org/10.1007/s10817-022-09619-1

Verifying Whiley Programs with Boogie

David J. Pearce[1] · Mark Utting[2] · Lindsay Groves[1]

Received: 22 April 2020 / Accepted: 14 January 2022 / Published online: 20 March 2022

Abstract

The quest to develop increasingly sophisticated verification systems continues unabated. Tools such as Dafny, Spec#, ESC/Java, SPARK Ada and Whiley attempt to seamlessly integrate specification and verification into a programming language, in a similar way to type checking. A common integration approach is to generate verification conditions that are handed off to an automated theorem prover. This provides a nice separation of concerns and allows different theorem provers to be used interchangeably. However, generating verification conditions is still a difficult undertaking and the use of more "high-level" intermediate verification languages has become commonplace. In particular, Boogie provides a widely used and understood intermediate verification language. A common difficulty is the potential for an impedance mismatch between the source language and the intermediate verification language. In this paper, we explore the use of Boogie as an intermediate verification language for verifying programs in Whiley. This is noteworthy because the Whiley language has (amongst other things) a rich type system with considerable potential for an impedance mismatch. We provide a comprehensive account of translating Whiley to Boogie which demonstrates that it is possible to model most aspects of the Whiley language. Key challenges posed by the Whiley language included: the encoding of Whiley's expressive type system and support for flow typing and generics; the implicit assumption that expressions in specifications are well defined; the ability to invoke methods from within expressions; the ability to return multiple values from a function or method; the presence of unrestricted lambda functions; and the limited syntax for framing. We demonstrate that the resulting verification tool can verify significantly more programs than the native Whiley verifier which was custom-built for Whiley verification. Furthermore, our work provides evidence that Boogie is (for the most part) sufficiently general to act as an intermediate language for a wide range of source languages.

✉ Mark Utting
m.utting@uq.edu.au

David J. Pearce
david.pearce@ecs.vuw.ac.nz

Lindsay Groves
lindsay@ecs.vuw.ac.nz

[1] Victoria University of Wellington, Wellington, New Zealand

[2] The University of Queensland, Brisbane, Australia

Keywords Whiley · Boogie · Verifying compiler · Intermediate verification language · Semantic translation · Impedance mismatch · Flow typing · Verification conditions

1 Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been studied for a long time. Part of the appeal of software verification is that it can ensure theoretical correctness of a software module for all possible usages. This is complementary to testing which, by acting at a more concrete level, may detect resource or hardware errors that are typically outside the scope of software verification [43].

According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [80]. A variety of tools have blossomed in this space, including Spec# [16], Dafny [104], Why3 [65], OpenJML [46], ESC/Java [68], VeriFast [85], SPARK/Ada [121], AutoProof for Eiffel [163], Frama-C [52], KeY [2], SPARK/Ada [13, 40] and Whiley [139, 159]. Automated theorem provers are integral to such tools and are responsible for discharging proof obligations [16, 45, 68, 85]. Various satisfiability modulo theory (SMT) solvers are typically used for this, such as Z3 [53], CVC4 [19, 20], Yices2 [61], Alt-Ergo [49], Vampire [82, 95] or Simplify [55]. These provide handcrafted implementations of important decision procedures, e.g. for linear and nonlinear arithmetic [23, 44, 62, 144], congruence [126, 128] and quantifier instantiation [54, 71, 145, 146]. Different solvers are appropriate for different tasks, so the ability to utilise multiple solvers can improve the chances of successful verification.

Verifying compilers often target an *intermediate verification language*, such as Boogie [14], WhyML [28, 65] or Viper [125], as these provide a nice separation of concerns and allow different theorem provers to be used interchangeably. SMT-LIB [21] provides another standard readily accepted by modern automated theorem provers, although it is often considered rather low level [28]. One issue faced by intermediate verification languages is the potential for an *impedance mismatch* [139] (see Sect. 5). This arises when constructs in the source language cannot be easily translated into those of the intermediate verification language (and vice versa).

Whiley is a programming language with first-class support for software specifications that is designed to simplify verification [43, 134, 135, 137–140, 159, 168, 169]. An important goal was to develop a system which is as accessible as possible and which one could imagine being used in a day-to-day setting. As such, Whiley superficially resembles a modern imperative language and employs *flow typing* [77, 133, 160] to eliminate unnecessary casts (which also aids specification). The ultimate aim is that all programs written in Whiley will be verified at compile time to ensure their specifications hold which, for example, has obvious application in safety-critical systems [40, 134]. In this paper, we explore Boogie as an intermediate verification language for Whiley. Our motivation is the desire to improve the verification capability of Whiley by leveraging the significant resources already invested in the development of Boogie (and Z3). A particular concern is the potential for an impedance mismatch arising, such as from Whiley's type system (e.g. which supports union types and flow typing).

The contributions of this paper include:

- **(Translation)** A comprehensive account of our encoding of Whiley programs into Boogie for the purpose of verification. Whilst in many cases the translation is straightforward, a number of challenges had to be overcome arising from Whiley's design, including: the

encoding of Whiley's expressive type system and support for flow typing and generics; Whiley's implicit assumption that expressions in specifications are well defined; the ability to invoke methods from within expressions; the ability to return multiple values from a function or method; the presence of unrestricted lambda functions; and Whiley's limited syntax for framing.
- **(Evaluation)** An empirical comparison between Boogie/Z3 and the native Whiley verifier using the existing suite of 1100+ tests provided for the Whiley compiler. The results confirm that Boogie/Z3 significantly outperforms the Whiley native verifier in terms of the number of tests passing.
- **(Case Studies)** A report into the use of Boogie/Z3 to verify a number of larger Whiley programs, including a web-based implementation of Conway's Game of Life and a number of challenges from the VerifyThis 2019 competition [59]. From these case studies we identify several areas in which the Whiley language or libraries could be improved to better exploit Boogie.

We note also that our work provides further evidence of Boogie's utility as a general-purpose intermediate verification language. In particular, compared with Dafny or Spec#, Whiley was developed entirely independently from Boogie and includes various design choices that are not necessarily a natural fit. As such, it was unclear from the outset of this project whether or not Boogie would be sufficiently general for this task. Finally, compared with our earlier paper [164], this paper represents a significant evolution and improvement of our translation. We also provide a much more detailed account which covers almost the entire language, including generics, lambdas, references and the handling of various soundness issues. Our evaluation now includes a number of larger case studies, and we have expanded the related work discussion.

Organisation. The remainder of this paper is organised as follows: Sect. 2 provides an introduction to Whiley and Boogie; Sect. 3 provides a detailed description of our Whiley-to-Boogie translator and discusses the various challenges encountered; Sect. 4 presents our evaluation using the existing Whiley compiler test suite and various case studies; Sect. 5 examines the related work; and finally, Sect. 6 concludes. Finally, for reference, "Appendix A" illustrates our verified version of Conway's Game of Life.

2 Background

We begin with an overview of Whiley and then a brief discussion of Boogie.

2.1 Whiley

The Whiley programming language has been developed to enable compile time verification of programs and, furthermore, to make this accessible to everyday programmers [139, 159]. The Whiley Compiler (WyC) attempts to ensure that all functions and methods in a program meet their specifications. When this succeeds, we know that: (i) all function/method postconditions are met (assuming their preconditions held on entry); (ii) all invocations meet the respective function or method precondition; (iii) runtime errors such as divide-by-zero, out-of-bounds accesses and null-pointer dereferences cannot occur. Notwithstanding, such programs may still loop indefinitely and/or exhaust available resources (e.g. stack or heap).

2.1.1 Primitive Types

Whiley provides a small number of primitive types, including: `null`, `bool`, `byte` and `int` (for unbound integers). Likewise, types can be composed into *records* (e.g. `{int x, int y}`), *arrays* (e.g. `int[]`) and *unions* (e.g. `null|int`). Here, the latter represents a type which is either `null` or an `int`. Records can be constructed using *literals* (e.g. `{x:1,y:2}`, whilst arrays can be constructed using either *literals* (e.g. `[1,2,3]`) or *generators* (e.g. `[0;3]` which gives `[0,0,0]`). The length of an array can also be queried dynamically (e.g. `|xs|`). As expected, *user-defined types* are supported and can be declared as follows:

```
type Point is { int x, int y }
```

Whiley also supports *type polymorphism* (i.e. generics) and *recursive types* (which are similar to algebraic data types) as follows:

```
type Node<T> is { T data, List<T> next }
type List<T> is null | Node<T>
```

The type `{T data, List<T> next}` indicates a record with two fields, `data` and `next`. Thus, a `List<T>` is either `null` or a record with the given structure. For completeness, we note that subtyping of generic types follows an (implicit) definition-site variance protocol [3]. Furthermore, user-defined types in Whiley offer greater flexibility than typically found with implementations of algebraic data types (e.g. in Haskell). For example:

```
type IntList is null | { int data, IntList next }

function id(IntList l) -> (List<int> r):
   return l
```

The above illustrates how one recursive type (`IntList`) can implicitly subtype another (`List<int>`). This highlights a key advantage of typing in Whiley over, for example, algebraic data types. The approach to typing taken in Whiley is, in fact, closer to *structural typing* [36, 60, 72, 117, 118] with certain caveats to ensure safe treatment of type invariants (see below).

2.1.2 Flow Typing

An unusual feature of Whiley is the use of a *flow typing system* [77, 132, 133, 160] coupled with *union types* [12, 84]. Union types support *runtime type tests* to discriminate their cases, as the following illustrates (recall `List<T>` from above):

```
function length<T>(List<T> list) -> int:
    if list is null:
        return 0
    else:
        return 1 + length(list.next)
```

This counts the number of nodes in a list. Here, we see flow typing in action as `list` is automatically retyped to `Node<T>` on the false branch [132, 133]. Flow typing turns out to be particularly useful when specifying programs. Specifically, in `(x is T) ==> e` it follows that `x` has type `T` within the expression `e`. This helps, for example, when writing postconditions (as we'll see shortly).

2.1.3 Value Semantics

The semantics of Whiley diverge from many mainstream languages (e.g. Java) in the treatment of compound data types, such as arrays. Specifically, arrays and records in Whiley have *value semantics*. This means they are passed and returned by value (as in Pascal, MATLAB [98] or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place [129, 151]. This latter point serves to give Whiley the appearance of an imperative language when, in fact, Whiley has a functional core. The following illustrates:

```
function fill<T>(T[] items, int n, T v) -> (T[] nitems):
  for i in 0..n:
    items[i] = v
  return items
```

Despite appearances, the above is a *pure* function which has no side effects. This contrasts with languages like Java, where arrays are references and updating them has unavoidable side effects. The following attempts to clarify this further:

```
  int[] xs = [1,2,3]
  int[] ys = xs
  ys[0] = 0
  assert xs[0] == 1
  assert ys[0] == 0
```

In a language like Java, the assertion `xs[0] == 1` would fail because `xs` and `ys` would alias each other. However, since this is not the case in Whiley, the above verifies without problem. We can think of arrays and records in Whiley as being *immutable*, so that updating them effectively means cloning them. The reason this semantics is adopted in Whiley is to facilitate their use in specification. Indeed, without a fundamental immutable collection type, verification is inherently challenging [99].

2.1.4 Side Effects

A `function` in Whiley is pure and cannot have side effects. In contrast, a `method` is impure and may have side effects, such as mutating the global heap or performing I/O. Whiley provides reference types which are allocated from a single global heap. For example, `&int` is a reference to an integer variable. The following illustrates the syntax:

```
  &int p = new 1
  &int q = p
  *p = 2
  assert *p == *q
```

Here, the assignment through `p` affects `q` (because they are aliases), and hence, the final assertion holds. We note that, at the time of writing, Whiley supports allocation but not deallocation (and, hence, currently relies on garbage collection).

Statements which mutate the heap must appear within the body of a `method` and, for example, are not permitted within a `function`. To illustrate a more complete example, here is the classical algorithm for reversing a linked list [6]:

```
type LinkedList<T> is null | &{T data, LinkedList<T> next}

method reverse<T>(LinkedList<T> v) -> (LinkedList<T> r):
    //
    LinkedList<T> w = null
    //
    while !(v is null):
        LinkedList<T> t = v->next
        v->next = w
        w = v
        v = t
    //
    return w
```

We note that the above is not yet fully specified, and this would be necessary before its behaviour could be fully verified (more on this later).

2.1.5 Packaging

Whiley currently supports a relatively limited form of packages and package management. For example, the standard library, `STD.wy`, can be added as a dependency and compiled against. The following illustrates a simple example:

```
import std::ascii
import append from std::array

function to_string(int[] items) -> (ascii::string str):
    ascii::string r = "["
    // Convert each element to an ascii string
    for i in 0..|items|:
        // Add comma (when necessary)
        if i != 0:
          r = append(r,",")
        // Add element as string
        r = append(r,ascii::to_string(items[i]))
    return append(r,"]")
```

The above illustrates a simple function for converting an integer array into a string. This employs standard library functions from the modules `std::ascii` and `std::array`.

```
type nat is (int n) where n >= 0

property contains<T>(T[] items, int n, T item)
where some { i in 0..n | items[i] == item }

function indexOf<T>(T[] items, T item) -> (int|null r)
// If valid index returned, element at r matches item
ensures (r is int) ==> (items[r] == item)
// If invalid index return, no element matches item
ensures (r is null) ==> !contains(items,|items|,item):
    //
    nat i = 0
    while i < |items|
    // Nothing so far equals item
    where !contains(items,i,item):
        //
        if items[i] == item:
            return i
        i = i + 1
    // Sanity check?
    assert i == |items|
    //
    return null
```

Fig. 1 Implementation of `indexOf()` in Whiley, returning the least index in `items` which matches `item`, or `null` if no match exists

2.1.6 Specification and Verification

We now consider those features of Whiley provided for specifying and verifying programs. Figure 1 provides an initial example to illustrate the salient features:

- **Properties** are used to specify things of interest, particularly to help with verification. They are *interpreted* meaning that, during verification, they can be expanded/unrolled as necessary. To facilitate this, they have a restricted form allowing them to be substituted in place for their body. In contrast, **functions** are *uninterpreted* which helps ensure verification remains (mostly) modular [78]. This means that, during verification, their actual implementation is ignored at call sites (more on this below).
- **Preconditions** are given by `requires` clauses and **postconditions** by `ensures` clauses. Multiple clauses are simply conjoined together. We have found that allowing multiple `requires` and/or `ensures` clauses can help readability, and note that JML [48], Spec# [16] and Dafny [104] also permit this.
- **Loop invariants** are given by `where` clauses. Figure 1 illustrates an inductive loop invariant covering indices from zero to `i` (exclusive). Similarly, **type invariants** arise from `where` clauses. For example, type `nat` has an invariant and is used for variable `i` to avoid the need for a loop invariant of the form `i >= 0`. We consider good use of type invariants as critical to improving the readability of function specifications.
- **Assertions** must be statically checked during verification, thus providing a useful debugging tool. For example, if during verification we are struggling to understand why a given postcondition is not met, assertions can be added to check our beliefs at a given point. In contrast, **assumptions** are not statically checked and, instead, are simply assumed to

hold during verification. As such, they are a useful tool for overriding the verifier in cases where it cannot establish something we know to be true.

- **Flow typing** simplifies postconditions (amongst other things) by ensuring that casts need not be given. For example, without flow typing, the first `ensures` clause from Figure 1 would require a cast for `r` on the right-hand side.

Being uninterpreted means a function's implementation can change arbitrarily without affecting callers *provided it still meets its specification*. However, it also means that functions need to be properly specified before they can be used, which is sometimes problematic (e.g. when several functions are developed in tandem). For example, consider the following:

```
function max(int x, int y) -> (int r):
   if x >= y:
      return x
   else:
      return y
```

Whilst the above function is implemented correctly, it has yet to be specified. Perhaps this has arisen because it is, in fact, part of a larger function being developed:

```
function max(int[] items, int i) -> (int r)
// At least one item must remain
requires 0 <= i && i < |items|
// Return greater than all remaining items
ensures all { k in i .. |items| | items[k] <= r }:
    if (i+1) == |items|:
        return items[i]
    else:
        return max(items[i], max(items,i+1))
```

At this moment, `max(int[],int)` cannot be statically verified because the specification for `max(int,int)` (or lack thereof) yields insufficient information at the call site.

Framing. A related aspect of static verification is the need for clarity around side effects and framing [9, 91, 92, 130, 131, 153]. A key issue is the ability to distinguish the value of state *before* a method call from that *after* it. Languages such as Dafny, JML and Boogie support this by allowing one to refer to the "old" state of a location (i.e. the value it held on entry). For example, in JML writing `\old(*p) < *p` in a method's postcondition indicates the value stored in `*p` is increased by the method. Whiley supports similar syntax as the following illustrates:

```
method swap(&int p, &int q)
ensures *p == old(*q) && *q == old(*p):
    int tmp = *p
    *p = *q
    *q = tmp
```

This simple method swaps the values referred to by `p` and `q`, and to specify it, we had to use the `old()` syntax. With the above specification for `swap(&int,&int)` we can verify, for example, the following snippet:

```
...
&int x = new 2
&int y = new 123
&int z = new 234
swap(x,y)
// Check expected outcome
assert (*x == 123) && (*y == 2)
// Check z unchanged
assert (*z == 234)
```

Here, the first `assert` follows from the specification of `swap(&int,&int)`. In contrast, the second follows because the state referred to by `z` is not reachable from any parameter passed to `swap(&int,&int)` and, hence, could not be modified by it.

2.2 Boogie

Boogie [14] is an intermediate verification language developed by Microsoft Research as part of the Spec# project [16]. Boogie is intended as a back end for other programming language and verification systems [106], and has found use in various tools, such as Dafny [104], VCC [45], and others (e.g. [25]). Boogie is both a specification language (which shares some similarity with Dijkstra's language of guarded commands [57]) and a tool for checking that Boogie "programs" are correct. The original Boogie language was *"somewhat like a high-level assembly language in that the control flow is unstructured but the notions of statically scoped locals and procedural abstraction are retained"* [14]. However, later versions support structured `if` and `while` statements to improve readability. Nevertheless, a non-deterministic `goto` statement is retained for encoding arbitrary control flow, which permits multiple target labels with non-deterministic choice. Boogie provides various primitive types including `bool`, `int` and map types, which can be used to model arrays and records. Concepts such as a "program heap" can also be modelled using a map from references to values.

Boogie supports `function` and `procedure` declarations which have an important distinction. In general, functions are pure and can be used within the Boogie logic, such as in axioms and specifications. In contrast, procedures are potentially impure and are intended to model methods in the source language. A procedure can be given a specification composed of `requires` and `ensures` clauses, and also a `modifies` clause indicating non-local state that can be modified. Most importantly, a procedure can be given an `implementation`, and the tool will attempt to ensure this implementation meets the given specification. The `requires` and `ensures` for procedures demarcate proof obligations, for which Boogie emits verification conditions in first-order logic to be discharged by Z3. In addition, the implementation of a procedure may include `assert` and `assume` statements. The former lead to proof obligations, whilst the latter give properties which the underlying theorem prover can exploit.

To illustrate Boogie, Figure 2 provides an example encoding of the `indexOf()` function into Boogie. Note that the example encodings used in this section are a little different to the more sophisticated encoding used later in the paper. At first glance, it is perhaps surprising how close to an actual programming language Boogie has become. Various features of the language are demonstrated with this example. Firstly, an array length operator is encoded

```
// array length operator
function len(arr:[int]int) returns (r: int);
// no negative length arrays
axiom (forall A:[int]int :: len(A) >= 0);

procedure indexOf(xs: [int]int, x:int) returns (r:int)
ensures r >= 0 && r <= len(xs);
ensures (r < len(xs)) ==> (xs[r] == x);
ensures (forall k:int :: (0<=k && k<r) ==> xs[k]!=x); {
  var i : int;
  i := 0;
  while (i < len(xs))
  invariant i >= 0 && i <= len(xs);
  invariant (forall k:int :: (0<=k && k<i) ==> xs[k] != x); {
    if(xs[i] == x) { break; }
    i := i + 1;
  }
  r := i;
}
```

Fig. 2 Simple Boogie program encoding an implementation of the `indexOf()` function, making extensive use of the structured syntax provided in later versions of Boogie

using an uninterpreted function `len()`, and accompanying `axiom`. Secondly, the input array is modelled using the map `[int]int`, which is a total mapping from *arbitrary integers* to *arbitrary integers*. For example, `xs[-1]` identifies a valid element of the map despite `-1` not normally being a valid array index (e.g. in Whiley). We can refine this to something closer to an array through additional constraints, as shown in the next section.

Whilst the structured form of Boogie is preferred, where possible, it is also useful to consider the unstructured form, which we use for a few Whiley constructs such as `switch` (Sect. 3.4.1). Figure 3 provides an unstructured encoding of the `indexOf()` function from Figure 2. In this version, the `while` loop is decomposed using a non-deterministic `goto` statement—the `goto LOOP_BODY, LOOP_EXIT` statement allows flow of control to jump to either label, but the `assume` statements after those labels block progress if their condition is false. Likewise, in this unstructured encoding, the loop condition and invariant are explicitly assumed (lines 8,9,12) and asserted (lines 15,16), rather than being done implicitly by the tool (as in Figure 2). The `havoc` statement *"assigns an arbitrary value to each indicated variable"* [14], so is used here to indicate that variable `i` contains an arbitrary integer value at this point.

Finally, we note that Boogie allows one to designate preconditions, postconditions and loop invariants as `free`. This allows Boogie to assume these conditions hold *without checking them*—thereby (potentially) reducing overall verification time [103].

3 Modelling Whiley in Boogie

Our goal is to model as much of the Whiley language as possible in Boogie, so that we can utilise Boogie for verifying Whiley programs. Indeed, the motivation for this project was the hope that Boogie would offer significantly better verification capability than the existing (and relatively ad hoc) native verifier used in Whiley (and, as Sect. 4 shows, this is the case). At

```
procedure indexOf(xs: [int]int, x:int) returns (r:int)
... {
   var i : int;
   i := 0;
   assert i >= 0 && i <= len(xs); // invariant
   assert (forall k:int :: (0<=k && k<i)==> xs[k]!=x);
  LOOP_HEAD:
   havoc i;
   assume i >= 0 && i <= len(xs); // assume invariant
   assume (forall k:int :: (0<=k && k<i)==> xs[k]!=x);
   goto LOOP_BODY, LOOP_EXIT;
  LOOP_BODY:
   assume i < len(xs); // assume loop condition
   if(xs[i] == x) { goto BREAK_EXIT; }
   i := i + 1;
   assert i >= 0 && i <= len(xs); // invariant
   assert (forall k:int :: (0<=k && k<i)==> xs[k]!=x);
   goto LOOP_HEAD;
  LOOP_EXIT:
   assume i >= len(xs); // assume negated condition
  BREAK_EXIT:
   r := i;
}
```

Fig. 3 Unstructured encoding of the example from Figure 2—the pre-/postconditions are omitted as they are unchanged from above, and likewise for `len()`

a superficial level, Whiley's native verifier is not so different from Boogie/Z3. In particular, it employs an intermediate assertion language in which verification conditions are encoded and then discharged using a purpose-built SMT solver [139]. A key advantage is that the generated verification conditions resemble the Whiley source language much more closely. Nevertheless, whilst this toolchain has potential, it remains relatively immature compared with Boogie/Z3 and the considerable resources invested in their development [16]. However, this transition is not without challenges as, despite their obvious similarities, there remain significant differences between Whiley and Boogie:

- **Types.** Whiley has a relatively rich (structural) type system which includes: *union*, *record*, *array*, *reference* and *lambda* types. Furthermore, there is support for type polymorphism through generics.
- **Flow Typing.** Whiley's support for flow typing is also problematic, as a given variable may have different types at different program points and there is a need to support runtime type tests [133].
- **Functions.** Whiley functions are defined via code bodies, whereas the body of a Boogie function can contain only a single expression.
- **Methods.** Whiley methods correspond quite well with procedures in Boogie, but may be invoked from within expressions in Whiley.
- **Definedness.** Whiley *implicitly* assumes specification elements (e.g. pre-/postconditions and invariants) are well defined. This differs from other tools (e.g. Dafny) which require programmers to *explicitly* ensure that specification elements are well defined.

To understand the **definedness** issue, consider a precondition that contains an array reference, like `requires a[i] == 0`. In a language like Dafny, one would additionally need to explicitly specify `i >= 0 && i < |a|` to avoid the verifier reporting an out-of-bounds error. Such preconditions are implicit in Whiley, so must be (automatically) extracted by our translator and made explicit in the generated Boogie.

We now present the main contribution of this paper, namely a mechanism for translating Whiley programs into Boogie, which is implemented in our translator program, called Wy2B.[1]

3.1 Types

Finding an appropriate representation of Whiley types is a challenge. We begin by considering the straightforward (i.e. naive) shallow translation of Whiley types into Boogie, and highlight why this fails. Then, we present a more sophisticated approach which corresponds more closely with a *deep embedding* of types.

Shallow Embedding. The simple and obvious translation of Whiley types into Boogie would be a direct translation to the built-in types of Boogie. Here, an `int` in Whiley is translated into a Boogie `int`, which is appropriate since both languages support unbounded integers. A Whiley array (e.g. `int[]`) then translates to a Boogie map (e.g. `[int]int`, with appropriate constraints), and Whiley records can also be translated using Boogie's map type. However, by itself, this is not sufficient to model all Whiley types. For example, the type `int|null` has no obvious corresponding representation in Boogie. Likewise, a Whiley type test such as `x is int` requires additional machinery. So this shallow embedding where Whiley types are directly translated into Boogie types is insufficient.

Deep Embedding. To support the more complex types found in Whiley such as unions, we provide a deep embedding of all types into Boogie.[2] Specifically, we model *all* Whiley values as disjoint members of a single set, `Any`, and model the various Whiley types as subsets of this:

```
type Any;          // The set of all Whiley values.
```

For each Whiley type `T`, we define a membership predicate `T#is(Any)` that holds for values in `T`, an extraction function `T#unbox(Any)` that maps `Any` to a Boogie type, and an injection function `T#box(T)` which does the reverse. We axiomatise these two functions to define a partial bijection between a type's representation and its corresponding subset of `Any`. We also add Boogie axioms to ensure the subtypes of `Any` which correspond to each built-in Whiley type (`int`, `bool`, `T[]`, etc.) are mutually disjoint. This embedding has several advantages. Firstly, it is easy to model a Whiley user-defined subtype `S` by defining a predicate `S#is(v)` as `(T#is(v) && ...)`. Secondly, union types simply map to disjunctions of these type predicates. Thirdly, Boogie can prove equality of two `Any` values only if they are constructed using the same `T#box()` injection function from values that are equal.

Finally, to aid with the translation of compound types in Whiley (such as arrays—see Sect. 3.1.2 below) a special constant, `Void`, is used:

```
const unique Void : Any;
```

[1] See https://github.com/Whiley/Whiley2Boogie.

[2] An alternative (though untested) approach would be to utilise Boogie's support for algebraic data types.

Observe that, since this value has (by design) no counterpart in Whiley, we must ensure it remains disjoint from all other Whiley values.

3.1.1 Primitives

Integers. The mapping functions for the Whiley `int` type of unbounded integers are as follows (recall `int` is also the Boogie name for integers).

```
function Int#is(Any v) returns (bool) {
  (exists i:int :: Int#box(i) == v)
}
function Int#unbox(Any) returns (int);
function Int#box(int) returns (Any);

axiom (forall i:int :: Int#unbox(Int#box(i)) == i);
axiom (forall i:int :: Int#box(i) != Void);
```

Bits and Bytes. Whiley includes a native `byte` type which supports the usual plethora of bitwise operators, including left- and right shifts. For this, Boogie provides a family of bitvector types (e.g. `bv8`, `bv16`, etc.) of which `bv8` provides a suitable match. To use this, however, we must exploit various internal functions to implement bitwise operators as follows:

```
function Byte#box(bv8) returns (Any);
function Byte#unbox(Any) returns (bv8);
function Byte#is(v : Any) returns (bool) {
   (exists b:bv8 :: Byte#box(b) == v)
}
function {:bvbuiltin "bv2int"} Byte#toInt(bv8)
 returns (int);
function {:bvbuiltin "(_ int2bv 8)"} Byte#fromInt(int)
 returns (bv8);
function {:bvbuiltin "bvnot"} Byte#Not(bv8)
 returns (bv8);
function {:bvbuiltin "bvand"} Byte#And(bv8, bv8)
 returns (bv8);
...
axiom (forall b:bv8 :: Byte#unbox(Byte#box(b)) == b);
axiom (forall b:bv8 :: Byte#box(b) != Void);
```

Coercions. In order to utilise our deep embedding, values must be coerced to / from primitive Boogie types. Consider an assignment `x = 0` where `x` has type `int|null`. Since union types in Whiley are encoded as type `Any` in Boogie, we must coerce the value `0` (of Boogie type `int`) into its embedded form via `Int#box()`. Such an assignment is thus translated as `x := Int#Box(0);`. In general, our translation attempts to minimise the amount of boxing/unboxing. For example, generated expressions of the form `Int#Unbox(Int#Box(x))` are automatically reduced to `x`, etc. Amongst other things, this helps to simplify debugging!

3.1.2 Arrays

Whiley arrays are fixed-length sequences of values whose length can be queried at runtime (recall from Sect. 2.1.3 they have value semantics). We model Whiley arrays using: (1) a Boogie map `[int]Any` from integers to `Any` values; and (2) an uninterpreted function returning the length. The embedding requires a number of additional axioms, as follows. As before, we provide extraction/injection functions as follows:

```
function Array#box([int]Any) returns (Any);
function Array#unbox(Any) returns ([int]Any);
function Array#is(v : Any) returns (bool) {
   (exists a:[int]Any :: Array#box(a) == v)
}
axiom (forall i:[int]Any :: Array#unbox(Array#box(i))==i);
axiom (forall a:[int]Any :: Array#box(a) != Void);
// Helper constraining index to be in-bounds
function Array#in(a : [int]Any, i : int) returns (bool) {
   (i >= 0) && (i < Array#Length(a))
}
```

A key aspect of our embedding is the treatment of indices which are *out-of-bounds*. The primary issue is that Boogie maps (e.g. `[int]Any`) are infinite structures with no concept of bounds. Elements which have not been explicitly defined always exist with some arbitrary value. This presents a problem for equality of arrays, as illustrated in Figure 4. To resolve this we fix all out-of-bounds indices to the special `Void` value, and enforce this throughout the axioms that follow.

Array Length. We employ the following function for extracting the length of an array:

```
// Extraction for array length
function Array#Length([int]Any) returns (int);
// Length of an array is non-negative
axiom (forall a:[int]Any :: 0 <= Array#Length(a));
// Updates don't affect array length
axiom (forall a:[int]Any,i:int,v:Any ::
   (v != Void && Array#in(a,i))
    ==> (Array#Length(a) == Array#Length(a[i:=v])));
```

In the above, we take steps to ensure the axioms remain consistent. To understand this, consider the last axiom above which holds the array length invariant across an update. The value `v` being assigned cannot be `Void` as, otherwise, we could artificially reduce an array's length (e.g. by assigning `Void` to the last element). Finally, whilst our encoding of arrays here may appear somewhat elaborate, it does allow us to exploit Boogie's internal notion of equality. An alternative, however, would be to define a bespoke equality operator for arrays (though this is complicated by the presence of unions and recursive types).

Array Initialisers. Array values in Whiley can be constructed using the array literal syntax (e.g. `[0,4,3]`, etc.). This creates an array containing the given values (zero-indexed). To translate this we employ a constructor, `Array#Empty(int)`, as follows:

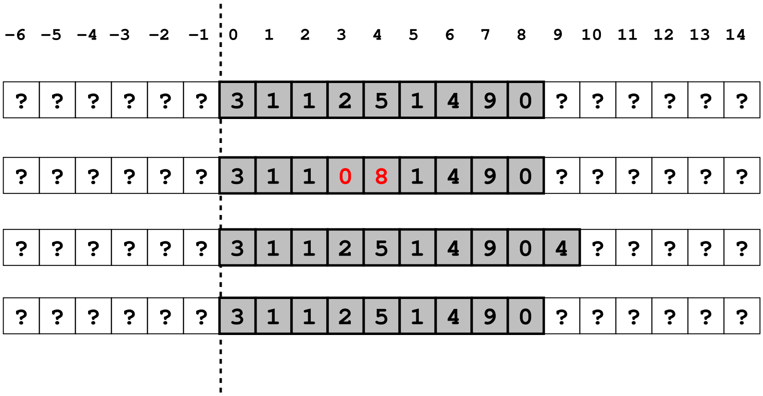

Fig. 4 A pictorial illustration of four arrays embedded using (infinite) Boogie maps, where undefined (i.e. out-of-bounds) values are shown as "?". We might expect the first and fourth arrays to be equal (i.e. since they have the same length and values within bounds), but this depends also on whether the out-of-bounds values are also equal. To ensure these two arrays are indeed equal, we fix these undefined values to some known constant (`Void`)

```
// Construct (empty) array literal of size n
function Array#Empty(int) returns ([int]Any);
// Fix out-of-bounds indices for array literal
axiom (forall l:int,i:int ::
  (i < 0 || l <= i) ==> (Array#Empty(l)[i] == Void));
// Fix in-bounds indices for array literal
axiom (forall l:int,i:int ::
  (0 <= i && i < l) ==> (Array#Empty(l)[i] != Void));
// Array length must match length of array literal
axiom (forall a:[int]Any,l:int ::
  (0 <= l && Array#Empty(l)==a) ==> (Array#Length(a)==l));
```

The intuition is that `Array#Empty(n)` constructs an *uninitialised* array of size `n`, whose elements must then be initialised individually. For example, the array literal `[6,3]` is translated into `Array#Empty(2)[0:=Int#box(6)][1:=Int#box(3)]`.

Array Generators. Array values can also be constructed using the array generator syntax, `[v;n]` (recall Sect. 2.1.1). The constructor, `Array#Generator(Any,int)`, is used for translating these as follows:

```
function Array#Generator(Any, int) returns ([int]Any);
// Every element of array generator matches given value
axiom (forall v:Any,l:int,i:int ::
 (0<=i && i<l && v!=Void) ==> Array#Generator(v,l)[i]==v);
// Fix out-of-bounds indices for array generator
axiom (forall v:Any,l:int,i:int ::
 (i < 0 || l <= i) ==> (Array#Generator(v,l)[i] == Void));
// Array length must match length of array generator
axiom (forall a:[int]Any,v:Any,l:int ::
 (0<=l && Array#Generator(v,l)==a) ==> Array#Length(a)==l);
```

3.1.3 Records

Records are encoded using maps, `[Field]Any`, where `Field` characterises *field names*. For every field name used within the program, a unique constant is created. For example, if the type `{int x, int y}` is used then the following constants are generated:

```
type Field;       // Set of all field names
const unique $x : Field;
const unique $y : Field;
```

These constants are then used as indices for the map encoding of the record (and any other record type containing a field `x` or `y`). The constants are marked `unique` to ensure they are disjoint. Thus, the number of constants generated depends on exactly what types are used within the target program. As for arrays, care must be taken when encoding a given record to ensure that all other fields are mapped to `Void`. Again, various functions and axioms are provided to allow records to be embedded within other compound types:

```
function Record#box([Field]Any) returns (Any);
function Record#unbox(Any) returns ([Field]Any);
function Record#is(v : Any) returns (bool) {
   (exists r:[Field]Any :: Record#box(r) == v)
}
axiom (forall i:[Field]Any ::
  Record#unbox(Record#box(i)) == i);
axiom (forall r:[Field]Any :: Record#box(r) != Void);
```

Like arrays, all fields not in a given record should hold `Void`. This cannot be enforced with an axiom as it depends upon the record type in question (i.e. what fields it has). Instead, this is enforced using constraints on parameters, returns and local variables as necessary. *Record Literals.* As for arrays, a simple constructor is used for translating record literals:

```
const unique Record#Empty : [Field]Any;
// Every field in an empty record holds Void
axiom (forall f:Field :: Record#Empty[f] == Void);
```

As an example, the record literal `{x:1,y:2}` would be translated into Boogie as `Record#Empty[$x:=Int#box(1)][$y:=Int#box(2)]`.

3.1.4 Generics

Type polymorphism in Whiley presents a number of challenges when translating to Boogie. Roughly speaking, we translate generic types (e.g. `T`) into Boogie's `Any` type. We will return to discuss this in more detail later (see §3.4).

3.1.5 Lambdas

The ability to pass around first-class functions and methods as lambdas also presents some challenges, since lambdas in Boogie are relatively restricted. We return to discuss this in more detail later (see §3.4.1), but for now it suffices to introduce the following which represents the set of all lambda values:

```
type Lambda; // Set of all lambda values

function Lambda#box(Lambda) returns (Any);
function Lambda#unbox(Any) returns (Lambda);
function Lambda#is(v : Any) returns (bool) {
   (exists l:Lambda :: Lambda#box(l) == v)
}
axiom (forall l:Lambda :: Lambda#unbox(Lambda#box(l))==l);
axiom (forall l:Lambda :: Lambda#box(l) != Void);
```

3.1.6 References

References in Whiley are modelled in a relative standard fashion as indexes into a *heap* represented as a map of the form `[Ref]Any` [103]. Again, we return to discuss this in more detail later (see Sect. 3.5), and for now, we simply introduce the `Ref` type:

```
type Ref; // Set of all Whiley references

function Ref#box(Ref) returns (Any);
function Ref#unbox(Any) returns (Ref);
function Ref#is(v : Any) returns (bool) {
   (exists r:Ref :: Ref#box(r) == v)
}
axiom (forall r:Ref :: Ref#unbox(Ref#box(r)) == r);
axiom (forall r:Ref :: Ref#box(r) != Void)
```

In addition, the following constant is provided for describing an arbitrary heap:

```
const unique Ref#Empty : [Ref]Any;
```

The above is useful in various situations where there is no logical heap (more on this later). In particular, since it does not provide any guarantee about its contents, it cannot be relied upon at all.

3.1.7 User-Defined Types

Our treatment of user-defined types follows naturally from our embedding of types discussed above. Roughly speaking, we can consider that every user-defined type in Whiley consists of two parts: firstly, its base or *underlying* type; secondly, its invariants (if any). For example, consider the following Whiley declaration:

```
type nat is (int x) where x >= 0
```

The underlying type of `nat` is `int`, and it enforces a single invariant `x >= 0`. In our translation to Boogie, this declaration would produce the following:[3]

[3] In practice, name mangling is applied to ensure uniqueness across modules and packages in Whiley.

```
type nat = int;

function nat#inv(x : int, HEAP : [Ref]Any) returns (bool) {
   x >= 0
}
function nat#is(x : Any, HEAP : [Ref]Any) returns (bool) {
   Int#is(x) && nat#inv(Int#unbox(x),HEAP)
}
```

This allows for several different use cases. For example, if we have a variable of type `nat` and wish to assume or assert its invariant, then `nat#inv()` can be applied directly. Alternatively, if we are reading such a variable from a boxed position (e.g. out of an array or record), then `nat#is()` can be applied. Observe also that, for uniformity, such methods always accept a `HEAP` parameter even if (as in this case) this is not used. This parameter is necessary for user-defined types which are, or contain, references. For example, consider this declaration which builds upon the definition of `nat`:

```
type pNat is (&nat p)
```

This describes the type of references to integer values which enforce the `nat` constraint. This would be translated as follows:

```
type pNat = Ref;

function pNat#inv(p:Ref,HEAP:[Ref]Any) returns (bool){true}
function pNat#is(p:Any, HEAP:[Ref]Any) returns (bool) {
  Ref#is(p) && nat#is(HEAP[Ref#unbox(p)],HEAP)
            && pNat#inv(Ref#unbox(p),HEAP)
}
```

Here, `pNat#is()` enforces `nat#is()` upon the element in `HEAP` referred to by `p`. Thus it becomes clear that the embedding of a reference type only makes sense in the context of a given `HEAP`.

3.2 Constants

Global constants in Whiley require care to ensure a safe translation. A well-known issue with Boogie arises when specifications written by the user (i.e. in Whiley) are translated into *unguarded* Boogie axioms. In such cases, the user can be considered as maliciously injecting problematic (though rarely useful) code.[4] For example, a user can (perhaps accidentally) insert `axiom false;` (or some equivalent thereof) into the generated Boogie file. Unfortunately, the presence of such a declaration allows Boogie to immediately verify all assertions in the file (i.e. regardless of whether they are correct or not) [102]. More importantly, Boogie does not report this as an error, and hence, it happens silently without the user being made aware. To see how this applies to constants in Whiley, consider the following (recall definition of `nat` from page 19):

[4] This is perhaps somewhat reminiscent of SQL injection attack, whereby a user submits arbitrary SQL (e.g. through a form) which is executed on the server (e.g. because an input string was not escaped properly).

```
final nat x = 0
```

The challenge here is to ensure the value being assigned adheres to any type invariant(s) required of `x`. One approach is to generate a typing axiom, such as `axiom nat#is(x)`, for this. Whilst this is sufficient for the above example, a problem arises if the value assigned was `-1` instead of `0`. In such case, the translation leads to the following:

```
const x : int;
axiom x == -1;
axiom nat#is(x);
```

Unfortunately, these axioms conflict as they imply both `x == -1` and `x >= 0` (which is equivalent to `axiom false`). To protected against this, we stratify our translation into two levels: the first establishes global constants are correctly initialised, and the second verifies functions and methods assuming they are correctly initialised. Following the approach taken in Dafny [104], this is done using a special constant `Context#Level`. The following illustrates the translation of our example above:

```
const Context#Level:int;

const x : nat;
axiom x == 1;
axiom (Context#Level > 1) ==> nat#is(x);

procedure x#check()
requires Context#Level == 1;
{
    assert nat#is(x);
}
```

The above verifies without trouble. However, were `x` to be initialised with `-1`, Boogie would now correctly report a failed proof obligation inside the `x#check()` method. Furthermore, note that all procedure bodies generated from functions or methods in Whiley require `Context#Level > 1` to ensure access to `x`'s invariant (see Figure 6 below).

3.3 Properties

Properties in Whiley are straightforward as they can be translated directly as Boogie functions. For example, consider the following property in Whiley:

```
property above(int[] xs, int n)
where all { i in 0..|xs| | n < xs[i] }
```

This is translated directly as follows (again name mangling would be applied in practice):

```
function above(HEAP : [Ref]Any, xs : [int]Any, n : int)
 returns (bool) { (forall i:int ::
  Array#in(xs,i) ==> n < Int#unbox(xs[i])
)}
```

```
function fill(int[] xs, int x) -> (int[] rs)
// Result has same dimension
ensures |xs| == |rs|:
    //fill elements!
    for i in 0..|xs|:
       xs[i] = x
    return xs
```

Fig. 5 Illustrating a simple function in Whiley which, for brevity, has not been fully specified

As for types, properties always accept a `HEAP` parameter for uniformity even when not needed. A key observation is that Boogie functions are strictly more expressive than properties in Whiley, and we will return later to consider the impact of this (see Sect. 4.4).

3.4 Functions

Recall that functions in Whiley are pure, have bodies comprised of statement blocks and may have multiple return values. This differs from functions in Boogie, whose bodies are made up of a single expression and can only return a single value. This presents challenges: firstly, the body of a Whiley function corresponds more closely with a Boogie procedure; but, secondly, functions in Whiley can be called from specification elements (e.g. pre-/postconditions) whereas Boogie procedures cannot. As such we provide a *two-pronged* translation (similar to that found in Dafny [102]) comprising: a prototype implemented as a Boogie function which can be invoked from a specification element; and a body, implemented as a Boogie procedure, which can be invoked directly from the body of other functions or methods.

Figure 5 illustrates a simple function written in Whiley which we adopt as a running example, whilst the generated Boogie for this is shown in Figure 6. We will endeavour to fully clarify all aspects of this figure over the coming pages, but for now we focus on the procedure's specification. Here, additional `requires` clauses are included to enforce the type of `xs` and, likewise, additional `ensures` clauses for the type of `rs`. Whilst the soundness assumption for constants was discussed above, we will return to discuss the purpose of the function prototype and linkage later. We note also that, whilst functions in Whiley cannot modify the heap, they can manipulate references as simple values (though cannot mutate through them).

Generics. Since Whiley supports type polymorphism, we might like to upgrade our `fill()` function as follows:

```
function fill<T>(T[] xs, T x) -> (T[] rs)
  ...
```

As discussed in Sect. 3.1.4, we translate the Whiley type `T` as Boogie type `Any`. In terms of verifying the above function in isolation, this presents no problems. However, in most cases, call sites of this function would expect to receive an array of the same type they put in. For example, consider this:

```
function zero(int[] xs) -> (int[] ys):
  return fill<int>(xs,0)
```

In this case, Boogie must be able to determine that the return from `fill()` is an array of integers. Thus, a mechanism is required to enable our translation to state *meta properties*

```
// Procedure
procedure fill(xs : [int]Any, x : int) returns (rs : [int]Any);
// Elements in xs have integer type
requires (forall i:int :: Array#in(xs,i) ==> Int#is(xs[i]));
// Out-of-bounds elements in xs are void
requires (forall i:int :: !Array#in(xs,i) ==> (xs[i] == Void));
// Soundness assumption
requires Context#Level > 1;
// Elements in rs have integer type
ensures (forall i:int :: Array#in(rs,i) ==> Int#is(rs[i]));
// Out-of-bounds elements in rs are void
ensures (forall i:int :: !Array#in(rs,i) ==> (rs[i] == Void));
// Given ensures clause
ensures Array#Length(xs) == Array#Length(rs);
// Function linkage
free ensures fill(HEAP,xs,x) == rs;
{
   var i : int;
   i := 0;
   while(i < Array#Length(xs))
   // type preservation
   invariant (forall k:int :: Array#in(xs,k) ==> Int#is(xs[k]));
   invariant (forall k:int :: (!Array#in(xs,k)) ==> (xs[k] == Void));
   // for loop invariant
   invariant (0 <= i) && (i <= Array#Length(xs));
   {
      // well-definedness for assignment
      assert 0 <= i;
      assert i < Array#Length(xs);
      // translation of assignment
      xs := xs[i:=Int#box(x)];
      // invariant preservation for assignment
      assert (forall k:int :: Array#in(xs,k) ==> Int#is(xs[k])
      assert (forall k:int :: (!Array#in(xs,k)) ==> (xs[k] == Void));
      //
      i := i + 1;
   }
   rs := xs;
   return;
}
// Function Prototype
function fill(HEAP:[Ref]Any, xs:[int]Any, x:int) returns ([int]Any);
```

Fig. 6 Illustrating the generated Boogie code for the `fill()` example. Note that this is somewhat simplified as various details related to name mangling and parameter shadowing are omitted

about the relationships between variable types (e.g. that they are the same). To do this, we introduce meta types as follows:

```
type Type;
// Meta type test
function Type#is([Ref]Any, Type, Any) returns (bool);
```

Here, the Boogie type `Type` represents the set of all meta types, whilst `Type#is()` performs a similar function as, for example, `Int#is()` (but for an arbitrary meta type). In this way, we extend the generated procedure for `fill<T>()` as follows:

```
procedure fill(T:Type, xs:[int]Any, x:Any)
  returns (rs:[int]Any);
// Elements in xs have type T
requires (forall i:int :: Array#in(xs,i)
                          ==> Type#is(HEAP,T,xs[i]));
...
// Elements in rs have type T
ensures (forall i:int :: Array#in(rs,i)
                          ==> Type#is(HEAP,T,rs[i]));
...
```

Here, we see the generic type `T` is now passed as an *argument* to procedure `fill()` and using this we can, for example, make statements about the return value. For example, the postcondition now tells us at a given call site that all elements in the returned array have the same type as those elements in the input array. To make this work, we still need one additional piece. Specifically, for every type which can be used to instantiate a type variable (e.g. `int` in `fill<int>()`) we construct a unique meta type constant. For example, the meta type constant for `int` is declared as follows:

```
// Int meta type
const unique Type#I : Type;
// Int meta axiom
axiom (forall HEAP:[Ref]Any,v:Any ::
        Type#is(HEAP,Type#I,v) <==> Int#is(v));
```

Finally, we note that user-defined types must be extended to use meta types as well. For example, consider the following:

```
type List<T> is (null | { List<T> next, T data } l)
```

The various Boogie support functions generated for this type (recall Sect. 3.1.7) must now accept a meta type parameter. For example, `List#is()` is defined as:

```
function List#is(T:Type, l:Any, HEAP:[Ref]Any)
returns (bool) { ... }
```

Overloading and Parameters. Overloading on parameter types is supported in Whiley, but not in Boogie. To resolve this, we employ name mangling for every property, function, method and type. The latter is necessary because mangling also includes package and module information. Likewise, parameters for Boogie procedures are immutable, whereas parameters to functions or methods can be mutated in Whiley. To resolve this, our translator generates a *shadow* variable for each parameter which is assigned the parameter's value on entry.

Function Linkage. Since functions in Whiley can be called from specification elements, a key question arises as to how such calls are encoded. Consider the following partial implementation of a stack:

```
type Stack is {int[] items, nat len} where len < |items|

function size(Stack b) -> (nat r):
   return b.len

function top(Stack s) -> (int v) requires size(s) > 0:
   return s.items[s.len]
```

Here, the … postcondition of `top()` uses other publicly visible functions to *hide* the implementation of `Stack`.[5] Our translation of `top()` looks roughly as follows:

```
procedure top(s : Stack) returns (v : int);
...
requires size(HEAP,s) > 0;
...
```

The key here is that `size(HEAP,s)` refers to the *function prototype* of `size()`, rather than its *procedure*. Furthermore, since `size(HEAP,s) == r` is ensured in the postcondition of procedure `size()`, we can verify statements such as the following:

```
  Stack s = ...
  if size(s) > 0:
    return top(s)
```

Partial Correctness. An important limitation of Whiley is that it cannot ensure termination. For example, there is no equivalent syntax to `decreasing` as found in Dafny. As a result, non-terminating recursive functions can be verified with almost any postcondition. The following illustrates such an example:

```
function inc(int x) -> (int y) ensures y > x:
   return inc(x)
```

Observe that the above function will never violate its postcondition and, hence, is correct *up to non-termination*. In the future, we expect Whiley to be extended with support for *variant* expressions such that a well-founded ordering over recursive calls can be specified to ensure termination.

3.4.1 Statements and Expressions

Translating most Whiley statements and expressions into Boogie is straightforward (see the similarities between Figures 1 and 2). Here, we describe only the interesting cases that present specific challenges.

Variable Scoping.

Boogie requires all local variables to be declared at the start of a procedure body where, like most modern languages, Whiley allows variables to be declared with block scopes. Whilst, in most cases, this is relatively trivial to manage there are cases where name clashes arise. The following illustrates:

[5] We note that Whiley supports a range of visibility modifiers for statically enforcing information hiding, though these are beyond the scope of this paper.

```
type imsg_t is {int kind, int data}
type bmsg_t is {int kind, bool data}

function read(imsg_t|bmsg_t m) -> (int r):
  if m is imsg_t:
    int tmp = m.data
    ...
  else:
    bool tmp = m.data
    ...
```

In this case, the same variable is declared twice with different types. This is a problem because they have incompatible types, and hence, we cannot declare a single Boogie variable to cover both. Instead, we apply name mangling to ensure variables in different scopes have unique names.

Well-Definedness. As highlighted already, Whiley's treatment of expressions (especially when used in specification elements such as pre-/postconditions) differs from other comparable systems (e.g. Dafny). In fact, handling this is straightforward and has been covered reasonably extensively elsewhere [102]. Essentially, when translating a Whiley expression, care must be taken to insert checks as necessary to ensure expressions are well defined. The following illustrates a simple example:

```
  ...
  int x = xs[i]
  ...
```

Here, there is an implicit assumption that `i >= 0` and `i < |xs|`. Of course, this may not actually be the case and we employ `assert` statements to check such preconditions. As such, the above is translated roughly as follows:

```
   ...
   assert 0 <= i;
   assert i < Array#Length(xs);
   x := Int#unbox(xs[i]);
   ...
```

Whilst, in many cases, the extraction of such checks is straightforward there are some challenges. For example, we employed window inference [148] here. To understand this, consider the following:

```
  ...
  if i < |xs| && xs[i] == w:
    ...
```

For this example, the following translation is not sufficient:

```
  ...
  assert (0 <= i);
  assert (i < Array#Length(xs));
  if(i < Array#Length(xs) && Int#unbox(xs[i]) == w) {
    ...
```

This translation is invalid because the second `assert` may not hold. This arises because this definedness check is for part of the condition *in a given context*. Instead, for every check extracted, we must additionally extract facts which have become known within the expression. Thus our translation, in fact, is as follows:

```
  ...
  assert (i < Array#Length(xs)) ==> (0 <= i);
  assert (i < Array#Length(xs)) ==> (i < Array#Length(xs));
  if(i < Array#Length(xs) && Int#unbox(xs[i]) == w) {
    ...
```

Another aspect of this issue is the well-definedness of specification elements, such as pre-/postconditions and loop invariants. Consider the following (albeit contrived) example:

```
function read(int i, int[] map) -> (int r)
requires map[i] >= 0:
  ...
```

Since the precondition for this function requires facts about `map[i]`, it follows (implicitly) that `map[i]` must be well defined (i.e. that `i` is within bounds). Thus, our translator extracts such additional requirements as necessary, as the following illustrates:

```
procedure read(i : int, map : [int]Any) returns (r : int);
...
requires (0 <= i) && (i < Array#Length(map));
requires Int#unbox(map[i]) >= 0;
...
```

A similar approach is taken to handling loop invariants and, perhaps surprisingly, also for postconditions. For example, consider the following (albeit also contrived) example:

```
function create(int n) -> (int[] xs)
ensures xs[0] == n:
  ...
```

In this case, it follows from the postcondition that `|xs| > 0` holds and, hence, is translated as follows:

```
procedure create(n : int) returns (xs : [int]Any);
...
ensures (0 <= 0) && (0 < Array#Length(xs));
ensures Int#unbox(xs[0]) == n;
...
```

Finally, we note that care must be taken in a number of contexts when extracting well-definedness conditions, such as for expressions nested within quantifiers.

Type Invariants. Our translation must ensure type invariants are properly preserved at all points. For example, consider the following (recall definition of `nat` from page 19):

```
...
nat x = 1
...
x = y + 1
```

In this case, we must establish that `x >= 0` holds after `x` is initialised, and also after it is subsequently reassigned. To do this, the above is translated as follows:

```
...
var x : nat;
x := 1;
assert nat#is(Int#box(x),HEAP);
...
x := y + 1;
assert nat#is(Int#box(x),HEAP);
```

Here the Boogie function `nat#is()` encapsulates the invariant for `nat` and is generated where translating the type declaration (recall Sect. 3.1.7).

Looking at Figure 6 provides further insight into this process. No assertion for invariant preservation is generated for `i := i + 1` because the type of variable `i` is unconstrained. In other words, since the check would correspond to `assert true;` we simply optimise it away. However, such optimisation remains relatively simplistic, as checks are still produced unnecessarily for the `xs := xs[i:=Int#box(x)]` assignment.

Invocation. Translating function invocations into Boogie presents something of a challenge, since functions can be invoked from arbitrary expressions (including specification elements discussed previously in §3.4). However, Boogie does not permit `procedure` invocations from within an expression, and provides only a simple statement form for calling procedures (e.g. `call x := f(y);`).[6] In short, this means function invocations must be extracted from expressions. Consider the following snippet in Whiley:

```
int y = f(x) + 1
```

The above is translated into the following Boogie sequence:

```
call f#114 := f(x);
y := f#114 + 1;
```

Here a temporary variable, `f#144`, is introduced to hold the value returned from `f(x)`. Thus, the order of evaluation for expressions is exposed by the order in which the calls are made prior to the final expression. In general, this approach works fine, but there are challenges. Short-circuit semantics presents the first challenge. For example, consider the following:

[6] Presumably, this is because Boogie wants to remain agnostic regarding execution order of expressions.

```
  if (x < 0) || (f(x) > 0):
    ...
```

In this case, we cannot just extract the function invocation and execute it before the `if` statement. Such a translation would model `f(x)` being executed every time the `if` statement is executed, which is not the case. Instead, we must carefully preserve short circuit semantics using unstructured branching as necessary. For example, we can translate the above as follows:

```
   if(x < 0) { goto trueLab; }
   call f#114 := f(x);
   if(f#114 <= 0) { goto falseLab; }
 trueLab:
   ...
 falseLab:
```

We can see that, whilst this gives a faithful rendition of the original program, it is quite low level and harder to comprehend. This issue is further compounded with loops, whose unstructured representation is far more verbose (recall Figure 2 versus Figure 3).

Finally, we note our approach above is reminiscent of that used for Spec# [113] but differs from Dafny (because Dafny does not permit method calls within expressions).

Assignments. Boogie supports assignments to variables (e.g. `x := y;`) and map elements (e.g. `xs[0] := 0;`). Unfortunately, our choice to represent arrays uniformly with Boogie type `[int]Any` presents some minor challenges. For a Whiley variable `xs` of type `int[]`, we could translate `xs[i] = 0` directly as `xs[i] := Int#box(0);`. However, for a Whiley variable `ys` of type `int[][]` a direct translation fails because the Boogie type for `ys` is still `[int]Any` (i.e. not `[int][int]Any` as needed for a direct translation). For simplicity, we translate array assignments uniformly regardless of the nesting level. For example, consider the following:

```
  ...
  xs[i] = 0
```

As seen in Figure 6, the above is translated using Boogie's `m[e->v]` operator as follows:

```
  xs := xs[i:=Int#box(0)];
```

A similar approach is needed for assignments to records and to the heap via references.

A slightly more challenging issue arises from *multiple assignments* in Whiley. These have interesting semantics from a verification perspective. Consider this example:

```
type Point is { int x, int y } where x < y || x > y

function swap(Point p) -> (Point r):
  p.x,p.y = p.y,p.x
  return p
```

The semantics of multiple assignments mean that the type invariant of `p` must hold *after* the assignment (hence the above correctly preserves its invariant). Observe, however, that attempting to assign each field individually would give a verification error, as the type invariant for `p` would be temporarily broken. Thus, our translation of the above would be:

```
...
t#0 := Int#unbox(p$97[$y]);
t#1 := Int#unbox(p$97[$x]);
p := p[$x:=Int#box(t#0)];
p := p[$y:=Int#box(t#1)];
assert Point#is(Record#box(p),HEAP);
...
```

Notice that the values of `p.y` and `p.x` are first stored in temporary variables to avoid interference between the left- and right-hand sides.

Another important aspect of multiple assignments is the semantics for conflicting assignments [75, 76]. The following illustrates:

```
xs[i],xs[j] = 0,1
```

They key question is what value is assigned to `xs[i]` when `i==j`. We follow Gries by resolving this based on the *order* of the right-hand side. Thus, when `i==j` above, `xs[i] == 1` holds after the assignment since `xs[i]` is first assigned `0` then `1`. This differs from Dafny where the above would be rejected unless `i!=j` was known.

Switches. Like many languages, Whiley supports multi-way branching via `switch` statements. Although Boogie has no switch statement, it does support non-deterministic `goto`. Hence, rather than using a sequence of if–else statements, we exploit this with appropriate constraints. The following illustrates:

```
switch c:
  case 0,1:
    ...
  default:
    ...
...
```

```
goto l1,l2;
l1:
  assume (c == 0) || (c == 1);
  ...
  goto l3;
l2:
  assume (c != 0) && (c != 1);
  ...
l3:
```

Here, `l1` corresponds with `case 0,1` whilst `l2` corresponds with the default case. Note also that cases do not fall through by default in Whiley. Furthermore, if there are nested `break`/`continue` statements these are translated into `goto`s as well.

Loops. Loops are also relatively easy to translate. Since Boogie supports only `while` loops, all other looping forms found in Whiley must be translated using this. Furthermore, since Boogie has no `break` or `continue` statement, we translate these using `goto`s as for `switch` statements. We note also that, for a do–while loop in Whiley, the loop invariant need not hold before the first iteration (which makes some proofs easier). Furthermore (if desired) one can always check the invariant on entry using an explicit `assert` statement.

One challenge faced in translating loops is the handling of types for variables which are modified in a loop. For example, in our translation of `fill()` our translator inserted additional loop invariants to preserve the type of variable `xs` (recall Figure 6). This is necessary because the postcondition for `fill()` restates that `rs` is an array of integers and this is not expressed explicitly in the type `[int]Any`. Indeed, this is stated for `xs` in the function's precondition but, since `xs` is modified in the loop, this information is lost within and after the loop (because Boogie sends its value to havoc). To resolve this, we must reassert this type information as a loop invariant. Furthermore, this is done for any variables modified in the loop.

A related issue, which our translator does not currently address, is that of preserving immutable properties of variables. Consider again the `fill()` example from Figure 5. In fact, this example does not verify as is with our translator! Again, key information about `xs` is lost within and after the loop. In this case, the information that needs to be preserved is that the *length* of `xs` is unchanged by the loop. In principle, our translator could be extended with a static analysis to infer this and add it implicitly as a loop invariant (but this remains future work). We note that this extends to records, as the following illustrates:

```
type Buffer is {nat len, int[] items} where len < |items|

function clean(Buffer b) -> (Buffer r)
// Buffer is emptied!
ensures (r.len == 0):
   b.len = 0
   for i in 0..|b.items|:
      b.items[i] = 0
   return b
```

Perhaps surprisingly, this also does not verify because the property `b.len == 0` is not preserved across the loop. This can be fixed by performing the assignment to `b.len` *after* the loop. Or, we could add a loop invariant to ensure `b.len == 0` is preserved.

Lambdas. Boogie provides syntax (e.g. `(lambda y:int :: y + 1)`) for lambdas (with map type `[int]int` in this case). They are comparable with Boogie functions and cannot, for example, call procedures, etc. As such, they are insufficient for representing lambdas in Whiley which can have side effects. Instead, we translate them into named Boogie procedures. Mostly this is straightforward, but a few challenges arise with captured variables. For example, consider the following:

```
type Pred<T> is function(T)->(bool)

function isBelow(int n) -> Pred<int>:
   return &(int v -> v < n)
```

Translating the lambda into a standalone `procedure` requires identifying captured variables (`n` in this case) and adding them as parameters. The following illustrates:

```
procedure lambda#131(HEAP : [Ref]Any, v : int, n : int)
returns (r : bool);
{
  ...
}
const unique lambda#131 : Lambda;
```

Here, the procedure contains the body of the lambda, which will include any necessary checks on the lambda itself. Likewise, the constant `lambda#131` is generated to represent this particular lambda. When translating an indirect invocation, we automatically generate a suitable prototype to invoke. For example:

```
  Pred<int> fn = ...
  bool b = fn(10)
```

The above Whiley snippet is then translated (roughly speaking) as follows:

```
   fn := ...;
   assert Pred#is(Type#L,Lambda#box(fn),HEAP);
   b := Bool#unbox(f_apply(fn,Int#box(10)));
```

Here, the function `f_apply()` is generated (in practice, with a suitable mangling) to represent the anonymous function being invoked. It accepts the lambda as a parameter, thus allowing one to exploit the fact that the same lambda returns the same value(s) when given the same parameter. Finally, we note that work remains to improve our translation of lambdas. In particular, information known about captured variables is not currently transferred to the generated `procedure`. Thus, the following fails to verify:

```
function isBelow(int[] xs, int i) -> Pred<int>
// index i within bounds
requires i >= 0 && i < xs[i]:
   // Return lambda
   return &(int v -> v < xs[i])
```

the generated `procedure` accepts the captured variables `i` and `xs`, but does not include a corresponding precondition. Whilst, in this case, it would be relatively easy to fix, in other cases it is more challenging (e.g. when a parameter has been modified prior to being captured). One approach, for example, would be to apply the Weakest Precondition transformer [17, 57, 101] to the body of the lambda (which should be relatively straightforward since this is just an expression).

3.5 Methods and Framing

Recall from Sect. 2.1.4 that methods in Whiley are permitted to have side effects and, for example, manipulate heap-allocated data through references. As such, Whiley methods correspond closely with procedures in Boogie. However, methods in Whiley can be called from expressions used in statements (though not from specification elements, such as pre-/postconditions or loop invariants). In many ways, the translation of methods follows that for functions, but with some important differences which we now consider.

Framing. Whilst the Whiley language provides relatively limited support for describing the effect a method has on the heap, a lot of machinery is nevertheless required to manage what can be expressed. As highlighted before, we adopt a relatively standard approach to modelling the heap. Specifically, a global variable `HEAP` of type `[Ref]Any` is provided to model this. For example, consider the following Whiley method:

```
method swap(&int p, &int q)
ensures *p == old(*q) && *q == old(*p):
   ...
```

Our translation produces both a procedure prototype and implementation in Boogie. The prototype for the above method looks roughly as follows:

```
procedure swap(p : Ref, q : Ref);
// Heap may be modified
modifies HEAP;
// Incoming typing constraints
requires Int#is(HEAP[p]) && Int#is(HEAP[q]);
// Postcondition
ensures Int#unbox(HEAP[p]) == old(Int#unbox(HEAP[q]))
        && Int#unbox(HEAP[q]) == old(Int#unbox(HEAP[p]));
// Outgoing typing guarantees
ensures Int#is(HEAP[p]) && Int#is(HEAP[q]);
// Frame condition (i)
free ensures ...
// Frame condition (ii)
free ensures ...
```

Observe that the `modifies` clause is provided as we must conservatively assume methods may modify the heap. Note also that `old()` in Whiley is translated directly using Boogie's `old()` syntax. There are two essential issues here: *typing* and *framing*. The former simply makes explicit guarantees on the shape of the heap provided by Whiley's type system. For example, that for an integer reference `p` there is indeed an integer value at `HEAP[p]`, etc. The latter aspect of framing is perhaps more interesting. We divide framing into two separate conditions (both of which are marked `free` since Whiley's type system guarantees them). These conditions rely on a simple predicate for determining whether a reference is reachable from—or *within*—the frame of a given variable (see Figure 7).

The first frame condition enforces *self-framing* [92] by ensuring that only locations within the method's *frame* can be modified:

```
...
// Frame Condition I
free ensures (forall r:Ref ::
   Ref#within(HEAP,r,p) || Ref#within(HEAP,r,q)       // (1)
                        || (old(HEAP[r]) == HEAP[r])  // (2)
                        || (old(HEAP[r]) == Void));   // (3)
...
```

```
// Check if reference within arbitrary value
function Any#within(HEAP:[Ref]Any, p:Ref, q:Any)
returns (bool) {
   (Ref#is(q) && Ref#within(HEAP,p,Ref#unbox(q))) ||
   (Array#is(q) && Array#within(HEAP,p,Array#unbox(q))) ||
   (Record#is(q) && Record#within(HEAP,p,Record#unbox(q)))
}
// Check if reference within array
function Array#within(HEAP:[Ref]Any, p:Ref, q:[int]Any)
returns (bool) {
    (exists i:int :: Any#within(HEAP,p,q[i]))
}
// Check if reference within record
function Record#within(HEAP:[Ref]Any, p:Ref, q:[Field]Any)
returns (bool) {
    (exists f:Field :: Any#within(HEAP,p,q[f]))
}
// Check if one reference (p) reachable from another (q)
function Ref#within(HEAP:[Ref]Any, p:Ref, q:Ref)
returns (bool) {
    (p == q) || Any#within(HEAP,p,HEAP[q])
}
```

Fig. 7 Illustrating the Boogie definition of a predicate for determining whether a reference `p` is *within* the footprint of given variable `q`. More specifically, it searches the contents of `q` (whatever that might be) looking for `p`, whilst traversing references as necessary

There are three parts of the condition as follows:

1. **(Mutable)** This identifies which locations could be modified by the method and, for these, does *not* provide a connection between the heap beforehand with that after.
2. **(Immutable)** For locations which could not be modified by the method, an explicit connection is made to ensure this between the heap beforehand and that after.
3. **(Allocated)** As a special case, heap locations which did not exist prior to the method (i.e. were mapped to `Void`) can have arbitrary values afterwards.

In essence, the footprint of a method (i.e. those locations it could write) is conservatively tied with its frame (i.e. those locations it could read). This provides a straightforward and extensible basis for reasoning about how methods modify the heap. For example, if syntax for describing the old heap in postconditions was added to Whiley, this would easily layer on top. The key is that, in the absence of more expressive syntax for restricting the locations a `method` may modify, we must adopt a worst-case assumption *that any reachable location could be modified.*

The second frame condition (known as the *swinging pivots restriction* [92]) prevents unreachable locations from "migrating" into the frame:

```
...
// Frame Condition II (Swinging Pivots)
free ensures (forall r:Ref ::
  // any reference r in postframe
 (Ref#within(HEAP,r,p) || Ref#within(HEAP,r,q)) ==>
  // (1) was either freshtly allocated, or
 (old(HEAP[r]) == Void ||
  // (2) was reachable from the preframe
  Ref#within(old(HEAP),r,p) || Ref#within(old(HEAP),r,q));
...
```

In essence, this ensures that any reference reachable from parameters `p` or `q` after the method was either freshly allocated, or was reachable from them beforehand. Note that, whilst for this particular method, these conditions are trivial they are required in general (e.g. for handling linked structures).

As a further example to illustrate the challenges addressed by the frame conditions, consider the following:

```
type LinkedList is null | &{ int data, LinkedList next }

method clear(LinkedList l):
   l->data = 0

method main():
   LinkedList l1 = new {data:1, next:null}
   LinkedList l2 = new {data:2, next:null}
   // Clear first node of l1 twice!
   clear(l1)
   clear(l1)
   // Check l2 unaffected
   assert l2->data == 2
```

Establishing that `l2` is *not* modified by the calls to `clear(l1)` above requires *both* frame conditions (something which is not immediately obvious at first glance). It is clear that the first frame condition (self-framing) allows us to establish that `l2` is not modified by the first call. One might then conclude the first condition is sufficient to establish this across both calls—but that is not the case! The challenge is that `clear(l1)` ensures `l2` is not modified, but allows `l1` to be modified. Without the second frame condition, the verifier might then consider that `l2` was within `l1` after the first call (e.g. that `l1->next == l2`). And, in such case, it would then rightly conclude that `l2` *could be modified by the second call*. As such, we see how the second frame condition helps to ensure that disjoint frames remain disjoint.

Finally, we note that our encoding makes heavy use of a recursive predicate (see Figure 7) which (as we have observed) can lead to the *butterfly effect* [111]. That is, where the verifier loops indefinitely unrolling predicates fruitlessly. In our experience, this typically happens when the condition being checked is invalid, and hence, the verifier cannot quickly find a proof by contradiction.

Allocation. Since data can be allocated on the heap in Whiley methods using the `new` operator, a translation of this operator is required. To this end, we employ the following:

```
procedure Ref#new(val : Any) returns (ref : Ref);
modifies HEAP;
// Location not previously allocated
ensures old(HEAP[ref]) == Void;
// Location now holds given value
ensures HEAP[ref] == val;
// Everything else untouched
ensures (forall r:Ref :: ref==r || old(HEAP[r])==HEAP[r]);
```

This simply returns an arbitrary location which was not previously allocated, and ensures it now holds the requested value. Recall that, at the time of writing, Whiley does not support explicit memory deallocation, and hence, no counterpart for this is required. Finally we note that, since allocations result in calls to `Ref#new`, they must be extracted from expressions as for method invocations above.

4 Experimental Results

In this section, we compare our Wy2B translator against the Whiley native verifier using the existing compiler test suite which consists of 1100+ (mostly) small Whiley programs. In particular, we are concerned with the number of tests that Wy2B can pass correctly, and note that the existing Whiley native verifier does not pass all the tests (e.g. because of outstanding bugs, etc.). In addition, we discuss our experiences using the new Wy2B toolchain on several larger case studies.

4.1 Micro-test Statistics

The Whiley compiler system includes a comprehensive suite of "micro"-test programs, which are small Whiley programs intended to methodically test all Whiley language features, including the Whiley native verifier. At the time of this evaluation (May 2021), this test suite included 731 "valid" micro-test case programs that should be verifiable, as well as 461 "invalid" micro-test case programs that should generate compiler errors or verification failures (to ensure that the compiler correctly catches them). Our first step in evaluating the correctness and usefulness of our new verifier is to apply it to this test suite. We use Boogie v2.8.26 and Z3 v4.8.10 for these evaluations.

When we applied our new Wy2B verifier to the *invalid* programs, ignoring 7 programs that are marked as IGNORE due to current limitations of the compiler front end, we found that all 454 of the remaining programs failed as expected. This confirms that the Boogie back end is correctly detecting verification issues in programs that should not be verifiable. For completeness, we illustrate one such example:

```
function f(int[] xs) -> int[]:
    xs[0] = 1
    return xs
```

The above "invalid" program is used to test that the verifier correctly reports a potential out-of-bounds access on line 2. Both the native verifier and our Wy2B verifier pass this test.

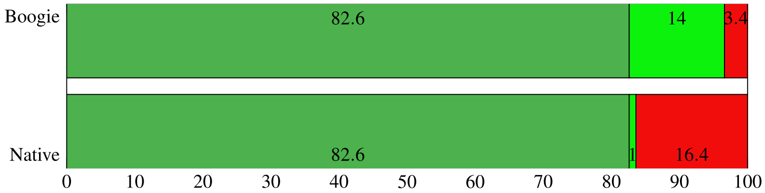

Fig. 8 Stacked bar chart of the Whiley native verifier and Boogie-based verifier results on the "valid" test programs. Green (left and middle bars) indicates percentage of programs fully verified, and red (right) indicates percentage where one or more proofs failed or timed out

The *valid* micro-test programs are small Whiley programs (ranging from 3 to 250 lines of code with an average length of 18 lines) that each contain several (2.2 on average) function and method definitions, some with specifications and some without. Around one third of the programs have functions or methods with requires/ensures specifications, one third use arrays (which generate array bound proof obligations), and 21% have loops with invariants. On average, our Wy2B translator generates 6.0 explicit proof obligations per micro-test program (to check array bounds, function call preconditions, etc.). This is on top of any explicit `assert` statements in the Whiley program and also in addition to the main proof obligations of Boogie, which are that each function or method body correctly implements its specification, and that every loop invariant is correctly preserved. Again, for completeness we illustrate one such example:

```
function f(int[] xs) -> (int r)
requires xs[0] >= 0
ensures r >= 0:
    return xs[0]

public export method test():
    int[] xs = [1,2,3]
    int x = f(xs)
    assert x >= 0
```

The above "valid" program is expected to pass verification without raising any errors. This means that, amongst other things, the verifier must prove that the body of `f` satisfies its specification, and within `test` must establish the precondition for the call `f(xs)` and that the final `assert` holds. Again, both the native verifier and our Wy2B verifier pass this test.

Figure 8 compares the percentages of these "valid" micro-tests that the native Whiley verifier and the Wy2B Boogie-based verifier can verify respectively. The leftmost bar on each row corresponds to the programs that both verifiers can verify (604 programs, or 82.6%). The middle bars show that the Whiley native verifier can verify an extra 7 programs (1.0%), whereas the Boogie verifier can verify an additional 102 programs (14.0%). So in total, the Whiley native verifier can verify 83.6% of the programs, whilst the Boogie verifier can verify a total of 96.6%.

We investigated the 7 programs that the Whiley native verifier could verify but Boogie could not, and found that 4 of them are verifiable by a later version of Boogie (v2.9.6.0) and Z3 (v4.8.12). The remaining three are due to outstanding issues with the translation to

Boogie related to lambda functions that return union types (Issue #59 in the Whiley2Boogie repository) and to proving the type invariants of cyclic data structures (Issue #61).

The larger number of programs that are verifiable by Boogie but not by the Whiley native verifier are largely because there are several Whiley language features that are not supported by the Whiley native verifier, such as:

- heap updates;
- reasoning about the results of calls to lambda functions;
- some kinds of generic types.

The Wy2B+Boogie toolchain takes 15:30 minutes (930 seconds) to translate and verify just the 706 test programs that it can verify, on a Dell Precision 5520 laptop with an Intel i7-7820HQ CPU @ 2.90GHz and 32Gb RAM, and a 60 second timeout for Boogie. This is 1.3 seconds on average for each small valid test program, which is acceptable performance for real-world usage. When run on all 731 programs with a timeout of 60 seconds, the whole test run takes around 17:55 minutes, because some of the more difficult programs hit the 60 second timeout and fail. This is around 1.5 seconds average for each test, with a maximum of 60 seconds for those that time out, which is still reasonable.

Another interesting performance issue is that we run Boogie with the **-useArrayTheory** flag by default—this uses the built-in SMT theory of arrays within Z3, which handles large arrays better, usually gives better performance, and enables more programs to be verified (without this flag, Boogie can verify only 665/731 = 91% of the valid test suite). However, there are a few programs (e.g. **While_Valid_71.whiley**) where performance becomes dramatically *worse* with this flag—it takes 4.5 minutes to report 5 unverifiable proof obligations with the flag, but less than one second to finish and report 7 unverifiable proof obligations without the flag.

The Whiley native verifier takes only four minutes to process the 600+ test programs that it can verify (around 2.5 programs/sec), which is significantly faster than the Boogie verifier, but takes 18:32 minutes to process all the 731 valid tests (around 1.5 secs/test on average). However, it is difficult to compare the actual proof times, because the Whiley verifier runs within a single Java JVM process, whereas the Wy2B+Boogie toolchain creates several separate processes and intermediate files for each test program.

4.2 Case Study: Conway Game of Life

The first case study we discuss is an interactive web page for playing the Game of Life by Conway [70]. This consists of a small `index.html` file to load the game, plus three Whiley modules:

- `model.whiley` (141 lines): defines the 2D board and the logic of the game;
- `view.whiley` (26 lines): defines how to draw the board onto an HTML canvas;
- `main.whiley` (87 lines): defines mouse event handlers and other controller methods.

The Whiley compiler compiles these three modules and generates JavaScript as output, which can then run in a standard web browser (see Figure 9). We focussed on verifying just the model component, since the others are just the view and controller components whose correct functioning is generally obvious by the visual updates of the canvas. We aimed to specify and verify as much of the functional behaviour of the model as possible, to try to explore the limits of the Boogie verification path. Figure 10 shows the main data structure that represents the board, plus a Whiley function that counts the number of neighbouring cells that are alive.

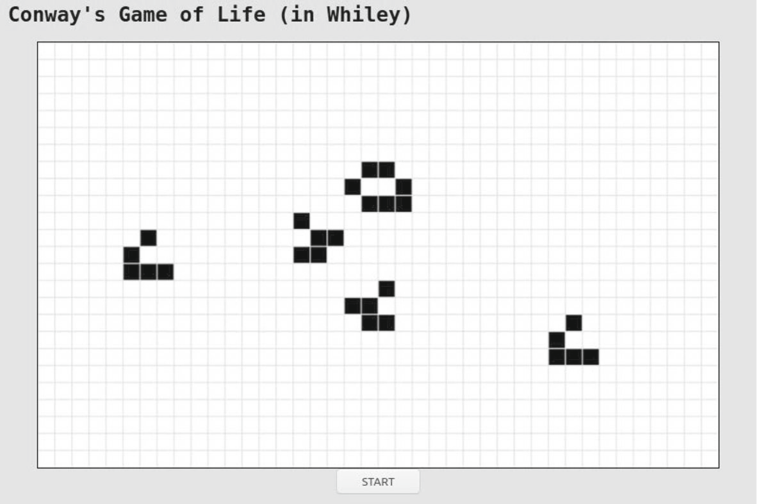

Fig. 9 Illustrating the web-based implementation of Conway's Game of Life developed in Whiley

"Appendix A" gives the full listing of `model.whiley` plus links to the corresponding output Boogie code.

In addition to adding specifications to `model.whiley`, we made some small changes to the code to make specification or verification easier:

- The original board `init` function took `width` and `height` inputs as arbitrary pixel sizes, but we changed these to be cell counts rather than pixels (since the size in pixels is just a GUI display issue) and required them to be greater than zero to avoid empty board cases that are not interesting in practice;
- We moved the cell-update code out of a doubly nested loop into a separate function, for better modularity and easier specification;
- Whiley currently supports only one-dimensional arrays, so the code implemented the 2D board as a one-dimensional array, where each (x, y) location was translated into an index `x + y*state.width`. We respected this data representation choice,[7] but initially had some difficulty with Boogie struggling to verify in-range assertions about these indexes, due to the nonlinear multiplication (`state.width` is initialised at the start of each game, so is not a static constant). Frequently, Boogie would go into an infinite loop trying to prove these assertions (or terminate with a timeout error if we set a time limit). Eventually we found that upgrading Z3 from version 4.8.9 to 4.8.10 solved most of these problems, and Boogie was then able to prove most of the required assertions, or give a quick failure result for those it could not prove. Even then, we found that it was sometimes necessary to try several different ways of specifying indexes and bounds before finding one that Boogie could verify. For example, it was much easier to verify the `count_living(...)` function when it took a single `index` parameter rather than separate `x` and `y` parameters—this is why in our final version the `count_living` function re-derives the `x` and `y` coordinates from the index. This meant that only one variable needed to be quantified in the `update` loop invariant, instead of both the

[7] An alternative would be to use an array of arrays, but we did not explore this option as we wanted to leave the code relatively unchanged.

```
/**
 * Define the board state with the requirement that the width and
 * height match the length of the cells array.
 */
public type State is {
    bool[] cells,
    uint width,
    uint height
} where |cells| == width * height && width > 0 && height > 0

/**
  * Count the number of living cells surrounding a given cell on the board.
  * Since there are at most eight neighbouring cells for any given cell, the result can
  * be at most eight.  Cells on the edge of the board are assumed to be next to dead cells.
  */
public function count_living(uint index, State state) -> (uint r)
requires index < |state.cells|
ensures r <= 8:
    int x = index % state.width
    int y = index / state.width
    //
    uint count = alive(x-1,y-1,state)
    count = count + alive(x-1,y,state)
    count = count + alive(x-1,y+1,state)
    count = count + alive(x,y-1,state)
    count = count + alive(x,y+1,state)
    count = count + alive(x+1,y-1,state)
    count = count + alive(x+1,y,state)
    count = count + alive(x+1,y+1,state)
    return count
```

Fig. 10 Snippets from the Game of Life case study: the State data structure with its invariants, and the count_living function that counts how many neighbouring cells are alive. As explained in the text, the `index` input of count_living is given a cell location (x, y) as `x+y*width`. Note that `uint` is defined in the standard library and has the same definition as `nat` (recall Figure 1)

`x` and `y` coordinates. Typically, we found that cases where Boogie did not terminate were due to array accesses that it couldn't prove were within bounds, and that adding redundant constraints to the specification to make it clear that they were in bounds would fix that problem. This process was rather frustrating, but reflects a limitation of SMT solvers (because nonlinear arithmetic is not decidable) rather than of the Whiley-to-Boogie translation.

After these changes, Boogie (v2.8.26) can easily verify all the functions in this program in 2.2 seconds, plus 2.8 seconds for the translation from Whiley to Boogie.

4.3 VerifyThis 2019 Competition Challenges

In this section, we briefly discuss our experience of translating and verifying several of the Dafny and JML solutions to the 'VerifyThis 2019' verification challenge [59].[8] These

[8] See the VerifyThis 2019 archive, https://www.pm.inf.ethz.ch/research/verifythis/Archive/2019.html, "Sample solutions by the organizers" in Dafny for Challenge 1 and the "OpenJML: David Cok" solution for Challenge 2.

challenges involve quite sophisticated algorithms, with full specifications of the functional behaviour, so are reasonably challenging verification tasks.

Dafny and Boogie were designed to work together, whereas Whiley was independently designed, and originally used several generations of custom-built "native" provers to discharge proof obligations. It is only recently that we have developed the Boogie back end as an alternative verifier. So a useful way to evaluate the usability of the Whiley+Boogie verifier is to take verification solutions that are written in Dafny, translate them into Whiley and see how well the verification works in comparison with Dafny+Boogie. This can help us to understand how various language features of Whiley help or hinder the verification process and how well Whiley translates into the underlying Boogie verifier, which is a common back end for both languages.

We translated and verified the following challenge solutions using Boogie v2.9.6.0 and Z3 v4.8.12—the resulting Whiley solutions can be seen on GitHub.[9]

Challenge 1A: Monotonic Segments This challenge takes an array and cuts it into monotonic segments, which are either increasing or decreasing. The Dafny solution used the built-in extensible sequences to specify some of the operations, but Whiley has only fixed size arrays. So to replicate Dafny's sequence append operator, we defined in Whiley an `append()` function that adds an element to the end of an array. To replicate the functionality of Dafny's sequence slicing, we added `start` and `end` parameters to each of the properties used where necessary, as these were the only uses of sequence slicing in the Dafny version. Interestingly, the lemmas found in the Dafny version were not necessary, as they were needed to prove properties of Dafny's sequence manipulations that were not relevant to Whiley. Some assertions found in the Dafny version were also not needed in the Whiley code, as they were only needed to prove properties of the sequence manipulation in Dafny. The Dafny solution was 72 lines of non-comment specification and source code (excluding curly braces), while the Whiley solution was slightly shorter at 56 lines (including 11 lines for `append<T>()`) and took roughly 30s to verify without the `-useArrayTheory` flag. We note, with that flag enabled, it would not verify the program within 20 minutes.

Challenge 1B: GHC Sort. This challenge was to verify a sorting algorithm used by the GHC Haskell compiler, which takes the monotonic segments from the previous challenge, reverses the decreasing ones, and then pairwise merges the segments into a sorted result. For this challenge, we added the same `append<T>()` function as above. As `01_ghc_sort` builds upon `01_findcuts`, the same start-and-end modifications to the properties were made, but a separate `slice` function was also added, as Dafny's sequence slicing was used more extensively than in the cutpoints solution. The lemmas from the Dafny code were not needed in the Whiley code, and neither were any of the assertions. New assertions were necessary to add to the Whiley code in the `merge_pair` and `monotonic_segments` functions to demonstrate properties of the implemented slice function. The `ghc_sort` function was simplified, as the Dafny solution uses an extra while loop to copy its output sequence into an array, which is not necessary in Whiley as arrays are used throughout. The `reverse` function was re-implemented slightly to avoid an `append()` on every iteration. The Dafny language includes multi-sets (bags) and the Dafny solution used these to prove that one sequence is a permutation of the other. However, the authors comment the "specification (and hence proofs) that the output is a permutation of the input is incomplete". Whiley does not have built-in support for multi-

[9] See https://github.com/DavePearce/VerificationBenchmarks/tree/master/Competitions/VerifyThis19.

sets, and it is difficult to recreate this using uninterpreted functions. As such, we also did not establish the permutation property in the Whiley version. Overall, the Dafny solution was 137 lines of non-comment specification and source code (excluding curly braces), and the Whiley solution was slightly longer at 152 lines. The Wy2B+Boogie verifier takes roughly 20 seconds to verify this program, and again failed to verify within 20mins with the Boogie **-useArrayTheory** flag.

Challenge 2A: Cartesian Trees. This challenge was to verify a stack-based algorithm for finding the nearest smaller value for each item in an array. There was no Dafny solution, so we started from the OpenJML solution, which has a single function with a doubly nested loop.

For this challenge, it was only necessary to add the loop invariants `|left| == |s|` and `|stack| == |s|` to the outer loop, as Whiley is not able to automatically determine that the sizes of the arrays are unmodified when the loop body only updates valid indexes in the array, whereas OpenJML can infer this invariant.

The OpenJML solution was 38 lines of code and specifications, and the Whiley solution is 35 lines. The Wy2B+Boogie verifier takes 1.8 seconds to verify this program, or 5 seconds if we add the Boogie **-useArrayTheory** flag.

4.4 Discussion

From our case studies and our micro-test results, we have observed that using Boogie to verify Whiley programs has significantly increased the verification abilities of Whiley. This is partly due to Boogie making it easier to provide proof support for a wider range of Whiley language features and partly due to the maturity and power of the underlying proof tools—the decades of careful proof engineering that have gone into Z3.

However, the use of Boogie and Z3 is not yet perfect. The Boogie **-useArrayTheory** flag is necessary in some case studies to handle large arrays, but in other case studies it can lead to vastly increased proof times or even non-termination. Also, we have observed that Boogie can often make effective use of recursive predicates to prove a valid proof obligation, but can go into an infinite unfolding loop if that proof obligation is difficult or unprovable.

On the Whiley side, we found that when reasoning about arrays it is helpful to define various supporting properties, such as taking slices of an array, appending two arrays, and counting the occurrences of a given element. It would be useful to develop a Whiley library of these supporting properties and this would be easier if Whiley properties could return arbitrary values, rather than being limited to Boolean results. This would allow them to be used as *specification-only* functions, which would make it easier for Boogie to reason about the domain-specific concepts that are captured by those functions.

5 Related Work

We now consider various tools with similar aims to Whiley, including several which also compile to Boogie.

5.1 Extended Static Checkers

The Extended Static Checker for Java (ESC/Java) [68] and its later successor (ESC/Java2) is perhaps one of the most influential tools in the area of verifying compilers [38, 48]. The tool essentially provides a verifying compiler for Java programs whose specifications are given as annotations in a subset of the *Java Modelling Language (JML)* [38, 39, 99]. JML provides a standard notation for expressing contracts in Java, and the following illustrates a simple method in JML which ESC/Java verifies as correct:

```
/*@ requires n >= 0;
  @ ensures \result >= 0;
  @*/
public static int method(int n) {
   int i = 0;
   /*@ maintaining i >= \old(i); */
   while(i < n) { i = i + 1; }
   return i;
}
```

Here, we can see pre- and postconditions are given for the method, along with an appropriate loop invariant. Since `\old(i)` refers to `i` on entry to the loop, we have `\old(i)==0` in this case. Despite some unsoundness (e.g. ignoring arithmetic overflow and unrolling loops a fixed number of times), the tool has been demonstrated in real-world settings. For example, Cataño and Huisman [37] used it to check specifications given for an independently developed implementation of an electronic purse. In addition to ESC/Java, a Runtime Assertion Checker (RAC) was developed for JML [34, 39, 99] as well as various utilities for specification-based testing [32, 42, 170, 171]. Likewise, Krakatoa [67] provided an alternative to ESC/Java for statically verifying Java programs based on the original Why platform. Finally, whilst the development of JML and its associated tooling stagnated somewhat over the last decade, we note more recent efforts through the OpenJML initiative [29, 46, 47, 149].

The approach taken to generating verification conditions in an earlier tool, ESC/Modula-3, was also adopted in ESC/Java [56]. In fact, ESC/Modula-3 was one of the earliest tools to use an intermediate verification language (based on Dijkstra's language of guarded commands [57]) and, in many ways, is Boogie's predecessor. Such a language typically includes assignment, **`assume`** and **`assert`** statements and non-deterministic choice. It is notable that the guarded command language used in ESC/Modula-3 lacked type information and used a similar encoding of types as ours, although Modula-3 has a simpler type system than Whiley. For example, a predicate `isT` was defined for each type to determine whether a given variable was in the type `T`. A similar approach was also taken in Leino's Ecstatic tool, where the subtyping relation was encoded using a `subtype()` predicate [100]. Again, every type was given a membership predicate with specific axioms stating their non-intersection and was contained in what Leino refers to as the *background predicate* and included with each generated verification condition. A key difference from ESC/Modula-3 is that ESC/Java employed a multi-stage process allowing "high-level" guarded command programs to be desugared into a lower-level form. Further refinements were also made with "passive form" which reduced the size of generated verification conditions, and supported unstructured control flow [17].

5.2 Spec#

The Spec# system followed ESC/Java and benefited from many of the insights gained in that project. Spec# added proper support for handling loop invariants [16], for handling safe object initialisation [64] and allowing temporary violations of object invariants through the `expose` keyword [109]. The latter is necessary to address the so-called *packing problem* which was essentially ignored by ESC/Java [15]. Two further improvements meant Spec# was capable of verifying a wider range of programs than ESC/Java: firstly, Spec# incorporated the new Z3 automated theorem prover (as opposed to Simplify) [53]; secondly, Spec# refined the language of guarded commands used in ESC/Java to form Boogie. Boogie was described as an *"effective intermediate language for verification condition generation of object-oriented programs because it lacks the complexities of a full-featured object-oriented programming language"* [14]. In essence, Boogie was a version of the guarded command language from ESC/Java which also supported a textual syntax, type checking, and static analysis for inferring loop invariants. Other important innovations included the ability to specify *triggers* to help guide quantifier instantiation, and the use of trace semantics to formalise the meaning of Boogie [107].

Leino and Schulte [113] provide an excellent account of how Spec# programs are encoded in Boogie, and there is much similarity with that presented here. For example, the heap is modelled using a global variable of type `[ref,name]any` where a special field, `alloc`, tracks whether a location is allocated. Like Whiley, Spec# permits method calls within expressions, and hence, a similar mechanism for safely extracting them is employed. Furthermore, key challenges arise in preserving class invariants across inheritance and ownership relationships. The approach adopted was based on *packing/unpacking* [15] which identify code regions where class invariants are not required to hold.

5.3 Dafny

Dafny [104, 105] is perhaps the most comparable related work to Whiley, and was developed independently at roughly the same time. That said, the goals of the Dafny project are somewhat different. In particular, the primary goal of Dafny is to provide a proof assistant for verifying algorithms rather than, for example, generating efficient executable code (though it does compile to C#). In contrast, Whiley aims to generate code which is, for example, suitable for embedded systems [134, 156]. Dafny is an imperative language with simple support for objects and classes without inheritance and, more recently, traits [1]. Like Whiley, Dafny employs unbound arithmetic and distinguishes between pure and impure functions. Dafny provides algebraic data types (which are similar to Whiley's recursive data types) and supports immutable collection types with value semantics that are primarily used for ghost fields to enable specification of pointer-based programs. Dynamic memory allocation is possible in Dafny, but no explicit deallocation mechanism is given and presumably any implementation would require a garbage collector.

Leino [102] provides a detailed description of how Dafny programs are translated into Boogie, much of which has already been touched upon earlier in this paper. Dafny also supports generic types and, unlike Whiley, dynamic frames [91]. As discussed in §2.1.6, the latter provides a suitable mechanism for reasoning about pointer-based programs. For example, Dafny has been used successfully to verify the Schorr–Waite algorithm for marking reachable nodes in a graph [104]. Finally, Dafny has been used to successfully verify benchmarks from the VSTTE'08 [108], VSCOMP'10 [93], VerifyThis'12 [83] challenges (and more).

Leino and Pit-Claudel [111] characterise the "Butterfly Effect" where minor changes to the program source cause significant instabilities in verification time. The authors argue one reason for this are so-called matching loops where the SMT solver repeatedly instantiates quantifiers or recursive predicates without making actual progress towards either a proof or a contradiction. Their approach is prototyped in Dafny and moves responsibility for trigger selection out of the SMT solver. This enables trigger selection to occur *before* quantifiers are rewritten into lower-level forms (i.e. as necessary for the SMT solver) where important triggers are obscured. Furthermore, whilst the authors don't expect Dafny users to write triggers themselves, they are expected to understand them in order to diagnose verification performance problems.

5.4 Why3

In addition to Boogie, the other main intermediate verification language in use is WhyML [28, 65]. This is part of the Why3 verification platform which is intended to enable a range of different theorem provers to be used in proving correctness, depending on the nature of the program being verified. For example, a short but extremely intricate C program for solving the N-Queens program has been fully verified with the aid of Why3 [66]. This was achieved by abstracting the original program into WhyML, and the proof required the use of three distinct theorem provers to discharge 41 verification conditions. Of these, 35 were discharged automatically by Alt-ERGO [158] or CVC3 [19], whilst the remainder were discharged manually using Coq [24]. Indeed, the authors of Why3 state [28]:

> *The Why3 platform can be used by itself, as some kind of standalone "meta" theorem prover, but the main purpose of Why3 is to be used as an intermediate language.*

WhyML is a first-order language with polymorphic types, pattern matching, inductive predicates, records and type invariants. It has also been used in the verification of C, Java and Ada programs (amongst others). Like Boogie, WhyML provides structured statements (e.g. **while** and **if** statements). In addition, a standard library is included which provides support for different theories (e.g. integer and real arithmetic, sets and maps).

Of note here is the Boogie to WhyML translation developed by Ameri and Furia [4] which, although largely successful, did expose some important mismatches between them. Their primary motivation was the wide support for alternative (even interactive) provers with Why3. The structured nature of WhyML presented some problems in handling Boogie's unstructured branching, and aspects of Boogie's polymorphic maps and bitvectors were problematic. They showed that Why3 could verify 83% of the translated programs with the same outcome as Boogie. However, they also identified three simple Boogie programs which Boogie either did not verify or incorrectly verified. Why3, on the other hand, handles these cases by virtue of its ability to use a wider range of provers. One of the cases, for example, failed to verify because of the way Z3 handles quantifier instantiation through triggers.

As another example, Spark/ADA is a commercially developed verifying compiler building upon Why3 which has seen good industrial uptake [13, 87]. For example, it has been used used for (amongst other things) *space-control systems* [33], *aviation systems* [40], *automobile systems* [81] and *railway systems* [58].

Finally, given our success here using Boogie to verify Whiley programs, we note it would be interesting future work to explore a WhyML back end for verifying Whiley programs as well.

5.5 Viper

Müller *et al.* [125] observed that existing intermediate verification languages (e.g. Boogie or WhyML) do not support separation logics and related permission-based logics. They identify that such systems have a more "higher-order nature" than typical software verification problems, and make extensive use of recursive predicates (which Boogie/Z3 does not support well). They developed an alternative intermediate verification language (Viper) which offers more precise handling of recursive predicates and protects against "infinite unrolling" using a least fixed-point semantic. The tool also supports two back ends, one of which generates an encoding in Boogie. This builds on earlier work looking at the encoding of abstract predicates and abstraction functions in the context of permission-based logics [79]. Here, abstract predicates describe the (potentially infinite) set of access permission a given object has, but this is problematic for an SMT solver which cannot arbitrarily unroll them. To handle this an encoding is employed which "versions" predicates to prevent arbitrary unrolling, along with various tactics to prevent unlimited matching loops.

An example of work utilising Viper is that of Ter-Gabrielyan *et al.* who argue that SMT solvers typically provide limited support for graph reachability problems, which is prohibitive for reasoning about mutable data structures that admit sharing in various forms [157]. By restricting themselves to problems involving acyclic structures of bounded outdegree, they obtained an encoding ammenable to first-order theorem provers which they demonstrated in the context of Viper. Finally, we note that Viper currently acts as the intermediate verification language for Chalice [110, 114], Prusti [8], Nagini [63], VerCors [5, 27] and more.

5.6 VeriFast

VeriFast is a modular program verifier for concurrent and sequential programs written in C and Java, which employs separation logic and fractional permissions to ensure memory safety [85, 86]. The tool comes from a line of work exploring the use of dynamic frames in the context of verification [152–154]. VeriFast is unusual in eschewing the use of quantifiers within specifications. Instead, inductive predicates are provided to model properties that would otherwise be expressed using quantified formulae. VeriFast supports algebraic data types to allow specifications to reason about locations contained in linked structures. Finally, VeriFast has been used to reason about memory safety in JavaCard programs and Linux device drivers [85], and also in the verification of FreeRTOS [124].

5.7 Frama-C

Frama-C [52] provides a set of sound software analyses for the industrial analysis of ISO C99 source code. The system uses the ACSL specification language as a platform on which different solver plugins can operate. For example, different plugins may use different approaches to checking functions meet their specifications, such as abstract interpretation or deductive verification. The ACSL specification language is based loosely upon JML and supports a variant of separation logic through the `\separated` command. An unusual feature of Frama-C (e.g. compared with Dafny or Whiley) is that multiple loop invariants may be specified at different positions within the loop [22].

Volkov *et al.* [165] developed an extension for *lemma functions* (similar to those in Dafny) which enables a more "interactive" style of verification, and applied this to various functions from the Linux Kernel. We note that, whilst Whiley lacks specific support for lemmas, a similar effect can be achieved using a `function` with a `void` return. Kosmatov and Signoles illustrate runtime assertion checking with Frama-C which they argue provides useful stepping stone prior to static verification [94]. We note similar findings in the context of an automated testing tool for Whiley [43].

Finally, Frama-C has been applied to a range of real-world problems. For example, it has been used in the context of Air Traffic Management systems to reason about floating point operations and establish bounds on rounding errors [73]. Similarly, AirBus has investigated the use of Frama-C within the context of the DO-178B standard for software in airborne systems and equipment [155]. Frama-C has also been used in the context of IoT devices employing AES encryption [26], for verifying components of Contiki, an open-source operating system for IoT [119], and the Xen hypervisor [143]. It has also been applied to verifying railway software [142] and combined with CBMC [97] for test case generation in the context of automotive controllers [127].

5.8 AutoProof

Eiffel [122] is an influential and widely used language that promotes the idea of "Design by Contract" as a lightweight alternative to formal specification [123].

Tschannen *et al.* characterise the AutoProof verifier for Eiffel as being *auto-active*—meaning it lies somewhere between fully automatic and manual (i.e. interactive) [163]. Here, an automated theorem prover is used (as for Dafny or Whiley) in conjunction with appropriate annotations (e.g. pre-/postconditions, loops invariants, etc.). AutoProof translates Eiffel programs into Boogie which, for example, allows strengthening of postconditions and weakening of preconditions in subclasses [162]. Of relevance here is the approach to framing. Whilst Eiffel has no specific syntax for framing, a `modifies` was implemented as a pragma. A default frame condition is employed which assumes only references mentioned in the postcondition can be modified. This utilises a similar rule to that in §3.5 for state preservation across method calls.

Finally, we note AutoProof has been used in various settings, such as for teaching a graduate course on software verification [69].

5.9 Other

Aside from various descriptions of Boogie's syntax and semantics [14, 103, 112], several works focus on the usability of Boogie as an intermediate verification language. For example, Chen and Furia were concerned with the "brittleness" of verification tools [41]. Specifically, a verifier is brittle if small (inconsequential) changes can have major impacts on the outcome (e.g. it no longer verifies). They investigated this in the context of Boogie by mutating various (verified) programs in ways that preserved correctness finding, perhaps surprisingly, several issues. For example, where the ordering of declarations in a Boogie program affected the chance of success. Indeed, for one program consisting of five (independent) statements, Boogie only managed to verify half of the 120 possible orderings. In a similar vein, the Boogie verification debugger (BDV) addresses the disconnect between counterexamples generated at the Boogie level and the source language above [74]. The tool employs plugins to convert Boogie counterexamples into a form recognisable in the source language, with plugins provided for VCC and Dafny. Likewise, Boogaloo attempts to improve the process of debugging failed verification attempts by generating concrete inputs (e.g. arguments) that illustrate the failing trace [141]. A key challenge was in providing a runtime semantics for

Boogie, and for input generation, a mixture of symbolic execution and constraint solving with Z3 was applied.

Segal and Chalin [150] attempted a systematic comparison of Boogie and Pilar. Here, Pilar is a component of the open-source Sireum framework and is similar in many ways to Boogie. They stated that it is *"not trivial to define a common intermediate language that can still support the syntax and semantics of many source languages"*. Their research method was to develop translations from Ruby into both Boogie and Pilar, and then compare. Various aspects of Ruby proved challenging for Boogie, including its dynamically typed nature and arrays. Their solution bears similarity to ours, as they defined an abstract Boogie type as the root of all Ruby values. Overall, they concluded that Boogie's type system makes it "*more flexible for languages with non-traditional type systems*" whereas Pilar is more suitable for traditional Object-Oriented languages.

Arlt *et al.* [7] presented a translation from SOOT's intermediate bytecode language (Jimple) to Boogie, with an aim of identifying unreachable code. As such, an important aspect of the translation was the preservation of feasible execution paths. Overall, they found many aspects of the translation straightforward. For example, Java's `instanceof` operator was modelled using an uninterpreted function. Another interesting aspect of their translation was the use of multiple typed heaps (a Burstall–Bornat heap [30]) to model the Java heap. However, some aspects of impedance mismatch were present and they had difficulty with monitor bytecodes, exceptions, certain chains of **if–else** statements and **finally** blocks.

On a related note, Cook *et al.* [50] focus on the impedance mismatch, arguing that *"existing theorem provers, such as Simplify, lack precise support for important programming language constructs such as pointers, structures and unions"*. For example, that integer types are almost never unbounded in practice, though verification tools often assume this. Likewise, that the lack of support for nonlinear arithmetic is often a problem (though we note useful advances have been made in the intervening years [23, 44, 51, 88, 96]). Their tool, Cogent, "implements an eager and accurate translation of ANSI-C expressions (including features such as bitvectors, structures, unions, pointers and pointer arithmetic) into propositional logic". It is in essence a layer that sits above tools like Boogie and encodes ANSI-C data types using bitvectors and we note the obvious similarity with the more recent tool, Frama-C, discussed above.

Rust provides another interesting perspective as there has also been growing interest in exploiting its safety guarantees for program verification. For example, RustHorn, translates Rust programs into *Constrained Horn Clauses (CHC)* which can then be discharged by a specialised CHC solver [120]. Likewise, Astrauskas *et al.* leverage Rust's type system to simplify the specification and verification of systems software [8]. Their tool, Prusti, extends Rust with a specification language embedded using annotations and statically checked using Viper [125]. The SMACK verifier which translates LLVM IR to Boogie/Z3 [14, 53] was also extended to Rust [11]. The CRUST tool [161] enables unsafe code to be checked using the C Bounded Model Checker (CMBC) [97]. This employs a custom C code generator for `rustc`, and correctly identified bugs arising during development of Rust's standard library. The widely used symbolic execution tool, Klee [35], was also extended for Rust allowing assertions to be checked statically [115, 116]. Finally, we note ongoing work to formalise subsets of the Rust language which could assist the development of verification tools [89, 90, 136, 166, 167].

Finally, Jahob supports multiple provers and is concerned with recursive data structures (e.g. trees, etc.) and their encoding in first-order logic [31, 147]. Bannwart and Müller presented a Hoare-style logic for a sequential bytecode language similar to JVM Bytecode or MSIL [10]. As expected, the unstructured nature of bytecode languages presented a key challenge here. In a similar fashion, Barnett and Leino consider the problem of translating MSIL bytecode into a form suitable for Boogie, in particular by turning unstructured loops into quantified expressions [18].

6 Conclusion

Using Boogie as an intermediate verification language eases the development of a verifying compiler, particularly as it handles verification condition generation, and offers high-level structures such as while loops and procedures with specifications. However, as with any intermediate language, there is potential for an impedance mismatch when Boogie structures do not exactly match the source language. Fortunately, this impedance mismatch can be circumvented in a variety of ways, such as translating to lower-level Boogie statements (e.g. with unstructured control flow). Furthermore, Boogie provides a good level of flexibility to define the "background theory" of a source language, such as its type system, its object structure, and support for heaps. This background theory is at a similar level of abstraction in Boogie as it would be in SMT-LIB so, whilst Boogie offers no major advantages in this area, it also has no disadvantages.

Our work provides a comprehensive account of the encoding of an independently developed non-trivial source language (Whiley) into Boogie. In doing this, we faced many challenges in figuring out a good encoding and, unfortunately, encountered many dead ends along the way. As such, we hope this work can offer guidance to researchers when developing verifying compilers for other languages. Indeed, it would be beneficial to have a repository of knowledge about different ways of encoding various language constructs. Some alternatives (particularly for various heap encoding techniques and procedure framing axioms) are discussed in the published Boogie papers, but there is no central repository of techniques or publications comparing encoding techniques. A major benefit of Boogie is, of course, its easy access to Z3. **We have shown that the Wy2B/Boogie/Z3 stack offers significant advantages over the native Whiley verifier in terms of the percentage of programs that can be verified automatically**. We note, however, that whilst Boogie/Z3 offers tangible benefits, they are not without their own challenges. For example, understanding why Boogie/Z3 cannot verify a particular program, or loops indefinitely, still requires considerable expertise. **We have also shown that a number of non-trivial case studies written in Whiley can be successfully verified with Boogie**. This has also helped us identify areas in which the Whiley language itself could be improved to better exploit Boogie.

Finally, interesting future work would be to explore translating Boogie's *counterexample models* back into Whiley-like notation to improve error reporting. We would also like to extend Whiley's support for framing and, consequently, our Boogie back end. Another interesting path would be a more detailed comparison against the Whiley verifier. As discussed in §3, this has a layered designed based on an intermediate assertion language and an underlying SMT solver. Hence, one could swap out the SMT solver for Z3 to provide a more accurate comparison with Boogie itself.

Acknowledgements The authors would like to thank the various anonymous reviewers of earlier drafts of this paper. They have helped to significantly improve this paper as well as the underlying tool itself. The authors are indebted to their care and consideration. Thanks also to Peter Müller and Rustan Leino for helpful suggestions about the unsoundness issues and ways of resolving them, and to Liam Kent for his work on the case studies.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Appendix A: Conway Game of Life

This shows the Whiley specifications and code for the Conway Game of Life example—just the `src/model.whiley` component that implements the game logic. The full source including the top-level HTML file can be seen in the GitHub repository https://github.com/utting/Conway.wy. The output Boogie code can be seen in the `bin` folder in that repository, with filename `conway.debug.bpl`. Note that we use the `debug=true` flag to turn off name mangling in order to make the generated Boogie code more human-readable.

```
import uint from std::integer

/**
 * Define the board state with the requirement that the width and
 * height match the length of the cells array.
 */
public type State is {
    bool[] cells,
    uint width,
    uint height
} where |cells| == width * height && width > 0 && height > 0

/**
 * Intialise the game in a window with given dimensions.  Since the width and
 * height come from the Canvas properties, assume they are non−negative.
 */
public function init(uint width, uint height) −> (State r)
requires width > 0 && height > 0
ensures r.width == width
ensures r.height == height
ensures |r.cells| == r.width*r.height
ensures all {i in 0..|r.cells| | r.cells[i] == false}:
    return {
        cells: [false; width*height],
        width: width,
        height: height
    }
```

```
/**
 * Click event handler that toggles a square on or off.
 * Since the click locations are generated on the JavaScript side, they could be
 * anything.  However, we know that the State will be valid since it is only
 * ever created and manipulated on the Whiley side.
 */
public function click(int x, int y, State s) −> (State r)
ensures s.height == r.height && s.width == r.width
ensures all {a in 0..s.width, b in 0..s.height | (a + b ∗ s.width < |r.cells|) && (a != x || b != y)
         <==> r.cells[a + b ∗ s.width] == s.cells[a + b ∗ s.width]}
ensures 0 <= x && x < s.width && 0 <= y && y < s.height ==>
         r.cells[x + y ∗ s.width] == !s.cells[x + y ∗ s.width]:
   // Check clicked location is within bounds.
   if x >= 0 && y >= 0 && x < s.width && y < s.height:
      int index = x + (y ∗ s.width)
      s.cells[index] = !s.cells[index]
   //
   return s

/**
 * Defines the rules for updating a single cell.
 */
public property updated_cell(int index, State s, bool out) where
   0 <= index && index < |s.cells| &&
   (out <==> (count_living((uint) index, s) == 3
        || (s.cells[index] && count_living((uint) index,s) == 2)))

/**
 * Update the game based on the current arrangement of live cells.
 * This applies the three rules of Conway's game of life to either
 * kill cells or to create new cells.
 */
public function update(State state)−> (State r)
ensures all {j in 0..|r.cells| | updated_cell(j, state, r.cells[j]) }
ensures state.width == r.width && state.height == r.height:
   // Create copy of cells array
   bool[] ncells = state.cells
   // Iterate through all cells
   for y in 0..state.height
   where |ncells| == |state.cells|
   where all {j in 0..(y∗state.width) | j < |ncells| && j >= 0 && ncells[j] == update_cell(j,state)}:
      for x in 0..state.width
      where |ncells| == |state.cells|
      where all {j in 0..(x + y∗state.width)| j < |ncells| && j >= 0 && ncells[j] == update_cell(j,state)}:
         int i = x + y∗state.width
         ncells[i] = update_cell(i, state)
   // Switch over new cells array
   state.cells = ncells
   // Done
   return state
```

```
/**
 * Calculate a cell update according to Conway’s game of life rules.
 */
public function update_cell(int index, State state) −> (bool out)
requires index >= 0 && index < |state.cells|
ensures updated_cell(index, state, out):
    uint count = count_living((uint) index, state)
    if state.cells[index]:
        switch count:
            case 0,1:
                // Any live cell with fewer than two live neighbours dies,
                // as if caused by under-population.
                return false
            case 2,3:
                // Any live cell with two or three live neighbours lives
                // on to the next generation.
                return true
            case 4,5,6,7,8:
                // Any live cell with more than three live neighbours dies,
                // as if by overcrowding.
                return false
    else if count == 3:
        // Any dead cell with exactly three live neighbours becomes alive, as if by reproduction.
        return true
    // Other dead cells remain dead
    return false

/**
 * Count the number of living cells surrounding a given cell on the board.
 * Since there are at most eight neighbouring cells for any given cell, the result can
 * be at most eight.  Cells on the edge of the board are assumed to be next to dead cells.
 */
public function count_living(uint index, State state) −> (uint r)
requires index < |state.cells|
ensures r <= 8:
    int x = index % state.width
    int y = index / state.width
    uint count = alive(x−1,y−1,state)
    count = count + alive(x−1,y,state)
    count = count + alive(x−1,y+1,state)
    count = count + alive(x,y−1,state)
    count = count + alive(x,y+1,state)
    count = count + alive(x+1,y−1,state)
    count = count + alive(x+1,y,state)
    count = count + alive(x+1,y+1,state)
    return count

/**
 * Determine whether a given cell is alive or not.  This returns an integer for
 * convenience when implementing the count_living function above.
 */
public function alive(int x, int y, State state) −> (uint r)
ensures (r == 0 || r == 1)
ensures r == 0 <==> (x < 0 || x >= state.width || y < 0 || y >= state.height ||
                    some {i in 0..|state.cells| | i == x + y∗state.width && !state.cells[i]}):
    if x < 0 || x >= state.width || y < 0 || y >= state.height || !state.cells[x + y∗state.width]:
        return 0
    else:
        return 1
```

References

1. Ahmadi, R., Leino, K.R.M., Nummenmaa, J.: Automatic verification of Dafny programs with traits. In: Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP), pp. 4:1–4:5. ACM Press (2015)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book—From Theory to Practice, LNCS, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Altidor, J., Huang, S.S., Smaragdakis, Y.: Taming the wildcards: combining definition- and use-site variance. In: Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI), pp. 602–613. ACM Press (2011)
4. Ameri, M., Furia, C.A.: Why just Boogie? Translating between intermediate verification languages. In: Proceedings of the Conference on Integrated Formal Methods (iFM), pp. 79–95 (2016)
5. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: International School on Formal Methods (SFM), LNCS, vol. 8483, pp. 172–216. Springer-Verlag (2014)
6. Appel, A.W.: Program Logics—for Certified Compilers. Cambridge University Press, Cambridge (2014)
7. Arlt, S., Rümmer, P., Schäf, M.: Joogie: from Java through Jimple to Boogie. In: Proceedings of the Workshop on State of the Art in Java Program analysis (2013)
8. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), p. Article 147. ACM Press (2019)
9. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 387–411. Springer-Verlag (2008). https://doi.org/10.1007/978-3-540-70592-5_17
10. Bannwart, F., Müller, P.: A program logic for bytecode. Electron. Notes Comput. Sci. **141**(1), 255–273 (2005)
11. Baranowski, M., He, S., Rakamarić, Z.: Verifying Rust programs with SMACK. In: Automated Technology for Verification and Analysis, pp. 528–535. Springer-Verlag (2018)
12. Barbanera, F., Caglini, M.D.C.: Intersection and union types. In: Proceedings of the Conference on Theoretical Aspects of Computer Software (TACS), pp. 651–674 (1991)
13. Barnes, J.: High Integrity Ada: The SPARK Approach. Addison Wesley Longman, Inc., Reading (1997)
14. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of the Formal Methods for Components and Objects (FMCO), pp. 364–387 (2006)
15. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. J. Object Technol. **3**(6), 27–56 (2004)
16. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
17. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE), pp. 82–87. ACM Press (2005)
18. Barnett, M., Leino, K.R.M.: To goto where no statement has gone before. In: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE), LNCS, vol. 6217, pp. 157–168. Springer-Verlag (2010)
19. Barrett, C., Tinelli, C.: CVC3. In: Proceedings of Conference on Computer Aided Verification (CAV), pp. 298–302 (2007)
20. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of Conference on Computer Aided Verification (CAV), *LNCS*, vol. 6806, pp. 171–177. Springer-Verlag (2011)
21. Barrett, C.W., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, Scotland, (SMT '10) (2010)
22. Baudin, P., Cuo, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language (version 1.8)
23. Beame, P., Liew, V.: Toward verifying nonlinear integer arithmetic. J. ACM **66**(3), 22:1-22:30 (2019)
24. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer-Verlag (2004)
25. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 113–132. ACM Press (2012)

26. Blanchard, A., Kosmatov, N., Loulergue, F.: A lesson on verification of IoT software with Frama-C. In: Proceedings of the Conference on High Performance Computing & Simulation (HPCS), pp. 21–30. IEEE (2018)
27. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Proceedings of the Symposium on Formal Methods (FM), vol. 8442, pp. 127–131. Springer-Verlag (2014)
28. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Workshop on Intermediate Verification Languages (2011)
29. Boerman, J., Huisman, M., Joosten, S.J.C.: Reasoning about JML: Differences between KeY and OpenJML. In: Proceedings of the Conference on Integrated Formal Methods (iFM), LNCS, vol. 11023, pp. 30–46. Springer-Verlag (2018)
30. Bornat, R.: Proving pointer programs in Hoare logic. In: Proceedings of the Conference on the Mathematics of Program Construction (MPC) (2000)
31. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.C.: Using first-order theorem provers in the Jahob data structure verification system. In: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), pp. 74–88 (2007)
32. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 123–133 (2002). https://doi.org/10.1145/566171.566191
33. Brandon, C., Chapin, P.: A SPARK/Ada cubesat control program. In: Proceedings of the Conference on Reliable Software Technologies (RST), pp. 51–64 (2013)
34. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Electron. Notes Comput. Sci. **80**, 75–91 (2003)
35. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the Conference on Operating Systems Design and Implementation (OSDI), pp. 209–224 (2008)
36. Cardelli, L.: Structural subtyping and the notion of power type. In: Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL), pp. 70–79. ACM Press (1988)
37. Cataño, N., Huisman, M.: Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In: Proceedings of the Symposium on Formal Methods Europe (FME), *LNCS*, vol. 2391, pp. 272–289. Springer-Verlag (2002)
38. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: Proceedings of the Formal Methods for Components and Objects (FMCO), pp. 342–363 (2005)
39. Chalin, P., Rioux, F.: JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In: Proceedings of the Symposium on Formal Methods (FM), LNCS, vol. 5014, pp. 246–261. Springer-Verlag (2008)
40. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Proceedings of the Conference on Interactive Theorem Proving (ITP), pp. 17–26 (2014)
41. Chen, Y., Furia, C.A.: Robustness testing of intermediate verifiers. In: Proceedings of the Conference on Automated Technology for Verification and Analysis (ATVA), LNCS, vol. 11138, pp. 91–108. Springer (2018)
42. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 231–255. Springer (2002). https://doi.org/10.1007/3-540-47993-7_10
43. Chin, J., Pearce, D.J.: Finding bugs with specification-based testing is easy! The Art, Science, and Engineering of Programming **5**, Article 13 (2021)
44. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: Proceedings of Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 10929, pp. 383–398. Springer-Verlag (2018)
45. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOL), LNCS, vol. 5674, pp. 23–42. Springer-Verlag (2009)
46. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Proceedings of the NASA Formal Methods Symposium, LNCS, vol. 6617, pp. 472–479. Springer-Verlag (2011)
47. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Proceedings of the Workshop on Formal Integrated Development Environment (F-IDE), vol. 149, pp. 79–92 (2014)
48. Cok, D.R., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Proceedings of the Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), pp. 108–128 (2005)

49. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-ergo 2.2. In: Workshop on Satisfiability Modulo Theories (SMT). HAL CCSD (2018)
50. Cook, B., Kroening, D., Sharygina, N.: Accurate theorem proving for program verification. In: Proceedings of the Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), pp. 96–114 (2004)
51. Corzilius, F., Loup, U., Junges, S., Ábrahám, E.: SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox - (tool presentation). In: Proceedings of Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 7317, pp. 442–448. Springer-Verlag (2012)
52. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Proceedings of the Conference on Software Engineering and Formal Methods (SEFM), LNCS, vol. 7504, pp. 233–247. Springer-Verlag (2012)
53. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 337–340 (2008)
54. de Moura, L.M., Bjørner, N.: Efficient E-matching for SMT solvers. In: Proceedings of the Conference on Automated Deduction (CADE), pp. 183–198 (2007)
55. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
56. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. SRC Research Report 159, Compaq Systems Research Center (1998)
57. Dijkstra, E.W.: Guarded commands, nondeterminancy and formal derivation of programs. Commun. ACM **18**, 453–457 (1975)
58. Dross, C., Efstathopoulos, P., Lesens, D., Mentré, D., Moy, Y.: Rail, space, security: three case studies for spark 2014. In: Proceedings of the Congress on Embedded Real Time Systems (ERTS) (2014)
59. Dross, C., Furia, C.A., Huisman, M., Monahan, R., Müller, P.: VerifyThis 2019: A program verification competition (extended report). CoRR arXiv:2008.13610 (2020)
60. Dubochet, G., Odersky, M.: Compiling structural types on the JVM: a comparison of reflective and generative techniques from scala's perspective. In: Proceedings of the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS), pp. 34–41. ACM Press (2009)
61. Dutertre, B.: Yices 2.2. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 8559, pp. 737–744. Springer-Verlag (2014)
62. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proceedings of Conference on Computer Aided Verification (CAV), pp. 81–94 (2006)
63. Eilers, M., Müller, P.: Nagini: A static verifier for Python. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 10981, pp. 596–603. Springer-Verlag (2018)
64. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 302–312. ACM Press (2003)
65. Filliâtre, J., Paskevich, A.: Why3—where programs meet provers. In: Proceedings of the European Symposium on Programming (ESOP), pp. 125–128 (2013)
66. Filliâtre, J.C.: Verifying two lines of C with Why3: An exercise in program verification. In: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE), LNCS, vol. 7152, pp. 83–97. Springer-Verlag (2012)
67. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 4590, pp. 173–177. Springer-Verlag (2007)
68. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
69. Furia, C.A., Poskitt, C.M., Tschannen, J.: The AutoProof verifier: usability by non-experts and on standard code. In: Proceedings of the Workshop on Formal Integrated Development Environment (F-IDE) (2015)
70. Games, M.: The fantastic combinations of John Conway's new solitaire game "life" by Martin Gardner. Scientific American **223**, 120–123 (1970)
71. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Proceedings of Conference on Computer Aided Verification (CAV), pp. 306–320 (2009)
72. Gil, J., Maman, I.: Whiteoak: introducing structural typing into Java. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 73–90. ACM Press (2008)

73. Goodloe, A.E., Muñoz, C., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: Proceedings of the NASA Formal Methods Symposium, LNCS, vol. 7871, pp. 441–446. Springer-Verlag (2013)
74. Goues, C.L., Leino, K.R.M., Moskal, M.: The Boogie Verification Debugger (tool paper). In: Proceedings of the Conference on Software Engineering and Formal Methods (SEFM), LNCS, vol. 7041, pp. 407–414. Springer-Verlag (2011)
75. Gries, D.: The multiple assignment statement. IEEE Trans. Softw. Eng. **4**, 89–93 (1978)
76. Gries, D.: The Science of Programming. Springer-Verlag, Berlin (1981)
77. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: Proceedings of the European Symposium on Programming (ESOP), pp. 256–275 (2011)
78. Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In: Proceedings of the European Symposium on Programming (ESOP), vol. 3924, pp. 279–293 (2006)
79. Heule, S., Kassios, I.T., Müller, P., Summers, A.J.: Verification condition generation for permission logics with abstract predicates and abstraction functions. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 451–476 (2013)
80. Hoare, C.: The verifying compiler: a grand challenge for computing research. J. ACM **50**(1), 63–69 (2003)
81. Hocking, A.B., Knight, J.C., Aiello, M.A., Shiraishi, S.: Arguing software compliance with ISO 26262. In: Proceedings of the Symposium on Software Reliability Engineering (ISSRE), pp. 226–231. IEEE Computer Society (2014)
82. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in Vampire. In: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 6605, pp. 60–64. Springer-Verlag (2011)
83. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis verification competition 2016—organizer's report (2013)
84. Igarashi, A., Nagira, H.: Union types for object-oriented programming. J. Obj. Technol. **6**(2) (2007)
85. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Proceedings of the NASA Formal Methods Symposium, pp. 41–55 (2011)
86. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS), pp. 304–311 (2010)
87. Jennings, T.J., Carré, B.A.: A subset of Ada for formal verification (SPARK). Ada User **9**(Supplement), 121–126 (1989)
88. Jovanovic, D.: Solving nonlinear integer arithmetic with MCSAT. In: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 10145, pp. 330–346. Springer-Verlag (2017)
89. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for Rust. In: Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL), p. Article 41 (2020)
90. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the Rust programming language. In: Proceedings of the ACM Symposium on the Principles Of Programming Languages (POPL), pp. 66:1–66:34. ACM Press (2018)
91. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Proceedings of the Symposium on Formal Methods (FM), LNCS, vol. 4085, pp. 268–283. Springer-Verlag (2006)
92. Kassios, I.T.: Dynamic frames and automated verification. Tech. Rep. Tutorial for the. 2nd COST Action IC0701 Training School (2011)
93. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report (VSComp). In: Proceedings of the Symposium on Formal Methods (FM), LNCS. Springer-Verlag (2011)
94. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: Proceedings of the Conference on Runtime Verification (RV), LNCS, vol. 8174, pp. 386–399. Springer-Verlag (2013)
95. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 8044, pp. 1–35. Springer-Verlag (2013)
96. Kremer, G., Corzilius, F., Ábrahám, E.: A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In: Computer Algebra in Scientific Computing (CASC), LNCS, vol. 9890, pp. 315–335. Springer-Verlag (2016)

97. Kroening, D., Tautschnig, M.: CBMC - C Bounded Model Checker. In: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 8413, pp. 389–391. Springer-Verlag (2014)
98. Lameed, N., Hendren, L.J.: Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In: Proceedings of the Confererence on Compiler Construction (CC), pp. 22–41 (2011)
99. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. **55**(1–3), 185–208 (2005)
100. Leino, K.R.M.: Ecstatic: An object-oriented programming language with an axiomatic semantics. In: Workshop on Foundations of Object-Oriented Languages (FOOL 4) (1997)
101. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93**(6), 281–288 (2005)
102. Leino, K.R.M.: Specification and verification of object-oriented software. Technical Report, Marktoberdorf Internation Summer School (2008)
103. Leino, K.R.M.: This is Boogie 2 (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/. Microsoft Research, Manuscript KRML 178
104. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), LNCS, vol. 6355, pp. 348–370. Springer-Verlag (2010)
105. Leino, K.R.M.: Leino, K.R.M.: Developing verified programs with Dafny. In: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE), LNCS, vol. 7152, pp. 82. Springer-Verlag (2012)
106. Leino, K.R.M.: Program proving using intermediate verification languages (IVLs) like Boogie and Why3. In: Proceedings of the Workshop on Safe Languages and Technologies for Structured and Efficient Parallel and Distributed/Cloud Computing (HILT), pp. 25–26 (2012)
107. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the Symposium on Applied Computing (SAC), pp. 615–622 (2009)
108. Leino, K.R.M., Monahan, R.: Dafny meets the verification benchmarks challenge. In: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE) LNCS, vol. 6217, pp. 112–126. Springer-Verlag (2010)
109. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: LASER Summer School, LNCS, vol. 6029, pp. 91–139. Springer-Verlag (2008)
110. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Foundations of Security Analysis and Design V (FOSAD), pp. 195–222 (2009)
111. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 9779, pp. 361–381. Springer-Verlag (2016)
112. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 6015, pp. 312–327. Springer-Verlag (2010)
113. Leino, K.R.M., Schulte, W.: A verifying compiler for a multi-threaded object-oriented language. In: Summer School Marktoberdorf 2006: Software System Reliability and Security, NATO ASI Series F. IOS Press, Amsterdam (2007)
114. Leino, K.R.M., Yessenov, K.: Stepwise refinement of heap-manipulating code in chalice. Formal Aspects of Computing **24**(4-6) (2012)
115. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of Rust programs by symbolic execution. In: International Conference on Industrial Informatics (INDIN), pp. 108–114 (2018)
116. Lindner, M., Fitinghoff, N., Eriksson, J., Lindgren, P.: Verification of safety functions implemented in Rust - a symbolic execution based approach. In: International Conference on Industrial Informatics (INDIN), vol. 1, pp. 432–439 (2019)
117. Malayeri, D., Aldrich, J.: Integrating nominal and structural subtyping. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 260–284 (2008)
118. Malayeri, D., Aldrich, J.: Is structural subtyping useful? An empirical study. In: Proceedings of the European Symposium on Programming (ESOP), LNCS, vol. 5502, pp. 95–111. Springer-Verlag (2009)
119. Mangano, F., Duquennoy, S., Kosmatov, N.: Formal verification of a memory allocation module of contiki with Frama-C: a case study. In: Proceedings of the Conference on Software Engineering and Formal Methods (SEFM), pp. 114–120. Springer-Verlag (2016)
120. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. In: Programming Languages and Systems, pp. 484–514. Springer-Verlag (2020)
121. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015). https://doi.org/10.1017/CBO9781139629294

122. Meyer, B.: Eiffel: a language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988)
123. Meyer, B.: Applying "design by contract". IEEE. Comput. **25**(10), 40–51 (1992)
124. Mühlberg1and, J.T., Freitas, L.: Verifying freertos: from requirements to binary code. In: Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS) (2011)
125. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), pp. 41–62 (2016)
126. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27** (1980)
127. Nguena-Timo, O., Langelier, G.: Test data generation for cyclic executives with CBMC and Frama-C: a case study. Electron. Notes Comput. Sci. **320**, 35–51 (2016)
128. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. **205**(4), 557–580 (2007)
129. Odersky, M.: How to make destructive updates less destructive. In: Proceedings of the ACM Symposium on the Principles Of Programming Languages (POPL), pp. 25–36 (1991)
130. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the Workshop on Computer Science Logic (CSL), pp. 1–19. Springer-Verlag (2001). https://doi.org/10.1007/3-540-44802-0_1
131. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Proceedings of the European Symposium on Programming (ESOP), pp. 439–458 (2011)
132. Pearce, D.J.: A calculus for constraint-based flow typing. In: Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP), p. Article 7 (2013)
133. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. In: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), pp. 335–354 (2013)
134. Pearce, D.J.: Integer range analysis for Whiley on embedded systems. In: Proceedings of the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, pp. 26–33 (2015)
135. Pearce, D.J.: Array programming in Whiley. In: Proceedings of the Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY) (2017). https://doi.org/10.1145/3091966.3091972
136. Pearce, D.J.: A lightweight formalism for reference lifetimes and borrowing in Rust. ACM Trans. Program. Lang. Syst. **43**(1), Article 3 (2021)
137. Pearce, D.J., Groves, L.: Reflections on verifying software with Whiley. In: Proceedings of the Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), pp. 142–159 (2013)
138. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: Proceedings of the Conference on Software Language Engineering (SLE), pp. 238–248 (2013)
139. Pearce, D.J., Groves, L.: Designing a verifying compiler: Lessons learned from developing Whiley. Science of Computer Programming, pp. 191–220 (2015)
140. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Whiley. In: Engineering Trustworthy Software Systems (SETTS), pp. 1–37. Springer-Verlag (2019). https://doi.org/10.1007/978-3-030-17601-3_1
141. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before: Executing an intermediate verification language. In: Proceedings of the Conference on Runtime Verification (RV), pp. 251–268 (2013)
142. Prevosto, V., Burghardt, J., Gerlach, J., Hartig, K., Pohl, H.W., Völlinger, K.: Formal specification and automated verification of railway software with Frama-C. In: International Conference on Industrial Informatics (INDIN), pp. 710–715. IEEE (2013)
143. Puccetti, A.: Static analysis of the XEN kernel using Frama-C. J. Univ. Comput. Sci. **16**(4), 543–553 (2010)
144. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the Conference on Supercomputing, pp. 4–13. IEEE Computer Society Press (1991)
145. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Proceedings of Conference on Computer Aided Verification (CAV), LNCS, vol. 9207, pp. 198–216. Springer-Verlag (2015)
146. Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., Barrett, C.W.: Quantifier instantiation techniques for finite model finding in SMT. In: Proceedings of the Conference on Automated Deduction (CADE), LNCS, vol. 7898, pp. 377–391. Springer-Verlag (2013)
147. Rinard, M.C.: Integrated reasoning and proof choice point selection in the Jahob system - mechanisms for program survival. In: Proceedings of the Conference on Automated Deduction (CADE), LNCS, vol. 5663, pp. 1–16. Springer-Verlag (2009)

148. Robison, P.J., Staples, J.: Formalizing a hierarchical structure of practical mathematical reasoning. J. Log. Comput. **3**(1), 47–61 (1993). https://doi.org/10.1093/logcom/3.1.47
149. Sánchez, J., Leavens, G.T.: Static verification of PtolemyRely programs using OpenJML. In: Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL), pp. 13–18. ACM Press (2014)
150. Segal, L., Chalin, P.: A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE), pp. 130–145 (2012)
151. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Proceedings of the Symposium Logic-Based Program Synthesis and Transformation (LOPSTR), pp. 1–24 (2001)
152. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An automatic verifier for a concurrent object-oriented language. In: Formal Methods for Open Object-Based Distributed Systems (FMOODS), pp. 220–239 (2008)
153. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1-2:58 (2012). https://doi.org/10.1145/2160910.2160911
154. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for Java-like programs based on dynamic frames. In: Proceedings of the Conference on Fundamental Approaches to Software Engineering, LNCS, vol. 4961, pp. 261–275. Springer-Verlag (2008)
155. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of avionics software products. In: Proceedings of the Symposium on Formal Methods (FM), LNCS, vol. 5850, pp. 532–546. Springer-Verlag (2009)
156. Stevens, M.: Demonstrating Whiley on an embedded system. Tech. rep., School of Engineering and Computer Science, Victoria University of Wellington (2014). http://www.ecs.vuw.ac.nz/~djp/files/MattStevensENGR489.pdf
157. Ter-Gabrielyan, A., Summers, A.J., Müller, P.: Modular verification of heap reachability properties in separation logic. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), p. Article 121. ACM Press (2019)
158. The Alt-Ergo automated theorem prover, http://alt-ergo.lri.fr/
159. The Whiley Programming Language. http://whiley.org
160. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Proceedings of the ACM International Conference on Functional Programming (ICFP), pp. 117–128 (2010)
161. Toman, J., Pernsteiner, S., Torlak, E.: Crust: A bounded verifier for rust. In: Proceedings of the Conference on Automated Software Engineering (ASE), pp. 75–80 (2015)
162. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Automatic verification of advanced object-oriented features: The AutoProof approach. In: LASER Summer School, LNCS, vol. 7682, pp. 133–155. Springer (2011)
163. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-active functional verification of object-oriented programs. In: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 9035, pp. 566–580. Springer-Verlag (2015)
164. Utting, M., Pearce, D.J., Groves, L.: Making Whiley Boogie! In: Proceedings of the Conference on Integrated Formal Methods (iFM), pp. 69–84 (2017)
165. Volkov, G., Mandrykin, M.U., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. CoRR arXiv:1811.05879 (2018)
166. Wang, F., Song, F., Zhang, M., Zhu, X., Zhang, J.: Krust: A formal executable semantics of rust. In: Proceedings of the Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 44–51 (2018)
167. Weiss, A., Patterson, D., Matsakis, N.D., Ahmed, A.: Oxide: The essence of rust (2019)
168. Weng, M., Utting, M., Pfahringer, B.: Bound analysis for Whiley programs. In: Proc. Usages of Symbolic Execution Workshop (2015)
169. Weng, M.H., Utting, M., Pfahringer, B.: Bound analysis for Whiley programs. Electron. Notes Comput. Sci. **320**, 53–67 (2016). https://doi.org/10.1016/j.entcs.2016.01.005
170. Xu, G.H., Yang, Z.: JMLAutoTest: A novel automated testing framework based on JML and JUnit. In: Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES), pp. 70–85 (2003)
171. Zimmerman, D.M., Nagmoti, R.: JMLUnit: The Next Generation. In: Proceedings of the Conference on Formal Verification of Object-Oriented Software, pp. 183–197. Springer (2010). https://doi.org/10.1007/978-3-642-18070-5_13