# Making Whiley Boogie!

Mark Utting[1(✉)], David J. Pearce[2], and Lindsay Groves[2]

[1] University of the Sunshine Coast, Sippy Downs, Australia
utting@usc.edu.au
[2] Victoria University of Wellington, Wellington, New Zealand
{david.pearce,lindsay}@ecs.vuw.ac.nz

**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 common-place. 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 report that, whilst a naive translation to Boogie is unsatisfactory, a more abstract encoding is surprisingly effective.

**Keywords:** Whiley · Boogie · Verifying compiler · Intermediate verification language · Semantic translation · Impedance mismatch

## 1 Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been studied for a long time. According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [15]. A variety of tools have blossomed in this space, including ESC/Java [14], Spec# [4], Dafny [18], Why3 [13], VeriFast [16], SPARK Ada [20], and Whiley [26,30]. Automated Theorem Provers are integral to such tools and are responsible for discharging proof obligations [4,7,14,16]. Various Satisfiability Modulo Theory (SMT) solvers are typically used for this, such as Simplify [9] or Z3 [21]. These provide hand-crafted implementations of important decision procedures, e.g. for linear arithmetic [12],


N. Polikarpova and S. Schneider (Eds.): IFM 2017, LNCS 10510, pp. 69–84, 2017.
DOI: 10.1007/978-3-319-66845-1_5

congruence [24] and quantifier instantiation [22]. 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 [3] or WhyML [6,13], as these provide access to a range of different solvers. SMT-LIB [29] provides another standard readily accepted by modern automated theorem provers, although it is often considered rather low-level [6]. One issue faced by intermediate verification languages is the potential for an *impedance mismatch* [26]. This arises when constructs in the source language cannot be easily translated into those of the intermediate verification language. In this paper, we explore Boogie as an intermediate verification language for the Whiley verifying compiler. A particular concern is the potential for an impedance mismatch arising from Whiley's expressive type system which, amongst other things, supports *union*, *intersection* and *negation types* [25]. The obvious translation between Whiley and Boogie is rather unsatisfactory, but with care, a suitable encoding can be found that works surprisingly well.

The contributions of this paper include:

- a novel translation from Whiley programs to Boogie. Whilst in many cases this is straightforward, there are a number of challenges to overcome arising from the impedance mismatch between Whiley and Boogie.
- the results of an empirical comparison between Boogie/Z3 and the native Whiley verifier. The results indicate that using Boogie to verify Whiley programs (via our translation) is competitive with the native Whiley verifier.

## 2 Background

We begin with a brief overview of Whiley and a more comprehensive 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 [26,30]. The Whiley Compiler (WyC) attempts to ensure that all functions in a program meet their specifications. When this succeeds, we know that: (1) all function postconditions are met (assuming their preconditions held on entry); (2) all invocations meet their respective function's precondition; (3) 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. RAM).

Figure 1 provides an interesting example which illustrates many of the salient features of Whiley:

- **Preconditions** are given by **`requires`** clauses and **postconditions** by **`ensures`** clauses. Multiple clauses are simply conjoined together. We found

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

function indexOf(int[] items, int 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 ==> all{i in 0..|items| | items[i] != item}:
    //
    nat i = 0
    while i < |items|
        where all { k in 0 .. i | items[k] != item }:
        //
        if items[i] == item:
            return i
        i = i + 1
    //
    return null
```

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

that allowing multiple **`requires`** and/or **`ensures`** clauses can help readability, and note that JML [8], Spec# [4] and Dafny [18] also permit this.

- **Loop Invariants** are given by **`where`** clauses. Figure 1 illustrates an inductive loop invariant covering indices from zero to `i` (exclusive).
- **Type Invariants** can be included with **`type`** declarations, as illustrated by type `nat`. This is the declared type of variable `i`, meaning no loop invariant of the form `i >= 0` is necessary. We consider good use of type invariants as critical to improving the readability of function specifications.
- **Flow Typing & Unions**. An unusual feature of Whiley is the use of a *flow typing system* [25] coupled with *union types*. This is illustrated by the return type "**`int|null`**" and the use of a type test in the postcondition. Specifically, in the predicate "`x` **`is`** `T ==> e`" it follows that `x` has type `T` within the expression `e`.

### 2.2 Boogie

Boogie [3] is an intermediate verification language developed by Microsoft Research as part of the Spec# project [4]. Boogie is intended as a back-end for other programming language and verification systems [19] and has found use in various tools, such as Dafny [18], VCC [7], and others (e.g. [5]). Boogie is both a specification language (which shares some similarity with Dijkstra's language of guarded commands [11]) 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"* [3]. 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, **function**s are pure and intended to model fundamental operators in the source language. In contrast, **procedure**s 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 **procedure**s 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.

Figure 2 provides an example encoding of the `indexOf()` function in Boogie. At first glance, it is perhaps surprising how close to an actual programming language Boogie has become. Various features of the language are demonstrated

```
1  function len(arr:[int]int) returns (r: int); // array length operator
2
3  axiom (forall A:[int]int :: len(A) >= 0); // no negative length arrays
4
5  procedure indexOf(xs: [int]int, x: int) returns (r: int)
6  ensures r >= 0 && r <= len(xs);
7  ensures (r < len(xs)) ==> (xs[r] == x);
8  ensures (forall k:int :: (0<=k && k<r) ==> xs[k] != x); {
9    var i : int;
10   i := 0;
11   while (i < len(xs))
12   invariant i >= 0 && i <= len(xs);
13   invariant (forall k:int :: (0<=k && k<i) ==> xs[k] != x); {
14     if(xs[i] == x) { break; }
15     i := i + 1;
16   }
17   r := i;
18 }
```

**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

```
1  procedure indexOf(xs: [int]int, x: int) returns (r: int) ... {
2     var i : int;
3     i := 0;
4     assert i >= 0 && i <= len(xs); // assert invariant on entry
5     assert (forall k:int :: (0<=k && k<i)==> xs[k] != x);
6    head:
7     havoc i;
8     assume i >= 0 && i <= len(xs); // assume invariant
9     assume (forall k:int :: (0<=k && k<i)==> xs[k] != x);
10    goto body,exit;
11   body:
12    assume i < len(xs); // assume loop condition
13    if(xs[i] == x) { goto exit; }
14    i := i + 1;
15    assert i >= 0 && i <= len(xs); // assert invariant
16    assert (forall k:int :: (0<=k && k<i)==> xs[k] != x);
17    goto head;
18   exit:
19    assume i >= len(xs); // assume negated condition
20    r := i;
21 }
```

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

with this example. Firstly, an array length operator is encoded using an uninterpreted function `len()`, and accompanying **`axiom`**. Secondly, the input array is modelled using the map **`[int]int`**, the meaning of which is somewhat subtle, in that it is not describing an array as in a programming language. Rather, it 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. 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. 3.2). Figure 3 provides an unstructured encoding of the `indexOf()` function from Fig. 2. In this version, the **`while`** loop is decomposed using a non-deterministic **`goto`** statement. Likewise, 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 Fig. 2). The **`havoc`** statement *"assigns an arbitrary value to each indicated variable"* [3], so is used here to indicate that variable `i` contains an arbitrary integer value at this point.

## 3 Modeling 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 the verification of Whiley programs, hopefully leading to better overall results compared with Whiley's native (and relatively adhoc) verifier. The key challenge here is the impedance mismatch between Whiley and Boogie. Despite their obvious similarities, there remain some significant differences:

- **Types.** Whiley has a relatively rich (structural) type system which includes: *union*, *intersection* and *negation* types.
- **Flow Typing.** Whiley's support for flow typing is also problematic, as a given variable may have different types at different program points [25].
- **Definedness.** Unlike many other tools (e.g. Dafny), Whiley implicitly assumes that specification elements (e.g. pre-/postconditions and invariants) 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 specify "`i >= 0 && i < |a|`" to avoid the verifier reporting an out-of-bounds error. Such preconditions are implicit in Whiley, so must be extracted and made explicit in a translation to Boogie.

We now present the main contribution of this paper, namely a mechanism for translating Whiley programs into Boogie. These are implemented in our translator program, called Wy2B.[1] Whilst, in some cases, this process is straightforward, there are a number of subtle issues to be addressed, such as the representation of Whiley types in Boogie.

### 3.1 Types

Finding an appropriate representation of Whiley types is a challenge. We begin by considering the straightforward (i.e. naive) translation of Whiley types into Boogie, and highlight why this fails. Then, we present a new and more sophisticated approach, which we refer to as the *set-based translation*.

*Naive Translation of Types.* The simple and obvious translation of Whiley types into Boogie is a direct translation to the built-in types of Boogie. Here, an **`int`** in Whiley is translated into a Boogie **`int`**, whilst a Whiley array (e.g. **`int[]`**) translates to a Boogie map (e.g. **`[int]int`**, with appropriate constraints). Similarly, records in Whiley can be translated using Boogie's map type. However, this approach immediately encounters some serious impedance mismatch problems. For example, the type "**`int|null`**" cannot be represented in Boogie because there is no corresponding Boogie type. In addition, a Whiley type test such as "`x` **`is int`**" cannot be translated. To resolve these issues requires an altogether different approach.

[1] Source code and test programs are viewable at https://github.com/utting/WhileyCompiler/tree/wyboogie and are based on Whiley release 0.4.0.

*Set-Based Translation of Types.* Our second approach to modeling Whiley types and data values uses a *set-based* approach. That is, we model *all* Whiley values as members of a single set, called `WVal` (short for *Whiley Value*), and model the various Whiley types as being subsets of this universal set. We also define several helper types for Whiley record labels and higher-order function/method names:

```
type WVal;         // The set of ALL Whiley values.

type WField;       // field names for records and objects.
type WFuncName;    // names of functions
type WMethodName; // names of methods
```

For each distinct Whiley type `T`, we define a subset predicate `isT()` that is true for values in `T`, an extraction function `toT()` that maps a `WVal` value to a Boogie type, and an injection function `fromT()` which does the reverse mapping. We axiomatize these two functions to define a partial bijection between `T` and the subset of `WVal` that satisfies `isT()`.

For example, the functions for the Whiley **int** type of unbounded integers (recall "`int`" is also the Boogie name for integers) are as follows. We also add Boogie axioms to ensure that the `WVal` subsets that correspond to each basic Whiley type (int, bool, array, etc.) are mutually disjoint.

```
function isInt(WVal) returns (bool);
function toInt(WVal) returns (int);
function fromInt(int) returns (WVal);

axiom (forall i:int :: isInt(fromInt(i)));
axiom (forall i:int :: toInt(fromInt(i)) == i);
axiom (forall v:WVal :: isInt(v) ==> fromInt(toInt(v)) == v);
```

The set-based approach has several advantages. Firstly, it is easy to define a Whiley user-defined subtype `SubT` by defining a predicate $\texttt{isSubT(v)} = (\texttt{isT(v)} \wedge ...)$. Secondly, the Whiley union, intersection and negation types simply map to disjunction, conjunction and negation of these type predicates. Thirdly, Boogie sees all Whiley values as `WVal` objects, so can prove equality of two of those objects only if they are constructed using the same `fromT` injection function from values that are equal. This is a weak notion of equality, which can be strengthened by adding type-specific axioms where needed, as we shall now demonstrate for arrays.

Whiley arrays are fixed length, whereas Boogie maps are total. To represent Whiley arrays, we model them using a Boogie map `[int]WVal` from integers to `WVal` objects, plus the explicit length of the array, but we encode these two values as a single WVal value, using Boogie equality axioms, as follows. We provide an extraction function for each of these components (`toArray()` and `arraylen()`, respectively), and an injection function that takes both components and constructs the corresponding fixed length array in `WVal`.

```
function toArray(WVal) returns ([int]WVal);
function arraylen(WVal) returns (int);
function fromArray([int]WVal,int) returns (WVal);

axiom (forall s:[int]WVal, len:int :: 0 <= len
      ==> isArray(fromArray(s,len)));
axiom (forall s:[int]WVal, len:int :: 0 <= len
      ==> arraylen(fromArray(s,len)) == len);
axiom (forall v:WVal :: isArray(v)
      ==> fromArray(toArray(v), arraylen(v)) == v);
axiom (forall v:WVal :: isArray(v)
      ==> 0 <= arraylen(v));
```

We also provide a convenience function for updating one element of an array, by using Boogie's `a[i := v]` map update function, which returns `a` with index `i` updated to be `v` with the necessary conversion functions.

```
function arrayupdate(a:WVal, i:WVal, v:WVal) returns (WVal) {
    fromArray(toArray(a)[toInt(i) := v], arraylen(a)) }
```

Whiley records are also modeled using Boogie maps, with all unknown fields mapping to a special `undef__field` value. To create records dynamically, we start from the empty record (no fields) called `empty__record`, and update the required fields with their values. Whiley objects are similar to records, but have an extensible set of field names, so that "subtypes" can have more fields than "supertypes" (note that Whiley uses structural subtyping, so it is not necessary to declare subtype relationships explicitly).

```
// Record literals use empty__record[f1 := v1][f2 := v2] etc.
const unique empty__record : [WField]WVal;
const unique undef__field:WVal;
axiom (forall f:WField :: empty__record[f] == undef__field);
```

Overall, this "types-as-subsets" approach has avoided the impedance mismatch of the naive translation and made it easy to map the rich value and type system of Whiley into Boogie. One minor practical issue, however, was that our first version of the translator tended to produce deeply nested unreadable sequences of redundant extraction and injection functions; this was because all subexpressions were converted to `WVal` results. For example, `x := 2*x+1` was translated to:

```
    x := fromInt(toInt(fromInt(toInt(fromInt(2)) * toInt(x)))
                 + toInt(fromInt(1)))
```

We solved this problem by tracking the Boogie type of each subexpression and inserting these coercion functions lazily, only where needed, giving a more concise and readable translation:

```
1     x := fromInt(2 * toInt(x) + 3)
```

### 3.2 Control Flow

Translating most Whiley declarations and statements into Boogie is straightforward (see the similarities between Figs. 1 and 2). Here, we describe the interesting cases that illustrate impedance mismatches between Whiley and Boogie.

The first issue is that Whiley function bodies are defined by code (with restrictions to ensure no external side-effects are possible), whereas Boogie functions are either uninterpreted or have a single expression as their body. To overcome this, we translate each Whiley function `f(i)` → `(o)` : `body` (where `i` and `o` are vectors of variables, possibly empty) into *four* Boogie definitions:[2]

1. A pure function "`f__pre(i) returns (bool)`" with an expression body that is the Whiley precondition of `f`, including any type invariants on the input parameters `i`. This is useful for generating proof obligations for calls to `f`;
2. A pure function "`f(i) returns (o)`", which is called whenever a Whiley expression calls `f`, with an axiom "$\forall$`i, o : WVal :: f(i) == o` $\wedge$ `f__pre(i)` $\implies$ `post`" (where `post` is the Whiley postcondition of `f`).
3. A procedure specification "`f__impl(i) returns (o)`" with precondition `f__pre(i)` and a postcondition of `post`;
4. A procedure implementation of "`f__impl(i) returns (o)`", which contains the translated `body` code of the Whiley function `f`.

This approach causes Boogie to generate proof obligations to ensure that `body` satisfies the procedure specification. A call to `f(i)` within a Whiley expression is translated to a Boogie function call to `f(i)`, which has the desired precondition and postcondition properties, but none of the extra properties of `code`. This is acceptable, since Boogie supports only *modular* verification, which means that calls to a function or procedure must be verified using its specification, not its implementation.

We translate Whiley *methods* (procedures) in the same way, but omit step 2 (the pure function), because Whiley methods typically have side effects. However, a complication is that expressions in executable Whiley code can call methods as well as functions, whereas Boogie only allows methods (procedures) to be called via a "`call`" statement, and not from within expressions. We currently translate simple method calls (those that appear at the outermost level of the right-hand-side of an assignment) into `call` statements, and throw a translation error for method calls within expressions. Extending the translator to handle these will require doing a pre-pass of all expressions to pull those methods calls out into separate `call` statements, and even this will not handle some scenarios of side-effect method calls within short-circuit boolean operators.

[2] Note that Boogie always separates procedure specifications and implementations, which are our definitions (3) and (4).

Another aspect of the impedance mismatch is that, unlike Whiley, Boogie has no do-while statement. We initially translated "**do:** `code` **while** `c`" as "`code;` **while**`(c) {code}`" in Boogie. But this did not handle **break** and **continue** statements within `code`. Next we tried translating do-while into a single while loop with a boolean flag to force a first iteration. However, this gave the wrong semantics for loop invariants—in a Whiley do-while loop the invariant need not be true before the first iteration of `code` (this makes some proofs easier). We now translate do-while statements into `code;` **while**`(c){code}`, but generate explicit labels for blocks in order to implement **break**/**continue** statements using gotos.

The Whiley `switch` statement posed similar challenges, since Boogie has no switch statement. Rather than translating this to a sequence of if-else statements, we translate it to a non-deterministic goto to all the available cases. Thus, we can use labels and goto statements to implement the Whiley semantics of `break`/`continue` statements (in Whiley, `break` exits the whole switch statement, while `continue` falls through to the following case).[3]

Other minor impedance mismatches that we encountered were:

- Function overloading is supported in Whiley, but not in Boogie. So we mangle the names of any overloaded functions.
- Function/procedure inputs are treated as mutable local variables in Whiley, but not in Boogie. To overcome this, for any input that is mutated, the translator generates a local variable that contains a copy of the input value.
- Boogie does not provide lambda expressions for functions (only for maps), so the translator has to convert Whiley lambda expressions to named functions (not yet implemented). This is adequate for simple lambda expressions, but not for lambda expressions that capture local variables.
- Boogie requires all local variables to be declared at the start of a procedure body, which makes it harder to generate temporary variables during translation of expressions.
- Typing versus proof. The Whiley compiler uses typing algorithms to distinguish bytes from integers (which are unbounded), and gives the bitwise operators different semantics on those two types. This proved hard to do in Boogie when we treated bytes as a subrange of integers and overloaded the bitwise operators, so we had to generate separate operators for byte and integer values, and axiomatise them differently.

## 4 Generating Verification Conditions

After translating a Whiley program into Boogie, we use the Boogie tool to generate and check proof obligations to ensure the usual correctness condition for each procedure: `pre` $\Longrightarrow$ `wp(body, post)` [3]. In addition to the inherent proof

[3] Our translator does not yet implement the translation of continue statements within switch statements, because it is rarely needed, but there are no technical obstacles to doing this.

obligations, we generate several kinds of Whiley-specific proof obligations by inserting **assert** statements in the generated Boogie code. Boogie then attempts to prove each of these assert statements.

The additional proof obligations we generate include any explicit assert statements included in the Whiley program, plus assertions to check three Whiley runtime correctness conditions that Boogie does not check automatically:

1. function calls satisfy their preconditions (Boogie functions are total, but Whiley functions have **requires** conditions, so we generate an assertion before any expression that calls a function, to check that its precondition is satisfied);
2. array indexes are in bounds; and
3. the divisor is non-zero in division and modulo expressions.

Since these assertions may be generated from subexpressions that are deeply nested inside complex predicates, we need to carefully define the assumptions that are available for each proof obligation. This is achieved via a recursive descent into each predicate and expression, collecting the context assumptions using the following window-inference rules [27]. A premise of the form $assert(A \implies P)$ means that this assertion is inserted into the generated Boogie program as a proof obligation.

$$\frac{A \vdash check(P) \quad A, P \vdash check(Q)}{A \vdash check(P \wedge Q)} \qquad \frac{A \vdash check(P) \quad A, \neg P \vdash check(Q)}{A \vdash check(P \vee Q)} \qquad \frac{A, x : int; a \leq x; x \leq b \vdash check(P)}{A \vdash check(\forall x : a \ldots b :: P)}$$

$$\frac{assert(A \implies d \neq 0) \quad A \vdash check(e) \quad A \vdash check(d)}{A \vdash check(e/d)} \qquad \frac{assert(A \implies f\_\_pre(a)) \quad A \vdash check(a)}{A \vdash check(f(a))} \qquad \frac{assert(A \implies 0 \leq i \wedge i < arraylen(a)) \quad A \vdash check(a) \quad A \vdash check(i)}{A \vdash check(a[i])}$$

The conjunction and disjunction rules are not symmetric—they assume the left-hand predicate while checking the right-hand predicate, but not vice versa. This is done to reflect the execution semantics of Whiley expressions, which is to execute subexpressions left-to-right.

## 5 Experimental Results

In this section, we discuss the effectiveness of the Wy2B translator as an alternative verification path, in terms of what Whiley language features can be translated, the limitations and challenges of the approach, and what percentage of valid Whiley programs can be verified using the Wy2B+Boogie+Z3 toolchain (using Boogie v2.3.0.61016 and Z3 v4.4.0, with no custom triggers on axioms).

Figure 4 shows statistics comparing how the native Whiley verifier (the y-axis) and the new Wy2B+Boogie+Z3 verifier (the x-axis) handle the nearly 500 valid test case programs in the Whiley distribution, which are intended to methodically test all Whiley language features. Each of these short test programs (ranging from 3 to 100 lines of Whiley code, with an average length of 18

lines) typically contains multiple function and method definitions, and each definition can generate several proof obligations. We classify the results according to whether the verifier: (i) fully verifies *all* the proof obligations for that program (**Fully**); (ii) fails to verify one or more of the proof obligations (**Partial**); (iii) generates proof obligations that cause Boogie errors (**Errors**)—this may be due to accidental use of reserved words, or Whiley constructs that are too complex for Boogie; (iv) the test program uses Whiley features that are not yet able to be translated to Boogie by our Wy2B translator (**NotImpl**).

All of these test programs are intended to be verifiable, but some have not yet been verified by either prover because they use language features that are not yet fully supported in the verifiers. For example, neither prover fully models the semantics of bitwise operators, lambda expressions, or heap allocation and the address-of operator yet. Instead, these features are modelled as uninterpreted functions, which means that general properties of those operators are sometimes provable, but assertions that depend upon the specific semantics of those operators are not yet provable.

**Wy2B+Boogie+Z3**

| Whiley Verifier | NotImpl | Errors | Partial | Fully | Total | Total% |
|---|---|---|---|---|---|---|
| Partial | 5 | 7 | 22 | 54 | 88 | 17.8% |
| Fully | 38 | 13 | 9 | 345 | 405 | 82.2% |
| Total | 43 | 20 | 31 | 399 | 493 | |
| Total% | 8.7% | 4.1% | 6.3% | 80.9% | | 100.0% |

**Fig. 4.** Comparison of Whiley Verifier results (y-axis) with Boogie+Z3 results (x-axis).

Overall, the Whiley verifier can fully verify 82.2% of the programs, while the Wy2B+Boogie tools can fully verify only 80.9%. Part of the reason for this difference is that there are several language constructs such as lambda expressions, multiple return values and calling methods (with side-effects) from within non-specification expressions, that are not yet implemented in the Wy2B translator (8.7% of programs). So there are opportunities for improvement in the Wy2B+Boogie path.

Considering a detailed breakdown of results, the largest result category is that there are 345 programs (70%) that can be fully verified with both verifiers. There are also 54 programs (11%) that can be fully verified with Wy2B+Boogie but only partially with the Whiley verifier, and 9 programs (1.8%) that can be fully verified with the Whiley verifier, but Boogie fails to fully verify. These nine cases are as follows:

**Complex_Valid_5.whiley:** Boogie fails to instantiate axioms to prove a subtype condition.

**ConstrainedList_Valid_14.whiley:** Boogie fails to prove a typing condition containing $(\exists i : int \,|\, 0 \leq i < |xs| \bullet xs[i] > 0)$, after the assignment "`xs[0]=1`".

**DoWhile_Valid_6.whiley:** Boogie cannot reestablish an invariant in a do-while loop.
**DoWhile_Valid_8.whiley:** similar problem, but with a **break** inside the loop.
**FunctionRef_Valid_9.whiley:** cannot establish a result type for an indirect function call of a function inside a record, inside the heap.
**MessageSend_Valid_2.whiley:** cannot establish result type of a heap reference.
**MessageSend_Valid_5.whiley:** ibid.
**RecursiveType_Valid_19.whiley:** complex recursive subtypes. 10 s timeout.
**RecursiveType_Valid_20.whiley:** ibid.

The Wy2B+Boogie toolchain takes 502 s to translate and verify the 493 test programs (9040 lines of Whiley code) on a MacBook Pro (Intel Core i5-4258U 2.4 GHz). This is approximately one second per program, which is acceptable performance for real-world usage. We run Boogie with a timeout of 10 s, but only three programs failed to verify due to timeouts.

## 6 Related Work

ESC/Modula-3 was one of the earliest tools to use an intermediate verification language [10]. This was based on Dijkstra's language of guarded-commands [11] 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 [17].

The ESC/Java tool followed ESC/Modula-3 in using guarded commands, but employed a multi-stage process allowing "high-level" guarded command programs to be desugared into a lower-level form [14]. Spec# followed this lineage of tools and the language of guarded commands used previously was reused in Boogie [4]. 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"* [3]. In essence, Boogie was a version of the guarded command language from ESC/Java which also supported a textual syntax, type checking, and a static analysis for inferring loop invariants. Other important innovations include the ability to specify *triggers* to help guide quantifier instantiation, and the use of a trace semantics to formalise the meaning of Boogie.

In addition to Boogie, the other main intermediate verification language in use is WhyML [6,13]. This is part of the Why3 verification platform which specifically exploits external theorem provers. 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).

Research on intermediate verification languages has often encountered impedance mismatch. Ameri and Furia present a translation from Boogie to WhyML which, although largely successful, did expose some important mismatches between them [1]. 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.

Segal and Chalin [28] attempted a systematic comparison of two intermediate verification languages: Boogie and Pilar. 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.

Müller *et al.* [23] argue that existing systems (e.g. Boogie, Why3) do not support separation logics and related permission-based logics. They identify that such systems have a "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 designed specifically for this. Finally, Arlt *et al.* [2] presented a translation from SOOT's intermediate bytecode language (Jimple) to Boogie, with an aim of identifying unreachable code. They found many aspects of the translation straightforward. For example, Java's `instanceof` operator was modelled using an uninterpreted function. 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.

## 7 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 (e.g. the Whiley **do-while** loop). Fortunately, this impedance mismatch can be circumvented by translating to lower-level Boogie statements, such as labels and gotos. 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 shows that the encoding one chooses when translating source language types and values into Boogie has a major impact on the effectiveness of the resulting system. 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 Wy2B+Boogie+Z3 is competitive with the native Whiley verifier in terms of the percentage of programs that it can verify automatically. Finally, interesting future work would be to explore translating Boogie's *counter-example models* back into Whiley-like notation to improve error reporting. Other priorities are to add support for a few remaining Whiley language features to our Wy2B translator, and to continue to improve the Whiley verifier.

## References


1. Ameri, M., Furia, C.A.: Why just Boogie? In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 79–95. Springer, Cham (2016). doi:10.1007/978-3-319-33693-0_6
2. Arlt, S., Rümmer, P., Schäf, M.: Joogie: from Java through Jimple to Boogie. In: Proceedings of SOAP (2013)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). doi:10.1007/11804192_17
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. CACM **54**(6), 81–91 (2011)
5. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of the OOPSLA, pp. 113–132. ACM Press (2012)
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Workshop on Intermediate Verification Languages (2011)
7. 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: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03359-9_2
8. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005). doi:10.1007/978-3-540-30569-9_6
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. JACM **52**(3), 365–473 (2005)
10. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. SRC Research Report 159, Compaq Systems Research Center (1998)
11. Dijkstra, E.W.: Guarded commands, nondeterminancy and formal derivation of programs. CACM **18**, 453–457 (1975)

12. Dutertre, B., Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). doi:10.1007/11817963_11
13. Filliâtre, J., Paskevich, A.: Why3 – where programs meet provers. In: Proceedings of ESOP, pp. 125–128 (2013)
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of PLDI, pp. 234–245 (2002)
15. Hoare, C.: The verifying compiler: a grand challenge for computing research. JACM **50**(1), 63–69 (2003)
16. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20398-5_4
17. 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)
18. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17511-4_20
19. Leino, K.R.M.: Program proving using intermediate verification languages (IVLs) like Boogie and Why3. In: Proceedings of HILT, pp. 25–26 (2012)
20. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
21. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
22. Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73595-3_13
23. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49122-5_2
24. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. JACM **27**, 356–364 (1980)
25. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 335–354. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_21
26. Pearce, D.J., Groves, L.: Designing a verifying compiler: lessons learned from developing Whiley. Sci. Comput. Program. **113**, 191–220 (2015)
27. Robison, P.J., Staples, J.: Formalizing a hierarchical structure of practical mathematical reasoning. J. Logic Comput. **3**(1), 47–61 (1993). http://dx.doi.org/10.1093/logcom/3.1.47
28. Segal, L., Chalin, P.: A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 130–145. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27705-4_11
29. The SMT-LIB standard: Version 2.0
30. The Whiley programming language. http://whiley.org