Why Just Boogie?

Translating Between Intermediate Verification Languages

Michael Ameri[1] and Carlo A. Furia[2⋆]

[1] Chair of Software Engineering, Department of Computer Science,
ETH Zurich, Switzerland mameri@student.ethz.ch
[2] Department of Computer Science and Engineering,
Chalmers University of Technology, Sweden furia@chalmers.se

Abstract. The verification systems Boogie and Why3 use their respective intermediate languages to generate verification conditions from high-level programs. Since the two systems support different back-end provers (such as Z3 and Alt-Ergo) and are used to encode different high-level languages (such as C# and Java), being able to translate between their intermediate languages would provide a way to reuse one system's features to verify programs meant for the other. This paper describes a translation of Boogie into WhyML (Why3's intermediate language) that preserves semantics, verifiability, and program structure to a large degree. We implemented the translation as a tool and applied it to 194 Boogie-verified programs of various sources and sizes; Why3 verified 83% of the translated programs with the same outcome as Boogie. These results indicate that the translation is often effective and practically applicable.

1 Introduction

Intermediate verification languages (IVLs) are intermediate representations used in verification technology. Just like compiler design has benefited from decoupling front-end and back-end, IVLs help write verifiers that are more modular: the front-end specializes in encoding the rich semantics of a high-level language (say, an object-oriented language such as C#) as a program in the IVL; the back-end generates verification conditions (VCs) from IVL programs in a form that caters to the peculiarities of a specific theorem prover (such as an SMT solver).

Boogie [3] and WhyML [6] are prime examples of popular IVLs with different, often complementary, features and supporting systems (respectively called Boogie and Why3). In this paper we describe a translation of Boogie programs into WhyML programs and its implementation as the tool `b2w`. As we illustrate with examples in Sec. 3, using `b2w` increases the versatility brought by IVLs: without having to design and implement a direct encoding into WhyML, users can take advantage of some of the best features of Why3 when working with high-level languages that translate to Boogie.

⋆ Work done mainly while affiliated with ETH Zurich.

Boogie vs. WhyML. While the roles of Boogie and WhyML as IVLs are similar, the two languages have different characteristics that reflect a focus on complementary challenges in automated verification. Boogie is the more popular language in terms of front-ends that use it as IVL, which makes a translation *from* Boogie more practically useful than one into it; it has a finely tuned integration with the Z3 prover that results from the two tools having been developed by the same group (Microsoft Research's RiSE); it combines a simple imperative language with an expressive typed logic, which is especially handy for encoding object-oriented or, more generally, heap-based imperative languages. In contrast, WhyML has a more flexible support for multiple back-end provers it translates to, including a variety of SMT solvers as well as interactive provers such as Coq; it can split VCs into independent goals and dispatch them to different provers; if offers limited imperative constructs within a functional language that belongs to the ML family, which brings the side benefit of being able to *execute* WhyML programs—a feature quite useful to debug and validate verification attempts.

Goals and evaluation. The overall goal of this paper is devising a translation $\mathcal{T}$ from Boogie to WhyML programs. The translation, described in Sec. 4, should preserve correctness and verifiability as much as possible. Preserving correctness means that, given a Boogie program p, if its translation $\mathcal{T}(p)$ is a correct WhyML program then p is correct (soundness); the converse should also hold as much as possible: if $\mathcal{T}(p)$ is incorrect then p is too (precision). Preserving verifiability means that, given a Boogie program p that verifies in Boogie, its translation $\mathcal{T}(p)$ is a WhyML program that verifies in Why3.

The differences, outlined above, between Boogie and WhyML and their supporting systems make achieving correctness and verifiability challenging. While we devised $\mathcal{T}$ to cover the entire Boogie language, its current implementation `b2w` does not fully support a limited number of features (branching, the most complex polymorphic features, and bitvectors) that make it hard to achieve verifiability in practice. In fact, while replacing branching (goto) with looping is always possible [9], a general translation scheme does not produce verifiable loops since one should also infer invariants (which are often cumbersome due to the transformation). Polymorphic maps are supported to the extent that their type parameters can be instantiated with concrete types; this is necessary since WhyML's parametric polymorphism cannot directly express all usages in Boogie, but it may also introduce a combinatorial explosion in the translation; hence, `b2w` fails on the most complex instances that would be unmanageable in Why3. Boogie's bitvector support is much more flexible than what provided by Why3's libraries; hence `b2w` may render the semantics of bitvector operations incorrectly.

These current implementation limitations notwithstanding (see Sec. 4 for details), we experimentally demonstrate that `b2w` is applicable and useful in practice. As Sec. 5 discusses, we applied `b2w` to 194 Boogie programs of different size and sources; most of the programs have not been written by us and exercise Boogie in a variety of different ways. For 83% (161) of these programs, `b2w` produces a WhyML translation that Why3 can verify as well as Boogie can verify the original, thus showing the feasibility of automating translation between IVLs.

Tool availability. For lack of space this paper omits some details that are available as a technical report [1]. The tool `b2w` is available as open source at: `https://bitbucket.org/michael_ameri/b2w/`

2 Related Work

Translations and abstraction levels. Translation is a ubiquitous technique in computer science; however, the most common translation schemes bridge *different abstraction levels*, typically encoding a program written in a high-level language (such as Java) into a lower-level representation which is suitable for execution (such as byte or machine code). Reverse-engineering goes the opposite direction—from lower to higher level—for example to extract modular and structural information from C programs and encode it using object-oriented constructs [19]. This paper describes a translation between intermediate languages—Boogie and Why3—which belong to *similar abstraction levels*. In the context of model transformations [15], so-called bidirectional transformations [18] also target lossless transformations between notations at the same level of abstraction.

Intermediate verification languages. The Spec# project [4] introduced Boogie to add flexibility to the translation between an object-oriented language and the verification conditions. Since its introduction for Spec#, Boogie has been adopted as intermediate verification language for numerous other front-ends such as Dafny [13], AutoProof [21], Viper [10], and Joogie [2]; its popularity demonstrates the advantages of using intermediate verification languages.

While Boogie retains some support for different back-end SMT solvers, Z3 remains its primary target. By contrast, supporting multiple, different back-ends is one of the main design goals behind the Why3 system [6]. Why3 also fully supports interactive provers, which provide a powerful means of discharging the most complex verification conditions that defy complete automation.

In all, while the Boogie and WhyML languages belong to a similar abstraction level, they are part of systems with complementary features, which motivates this paper's idea of translating one language into the other.

Other intermediate languages for verification are Pilar [17], used in the Sireum framework for SPARK; Silver [10], an intermediate language with native support for permissions in the style of separation logic; and the flavor of dynamic logic for object-oriented languages [16] used in the KeY system. Another approach to generalizing and reusing different translations uses notions from model transformations to provide validated mappings for different high-level languages [5]. Future work may consider supporting some of these intermediate languages and approaches.

3 Motivating Examples

Verification technology has made great strides in the last decade or two, but a few dark corners remain where automated reasoning shows its practical limitations. Fig. 1 provides three examples of simple Boogie programs that trigger incorrect or otherwise unsatisfactory behavior. We argue that translating these programs to WhyML makes it possible to verify them using a different, somewhat complementary verification tool; overall, confidence in the results of verification is improved.

Procedure `not_verify` in Fig. 1 has a contradictory postcondition (notice $\texttt{N} < \texttt{N}$, `N` is a nonnegative constant, and the loop immediately terminates). Nonetheless, recent

versions of Boogie and Z3 successfully verify it.[3] More generally, unless the complete toolchain has been formally verified (a monumental effort that has only been performed in few case studies [14,11,12]), there is the need to *validate* the successful runs of a verifier. Translating Boogie to Why3 provides an effective validation, since Why3 has been developed independent of Boogie and uses a variety of backends that Boogie does not support. Procedure `not_verify` translated to Why3 (Fig. 2) does not verify as it should.

Procedures `lemma_yes` and `lemma_no` in Fig. 1 demonstrate Boogie's support for mathematical real numbers, which is limited in the way the power operator $**$ is handled. Boogie vacuously verifies both properties $2^3 > 0$ and $2^3 < 0$, even though Z3 outputs some unfiltered errors that suggest the verification is spurious. Why3 provides a more thorough support for real arithmetic; in fact, it verifies the translated procedure `lemma_yes` but correctly fails to verify `lemma_no`.

The loop in procedure `trivial_inv` in Fig. 1 includes an invariant asserting that `i` takes only even values. Even if this is clearly true, Boogie fails to check it; pinning down the precise cause of this shortcoming requires knowledge of Boogie's (and Z3's) internals, although it likely is a manifestation of the "triggers" heuristics that handle (generally undecidable) quantified expressions. However, if we insist on verifying the program in its original form, Why3 can dispatch verification conditions to *interactive* provers, where the user provides the crucial proof steps.[4] Cases such as the loop invariant of `trivial_inv` where a proof is "obvious" to a human user but it clashes against the default strategies to handle quantifiers are prime candidate to exploit interactive provers.

```
const N: int;
axiom 0 ≤ N;

procedure not_verify()
  ensures (∀ k, l: int ●
   0 ≤ k ≤ l < N ⟹ N < N);
{
  var x: int;
  x := −N;
  while (x ≠ x) { }
}
```

```
procedure lemma_yes()
  ensures 2.0**3.0 > 0.0;
{  }

procedure lemma_no()
  ensures 2.0**3.0 < 0.0;
{  }
```

```
procedure trivial_inv()
{
  var i: int;
  i := 0;
  while (i < 10)
   invariant 0 ≤ i ≤ 10;
   invariant
    (∃ j: int ● i = 2*j);
  { i := i + 2; }
}
```

Fig. 1. Three simple Boogie programs for which automated reasoning is limited.

4 Boogie-to-Why3 Translation

Intermediate languages for verification combine programming constructs and a logic language. When used to encode programs written in a high-level language, the programming constructs encode program behavior, and the logic constructs encode specifications, constrain the semantics to conform to the high-level language's (typically through axioms), and support other kinds of annotations (such as triggers).

[3] `https://github.com/boogie-org/boogie/issues/25`

[4] Why3 can also check the invariant automatically by relying on the CVC4 SMT solver.

```
constant N: int
axiom A0: 0 ≤ N;

val not_verify (): ()
  ensures { ∀ k, l: int .
   0 ≤ k ≤ l < N → N < N }

let not_verify_impl(): ()
  ensures { ∀ k, l: int .
   0 ≤ k ≤ l < N → N < N }
 =(
   let x = ref (any int) in
    x.contents ← -N;
    while
     (x.contents ≠ x.contents)
    do done;
  end )
```

```
val lemma_yes (): ()
  ensures
    { (pow 2.0 3.0) >. 0.0 }

val lemma_no (): ()
  ensures
    { (pow 2.0 3.0) <. 0.0 }

let lemma_yes_impl (): ()
  ensures
    { (pow 2.0 3.0) >. 0.0 }
=( )

let lemma_no_impl (): ()
  ensures
    { (pow 2.0 3.0) <. 0.0 }
=( )
```

```
val trivial_inv (): ()

let trivial_inv_impl (): ()
=(
  let i = ref (any int) in
   i.contents ← 0;
   while (i.contents < 10) do
    invariant
      { 0 ≤ i.contents ≤ 10 }
    invariant
      { ∃ j: int .
          i.contents = 2*j }
    i.contents ← i.contents + 2;
   done;
 )
```

Fig. 2. The translation to WhyML of the three Boogie programs in Fig. 1.

Both Boogie and WhyML provide, as logic language, a typed first-order logic with arithmetic. Boogie's programming constructs are a simple imperative language with both structured (while loops, procedures) and unstructured (jumps, global variables) statements. WhyML's programming constructs combine an ML-like functional language with a few structured imperative features such as mutable variables and loops.

Correspondingly, we define a translation $\mathcal{T}$: *Boogie* $\rightarrow$ *WhyML* of Boogie to WhyML as the composition $\mathcal{E} \circ \mathcal{D}$ of two translations: $\mathcal{D}$: *Boogie* $\rightarrow$ *Boogie* is a desugaring[5] which rewrites away the Boogie constructs, such as *call-forall*, that have no similar construct in WhyML by expressing them using other features of Boogie. Then, $\mathcal{E}$: *Boogie* $\rightarrow$ *WhyML* encodes Boogie programs simplified by $\mathcal{D}$ as WhyML programs. For simplicity, the presentation does not sharply separate the two translations $\mathcal{D}$ and $\mathcal{E}$ but defines either or both of them as needed to describe the translation of arbitrary Boogie constructs.

A single feature of the Boogie language significantly compounds the complexity of the translation: *polymorphic maps*. For clarity, the presentation of the translation initially ignores polymorphic maps. Then, Sec. 4.8 discusses how the general translation scheme can be extended to support them.

As running examples, Fig. 2 shows how $\mathcal{T}$ translates the examples of Fig. 1. For lack of space, we focus on describing the most significant aspects of the translation that are also implemented; see [1] for the missing details.

4.1 Types

Primitive types are `int` (mathematical integers), `real` (mathematical reals), and `bool` (Booleans). $\mathcal{T}$ translates primitive types into their Why3 analogues as shown in Tab. 3.

Type constructors. A Boogie type declaration using the *type constructor* syntax introduces a new parametric type `T`. $\mathcal{T}$ translates it to an algebraic type with constructor `T`: $\mathcal{T}($ `type T` $\mathtt{a}_1 \ldots \mathtt{a}_m$ $) =$ `type T` $\mathtt{'a}_1 \ldots \mathtt{'a}_m$ for $m \geq 0$, where ticks `'` identify type parameters in WhyML.

[5] This is unrelated to Boogie's built-in desugaring mechanism (option `/printDesugared`).

T	$\mathcal{T}(T)$	Why3 libraries
`int`	`int`	`int.Int`, `int.EuclideanDivision`
`real`	`real`	`real.RealInfix`, `real.FromInt`, `real.Truncate`, `real.PowerReal`
`bool`	`bool`	`bool.Bool`

Table 3. Translation of primitive types, and Why3 libraries supplying the necessary operations.

Map types. A Boogie *map type* `M` declared as: `type M = [T1, ... Tn] U` defines the type of a mapping from $\mathtt{T}_1 \times \cdots \times \mathtt{T}_n$ to `U`, for $n \geq 1$. Why3 supports maps through its library `map.Map`; hence, $\mathcal{T}(\mathtt{M}) = \mathtt{map}\ (\mathcal{T}(\mathtt{T}_1), \ldots, \mathcal{T}(\mathtt{T}_n))\ \mathcal{T}(\mathtt{U})$, where an n-tuple encapsulates the n-type domain of `M`.

4.2 Constants

The translation of constant declarations is generally straightforward, following the scheme:

$$\mathcal{T}(\mathtt{const\ c:\ T}) = \mathtt{constant\ c:}\ \mathcal{T}(\mathtt{T})$$

$\mathcal{T}$ expresses *unique* constants and *order* constraints by axiomatization.

4.3 Variables

Why3 supports mutable variables through the reference type `ref` from theory `Ref`. Boogie global variable declarations become global value declarations of type `ref`; Boogie local variable declarations become **`let`** bindings with local scope. Thus, if `v` is a global variable and `l_v` is a local variable in Boogie:

$$\begin{array}{ll} \text{global variable} & \mathcal{T}(\mathtt{var\ v:\ T}) = \mathtt{val\ v:\ ref}\ \mathcal{T}(\mathtt{T}) \\ \text{local variable} & \mathcal{T}(\mathtt{var\ l\_v:\ T}) = \mathtt{let\ l\_v = ref\ (any}\ \mathcal{T}(\mathtt{T}))\ \mathtt{in} \end{array}$$

The expression **`any`** `T` provides a nondeterministic value of type `T`.

4.4 Functions

Boogie function *declarations* become WhyML function declarations:

$$\mathcal{T}\big(\mathtt{function\ f(x_1:\ T_1, \ldots,\ x_n:\ T_n)\ returns\ (U)}\big) \\ = \mathtt{function\ f}\ (\mathtt{x}_1 : \mathcal{T}(\mathtt{T}_1)) \cdots (\mathtt{x}_n : \mathcal{T}(\mathtt{T}_n)) : \mathcal{T}(\mathtt{U}) \quad (1)$$

WhyML function *definitions* require, unlike Boogie's, a variant to ensure that recursion is well-formed. Therefore, Boogie function definitions are not translated into WhyML function definitions but are axiomatized.[6]

4.5 Expressions

Variables. Since a Boogie variable `v` of type `T` turns into a value `v` of type `ref` $\mathcal{T}(\mathtt{T})$, occurrences of `v` in an expression translate to `v.contents`, which represents the value attached to reference `v`.

[6] To take advantage of Why3's well-formedness checks, we plan to offer translations of Boogie functions to WhyML functions as a user option in future work.

Map expressions. $\mathcal{T}$ translates map selection and update using functions `get` and `set` from theory `Map`. If `m` is a map of type `M` defined in Sec. 4.1, then:

	E	$\mathcal{T}(E)$
selection	`m[`e_1`,...,` e_n`]`	`get` $\mathcal{T}$`(m) (`$\mathcal{T}(e_1)$`,...,`$\mathcal{T}(e_n)$`)`
update	`m[`e_1`,...,` e_n`:= f]`	`set` $\mathcal{T}$`(m) (`$\mathcal{T}(e_1)$`,...,`$\mathcal{T}(e_n)$`)` $\mathcal{T}$`(f)`

Lambda expressions. The translation desugars lambda expression into constant maps: $\mathcal{D}$`(λ x`$_1$`: T`$_1$`,...,x`$_n$`: T`$_n$`• e)` $=$ `lmb`, where `const lmb :` $[$`T`$_1$`,...,T`$_n$$]\tau$`(e)` is axiomatized by `axiom (∀ x`$_1$`: T`$_1$`,...,x`$_n$`: T`$_n$`• lmb[x`$_1$`,...,x`$_n$`] = e)`, and τ`(e)` is `e`'s type.

4.6 Procedures

Boogie procedures have a declaration (signature and specification) and zero or more implementations. The latter follow the general syntax of Fig. 4 (left). For simplicity of presentation, `p` has one input argument, one output argument, and one local variable, but generalizing the description to an arbitrary number of variables is straightforward.

```
procedure p(t: T where Wt)
  returns (u: U where Wu);
  requires      R;
  free requires fR;
  modifies      M;
  ensures       E;
  free ensures  fE;

implementation p(t: T)
  returns (u: U)
  {
    var l: L where Wl;
    B
  }
```

```
val p (t : T(T)): T(U)
  requires { T(R) }
  writes   { M }
  returns  { | u → T(E) }
  returns  { | u → T(fE) }
  returns  { | u → T(Wu) }

let p_impl0 (t: T(T)): T(U)
  requires { T(R) } requires { T(fR) }
  returns  { | u → T(E) }
=(
  T(var u: U; var l: L;)
  assume { T(Wg) }  -- where of globals
  assume { T(Wt) }  -- where of inputs
  assume { T(Wl) }  -- where of locals
  assume { T(Wu) }  -- where of outputs
  try ( T(B) )
  with | Return → assume { true } end
  T(u)
 )
```

Fig. 4. Translation of a Boogie procedure (left) into WhyML (right).

The specification of procedure `p` consists of preconditions `requires`, frame specification `modifies`, and postconditions `ensures`. Specification elements marked `free` are assumed without being checked.

$\mathcal{T}$ translates a generic procedure `p` as shown in Fig. 4 (right). The declaration of `p` determines `val p`, which defines the semantics of `p` for clients: the `free` precondition `fR` does not feature there because clients don't have to satisfy it, whereas both `free` and non-`free` postconditions are encoded as `returns` conditions. The implementation of `p` determines `let p_impl0`, which triggers the verification of the implementation against its specification: both `free` and non-`free` preconditions are encoded, whereas the `free` postcondition `fE` does not feature there because implementations don't have to satisfy it. The body introduces `let` bindings for the local variable `l` and for a new local variable `u` which represents the returned value; these declarations are translated as discussed

in Sec. 4.3. Then, a series of **assume** encode the semantics of Boogie's **where** clauses, which constrain the nondeterministic values variables can take (`Wg` comes from any global variables, which are visible everywhere); `p`'s body `B` is translated and wrapped inside an exception-handling block **try**, which does not do anything other than allowing abrupt termination of the body's execution upon throwing a `Return` exception (see Sec. 4.7 for details). Regardless of whether the body terminates normally or exceptionally, the last computed value of `u` is returned in the last line, and checked against the postcondition in **returns**. In all, the modular semantics of Boogie's procedure `p` is preserved.

4.7 Statements

Assignments. Assignments involve variables (global or local), which become mutable references in WhyML: $\mathcal{T}(\texttt{v} := \texttt{e}) = \texttt{v.contents} \leftarrow \mathcal{T}(\texttt{e})$. Boogie parallel assignments become simple assignments using **let** bindings of limited scope:

$$\mathcal{T}(\texttt{v}_1,\ldots,\texttt{v}_m := \texttt{e}_1,\ldots,\texttt{e}_m) = \begin{cases} \textbf{let } \texttt{e'}_1 = \mathcal{T}(\texttt{e}_1),\ldots,\texttt{e'}_m = \mathcal{T}(\texttt{e}_m) \textbf{ in} \\ \quad \mathcal{T}(\texttt{v}_1 := \texttt{e'}_1); \cdots; \mathcal{T}(\texttt{v}_m := \texttt{e'}_m) \end{cases} \tag{2}$$

Havoc. An abstract function **val** `havoc (): 'a` provides a fresh, nondeterministic value of any type `'a`. It translates Boogie's **havoc** statements following the scheme:

$$\mathcal{T}(\textbf{havoc } \texttt{u, v}) = \mathcal{T}(\texttt{u}) \leftarrow \texttt{havoc()}; \mathcal{T}(\texttt{v}) \leftarrow \texttt{havoc()}; \textbf{assume } \{\, \mathcal{T}(\texttt{Wu}) \,\}; \textbf{assume } \{\, \mathcal{T}(\texttt{Wv}) \,\}$$

where `Wu` and `Wv` are the **where** clauses of `u`'s and `v`'s declarations; the generalization to an arbitrary number of variables is obvious. It is important that the **assume** statements follow all the calls to `havoc`: since `Wv` may involve `u`'s value, **havoc** `u, v` is not in general equivalent to **havoc** `u;` **havoc** `v`; the translation reflects this behavior.

Return. The behavior of Boogie's **return** statement, which determines the abrupt termination of a procedure's execution, is translated to WhyML using exception handling. An exception handling block wraps each procedure's body, as illustrated in Fig. 4, and catches an **exception** `Return`; thus, $\mathcal{T}(\textbf{return}) =$ **raise** `Return`.

Loops. Fig. 5 shows the translation of a Boogie loop into a WhyML loop. An invariant marked as **free** can be assumed but need not be checked; correspondingly, the translation adds assumptions that ensure it holds at loop entrance and after every iteration. The exception handling block surrounding the loop in WhyML emulates the semantics of the control-flow breaking statement **break**: $\mathcal{T}(\textbf{break}) =$ **raise** `Break`.

4.8 Polymorphic Maps

We now consider *polymorphic map* types, declared in Boogie as:

$$\textbf{type } \texttt{pM} = \langle \boldsymbol{\alpha} \rangle \ \texttt{[T}_1,\ldots,\texttt{T}_n\texttt{] U} \tag{3}$$

where $\boldsymbol{\alpha}$ is a vector $\alpha_1,\ldots,\alpha_m$ of $m > 0$ type parameters, and some of the types $\texttt{T}_1,\ldots,\texttt{T}_n,\texttt{U}$ in `pM`'s definition depend on $\boldsymbol{\alpha}$. In the next paragraph, we explain why

```
while (b)
 invariant I;
 free invariant fI;
{ B }
```

```
assume { 𝒯(fI) }
try while 𝒯(b) do
  invariant { 𝒯(I) }
  invariant { 𝒯(fI) }
  𝒯(B)
  assume { 𝒯(fI) }
done;
with | Break → assume { 𝒯(fI) } end
```

Fig. 5. Translation of a Boogie loop (left) into WhyML (right).

polymorphic maps cannot be translated to WhyML directly. Instead, we replace them with several monomorphic maps based on a global analysis of the types that are actually used in the Boogie program being translated. The result of this rewrite is a Boogie program without polymorphic maps, which we can translate to Why3 following the rules we previously described. The shortcoming of this approach is that it gives up *modularity*: verification holds only for the concrete types that are used (closed-word assumption); this seems to be necessary to express Boogie's extremely liberal polymorphism without resorting to intricate "semantic" translations, which would likely fail verifiability.

Boogie vs. WhyML polymorphism. While WhyML also supports generic polymorphism, its usage is more restrictive than Boogie's. The first difference is that *mutable* maps cannot be polymorphic in WhyML. The second difference is that, in some contexts, a variable of polymorphic map type in Boogie effectively corresponds to *multiple* maps. Consider, for example, a `type Mix =` $\langle\alpha\rangle[\alpha]\alpha$ of maps from generic α to α; Boogie accepts formulas such as `axiom (∀ m: Mix • m[0] = 1 ∧ m[true])` where `m` acts as a map over `int` in the first conjunct and as a map over `bool` in the second. WhyML, in contrast, always makes the type parameters explicit; hence, a logic variable of type `map 'a 'a` denotes a single map of a generic type that can only feature in expressions which do not assume anything about the concrete type that will instantiate `'a`.

Besides type declarations and quantifications, polymorphic maps can appear within polymorphic functions and procedures, declared as:

$$\texttt{function pF}\langle\boldsymbol{\alpha}\rangle\texttt{(x}_1\texttt{: T}_1\texttt{,...,x}_n\texttt{: T}_n\texttt{) returns (U)} \tag{4}$$

$$\texttt{procedure pP}\langle\boldsymbol{\alpha}\rangle\texttt{(x}_1\texttt{: T}_1\texttt{,...,x}_n\texttt{: T}_n\texttt{) returns (u: U)} \tag{5}$$

Type analysis. We have seen that a Boogie polymorphic map may correspond to multiple monomorphic maps in certain contexts. The translation reifies this idea based on global type analysis: for every item (constant, program or logic variable, or formal argument) `pm` of polymorphic map type `pM` as in (3), it determines the set $types(\texttt{pm})$ of all actual types `pm` takes in expressions or assignments, as outlined in Tab. 6. This in turn determines the set $types(\texttt{pM})$ as the union of all sets $types(\texttt{p})$ for `p` of type `pM`.

The types in $types(\texttt{pM})$ include in general both concrete and parametric types. For example, the program of Fig. 7 (left) determines $types(\texttt{m}) = \{\texttt{[int]int}, [\beta]\beta\}$,

			$types$(pm) includes $[t_1, \ldots, t_n]u$ such that:
expressions	read	pm	pm :: $[t_1, \ldots, t_n]u$
	select	pm[$e_1, \ldots, e_n$]	$e_1 :: t_1, \ldots, e_n :: t_n$, pm[$e_1, \ldots, e_n$] :: u
	update	pm[$e_1, \ldots, e_n$:= f]	$e_1 :: t_1, \ldots, e_n :: t_n$, f :: u
	function reference	f(it)	it :: $[t_1, \ldots, t_n]u$, where `function f(pm: pM)`
statements	copy	pm := it	it :: $[t_1, \ldots, t_n]u$
	assignment	pm[$e_1, \ldots, e_n$] := f	$e_1 :: t_1, \ldots, e_n :: t_n$, f :: u
	havoc	`havoc` pm	–
	procedure call in	`call` p(it)	it :: $[t_1, \ldots, t_n]u$, where `procedure p(pm: pM)`
	procedure call out	`call` it := p()	it :: $[t_1, \ldots, t_n]u$, where `procedure p() returns(pm: pM)`

Table 6. Each occurrence of an item pm of polymorphic map type pM determines the set $types$(pm) of actual types. (x :: t denotes that x has type t.)

$types$(n) = {`[bool]bool`}, and $types$(M) = $types$(m) ∪ $types$(n), where β is procedure p's type parameter (since p is not called anywhere, that's the only known actual type of x). Let $conc$(pM) denote the set of all *concrete* types in $types$(pM).

```
type M = ⟨α⟩ [α]α;
var m: M;
axiom (∀ n: M • n[true]);

procedure p⟨β⟩(x: β)
  requires (∀ i: int • m[i] = i);
  modifies m;
{ m[x] := x; }
```

```
type (M_int, M_bool, M_a) = ([int]int, [bool]bool, [a]a);
var (m_int, m_bool, m_a): (M_int, M_bool, M_a);
axiom (∀ (n_int, n_bool, n_a): (M_int, M_bool, M_a) •
         n_bool[true]);

procedure (p_int, p_bool, p_a)(x: (int, bool, a))
  requires (∀ i: int • m_int[i] = i);
  modifies (m_int, m_bool, m_a);
{ (m_int, m_bool, m_a)[x] := x }
```

Fig. 7. An example of how polymorphic maps (left) translate to monomorphic (right). Procedure p translates to 3 procedures p_int, p_bool, and p_a, each with argument of type int, bool, or a.

Desugaring polymorphic maps. To describe how the translation replaces polymorphic maps by monomorphic maps, we introduce a pseudo-code notation that allows *tuples* (in round brackets) of program elements where normally only a single element is allowed. The semantics of this notation corresponds quite intuitively to multiple statements or declarations. For example, a variable declaration `var (x, y): (int, bool)` is a shorthand for declaring variables `x: int` and `y: bool`; a formula `(x, y) = (3, true)` is a shorthand for `x = 3 ∧ y`; and a procedure declaration using the tuple notation `procedure (p_int, p_bool)(x: (int, bool))` is a shorthand for declaring two procedures `p_int(x: int)` and `p_bool(x: bool)`.

We also use the following notation: given an n-vector $\boldsymbol{a} = a_1, \ldots, a_n$ and a type expression T parametric with respect to $\boldsymbol{\alpha}$, $T_{\boldsymbol{a}}$ denotes T with a_k substituted for α_k, for $k = 1, \ldots, n$. If $\mathbb{T}$ is a set of types obtained from the same type expression T, such as $types$(pM) with respect to pM's definition, and id is an identifier, let (𝕋) denote $\mathbb{T}$ as a tuple, and (id_𝕋) denote the tuple of identifiers $\mathtt{id\_}\boldsymbol{t}$ such that $T_{\boldsymbol{t}}$ is the corresponding type in $\mathbb{T}$. In the example of Fig. 7, if T = [α]α then $T_{\texttt{int}}$ = `[int]int`, ($types$(m)) = (`[int]int`,[β]β), and (j_$types$(m)) = `(j_int, j_`β`)`. Throughout, we also assume that an uninterpreted type $\mathtt{a}_k$ is available for $k = 1, \ldots, n$, that $M_{\mathtt{a}}$ denotes the type expression `[T1, ..., Tn] U` in (3) with each α_k replaced by $\mathtt{a}_k$, and that $conc^+$(pM) = $conc$(pM) ∪ {$M_{\mathtt{a}}$}.

Declarations. Type declaration (3) desugars to several type declarations:

$$\texttt{type (pM\_}conc^+\texttt{(pM)))} = \texttt{(}conc^+\texttt{(pM))} \tag{6}$$

The declaration of an *item* `pm: pM`, where `pm` can be a constant, or a program or logic variable, desugars to a declaration $\texttt{(pm\_}conc^+\texttt{(pM)))}\texttt{: (}conc^+\texttt{(pM))}$ of multiple items of the same kind. The declaration of a *procedure* or *function* `g` with an (input or output) argument `x: pM` desugars to a declaration of multiple procedures or functions $\texttt{(g\_}conc^+\texttt{(pM))(x: (}conc^+\texttt{(pM))}$—multiple declarations each with one variant of `x`; if `g` has multiple arguments of this kind, the desugaring is applied recursively to each variant. Fig. 7 (right) shows how the polymorphic map type `M` and each of the items `m` and `n` of type `M` become 3 monomorphic types and 3 items of these monomorphic types.

For every polymorphic function or procedure `g` with type parameters $\boldsymbol{\beta}$, also consider any one of their arguments declared as `x:` X. If X is a type expression that depends on $\boldsymbol{\beta}$, and there exists a map type $\texttt{[}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{]}V_0$ in $types(\texttt{pM})$ such that $X = V_k$ for some $k = 0, \ldots, n$, then `g` becomes $\texttt{(g\_}\mathbb{V}_k\texttt{)(x: (}\mathbb{V}_k\texttt{))}$—corresponding to multiple `g`'s each with one argument, where $\mathbb{V}_k = \left\{\overline{V}_k \mid \texttt{[}\overline{V}_1\texttt{,}\ldots\texttt{,}\overline{V}_n\texttt{]}\overline{V}_0 \in conc^+\texttt{(pM)}\right\}$ is the set of all concrete types that instantiate the kth type component. This transformation enables assigning arguments to polymorphic maps inside polymorphic functions or procedures that have become monomorphic. Fig. 7 (right) shows how argument `x:` β becomes an argument of concrete type `int`, `bool`, or `a`, since $\texttt{[}\beta\texttt{]}\beta \in types(\texttt{M})$. (As procedure `p` does not use β elsewhere, we drop it from the signature.)

Expressions. Every occurrence—in expressions, as l-values of assignments, and as targets of `havoc` statements—of an item `w` of polymorphic type `W` whose declaration has been modified to remove polymorphic map types is replaced by one or more of the newly introduced monomorphic types as follows. If `w`'s actual type within its context is a *concrete* type `C`, then we replace `w` with $\texttt{w\_}c$ such that $\texttt{W}_c = \texttt{C}$; otherwise, `w`'s actual type is a *parametric* type, and we replace `w` with the tuple $\texttt{(w\_}X\texttt{)}$, including all variants of `w` that have been introduced. In Fig. 7 (right), `n[true]` rewrites to just `n_bool[true]` since the concrete type is `bool`; the assignment in `p`'s body, whose actual type is parametric with respect to β, becomes an assignment involving each of the three variants of `m` corresponding to the three variants of `p` that have been introduced.

5 Implementation and Experiments

5.1 Implementation

We implemented the translation $\mathcal{T}$ described in Sec. 4 as a command-line tool `b2w` implemented in Java 8. `b2w` works as a staged filter: 1) it parses and typechecks the input Boogie program, and creates a Boogie AST (abstract syntax tree); 2) it desugars the Boogie AST according to $\mathcal{D}$; 3) it transforms the Boogie AST into a WhyML AST according to $\mathcal{E}$; 4) it outputs the WhyML AST in the form of code.

Stage 1) relies on Schäf's parsing and typechecking library Boogieamp[7], which we modified to support access using the visitor pattern, AST in-place modifications, and

[7] `https://github.com/martinschaef/boogieamp`

the latest syntax of Boogie (e.g., for integer vs. real division). Stages 2) and 3) are implemented by multiple AST visitors, each taking care of a particular aspect of the translation, in the style of [20]; the overhead of traversing the AST multiple times is negligible and improves modularity: handling a new construct (for example, in future versions of Boogie) or changing the translation of one feature only requires adding or modifying one feature-specific visitor class.

5.2 Experiments

The goal of the experiments is ascertaining that `b2w` can translate realistic Boogie programs producing WhyML programs that can be verified taking advantage of Why3's multiple back-end support. The experiments are limited to fully-automated verification, and hence do not evaluate other possible practical benefits of translating programs to WhyML such as support for interactive provers and executability for testing purposes.

Programs. The experiments target a total of 194 Boogie programs from three groups according to their origin: group NAT (native) includes 29 programs that encode algorithmic verification problems directly in Boogie (as opposed to translating from a higher-level language); group OBJ (object-oriented) includes 6 programs that are based on a heap-based memory model; group TES (tests) includes 159 programs from Boogie's test suite. Tab. 8 summarizes the sizes of the programs in each group.

		LOC BOOGIE				LOC WHYML			
GROUP	#	m	μ	M	Σ	m	μ	M	Σ
NAT	29	20	73	253	2110	62	128	318	3716
OBJ	6	44	146	385	878	90	208	446	1245
TES	159	3	21	155	3272	36	64	290	10180
Total:	194	3	34	385	6260	36	106	446	15141

Table 8. A summary of the Boogie programs used in the experiments, and their translation to WhyML using `b2w`. For each program GROUP, the table reports how many programs it includes (#), the minimum m, mean μ, maximum M, and total Σ length in non-comment non-blank lines of code (LOC) of those BOOGIE programs and of their WHYML translations.

The programs in NAT, which we developed in previous work [8,7], include several standard algorithms such as sorting and array rotation. The programs in OBJ include 2 simple examples in Java and 1 in Eiffel, encoded in Boogie by Joogie [2] and AutoProof [21] (we manually simplified AutoProof's translation to avoid features `b2w` doesn't support), and 3 algorithmic examples adapted from NAT to use a global heap in the style of object-oriented programs. Among the 515 programs that make up Boogie's test suite[8] we retained in TES those that mainly exercise features supported by `b2w`.

Setup. Each experiment targets one Boogie program `b`: it runs Boogie with command `boogie b` and a timeout of 180 seconds; it runs `b2w` to translate `b` to `w` in WhyML; for each SMT solver `p` among Alt-Ergo, CVC3, CVC4, and Z3, it runs Why3 with command `why3 prove -P p w`, also with a timeout of 180 seconds. For each run we collected the wall-clock running time, the total number of verification goals, and how many of such goals the tool verified successfully.

[8] `https://github.com/boogie-org/boogie/tree/master/Test`

All the experiments ran on a Ubuntu 14.04 LTS GNU/Linux box with 8-core Intel i7-4790 CPU at 3.6 GHz and 16 GB of RAM, with the following tools: Alt-Ergo 0.99.1, CVC3 2.4.1, CVC4 1.4, Z3 4.3.2, Mono 4.2.2, Boogie 2.3.0.61016, and Why3 0.86.2. To account for noise, we repeated each verification three times and report the mean value of the 95th percentile of the running times.

GROUP	#	B = W	B > W	B < W	0=0	50=50	100=100	SPURIOUS
NAT	29	19	10	0	1	0	18	0
OBJ	6	5	0	1	1	2	2	0
TES	159	137	21	1	71	21	45	0
Total:	194	161	31	2	73	23	65	0

Table 9. A summary of how Boogie performs in comparison with Why3. For each program GROUP, the table reports how many programs it includes (#), for how many of the programs Boogie verifies as many goals (B = W), more goals (B > W), or fewer goals (B < W) than Why3 with any of the SMT solvers; for how many of the programs both Boogie and Why3 verify none (0=0), some but not all (50=50), or all (100=100) of the goals; the last column (SPURIOUS) indicates that `b2w`'s translation never introduces spurious goals that are proved by Why3 (that is, if Boogie's input has zero goals, so does WhyML's translation).

Results. Tab. 9 shows a summary of the results where we compare Why3's best performance, with any one of the four SMT solvers, against Boogie's. The most significant result is that the WhyML translation produced by `b2w` behaves like the Boogie original in 83% (161, B=W) of the experiments. This means that Boogie may fail to verify all goals (column 0=0), verify some goals and fail on others (column 50=50), or verify all goals (column 100=100); in each case, Why3 consistently verifies the same goals on `b2w`'s translation. Indeed, many programs in TES are tests that are supposed to fail verification; hence, the correct behavior of the translation is to fail as well. We also checked the failures of programs in NAT and OBJ to ascertain that `b2w`'s translation preserves correctness. Tab. 9 does not show this, but we also found another 2 programs in NAT where Why3 proves the same goals as Boogie only by combining the results of multiple SMT solvers.

Boogie verifies more goals than Why3 in 16% (31, B > W) of the experiments, where it is more effective because of better features (default triggers, invariant inference, SMT encoding) or simply because of some language features that are not fully supported by `b2w` (examples are Z3-specific annotations, which `b2w` simply drops, and `goto`, which `b2w` encodes as `assert false` to ensure soundness). In 1% (2, B < W) of the experiments, Why3 even verifies more goals than Boogie. One program in OBJ is a genuine example where Why3's Z3 encoding is more effective than Boogie's; the one program in TES should instead be considered spurious, as it deploys some trigger specifications that are Boogie-specific (negated triggers) or interact in a different way with the default triggers. As this was the only program in our experiments that introduced clearly spurious behavior, the experiments provide convincing evidence that `b2w`'s translation preserves correctness and verifiability to a large degree.

Tab. 10 provides data about the experiments' running times, and differentiates the performance of each SMT solver with Why3. Z3 is the most effective SMT solver in terms of programs it could completely verify (columns ∀), followed by Alt-Ergo. While CVC3 is generally the least effective, it has the advantage of returning very quickly (only 0.2 seconds of average running time), even more quickly than Z3 in Boogie.

		Z3 BOOGIE						ALT-ERGO WHY3						CVC3 WHY3						CVC4 WHY3						Z3 WHY3					
		OUTCOME			TIME			OUTCOME			TIME			OUTCOME			TIME			OUTCOME			TIME			OUTCOME			TIME		
GROUP	#	μ	$\forall$	$\not\exists$	μ	Σ	∞	μ	$\forall$	$\not\exists$	μ	Σ	∞	μ	$\forall$	$\not\exists$	μ	Σ	∞	μ	$\forall$	$\not\exists$	μ	Σ	∞	μ	$\forall$	$\not\exists$	μ	Σ	∞
NAT	29	93	25	1	0.4	12	0	61	14	6	20.6	598	0	28	1	12	0.2	5	0	33	2	11	30.1	873	0	73	16	5	12.6	367	0
OBJ	6	52	2	2	3.9	23	0	46	1	2	30.1	181	0	46	1	2	0.2	1	0	52	2	2	28.4	170	0	68	3	1	23.7	142	0
TES	159	45	55	71	0.3	53	0	37	45	85	25.8	4096	1	33	39	91	0.1	18	0	37	45	86	27.4	4360	1	37	44	86	25.9	4121	1
Total:	194	60	82	74	0.7	88	0	53	60	93	22.6	4875	1	30	41	105	0.2	24	0	35	49	99	29.7	5403	1	69	63	92	14.5	4630	1

Table 10. For each program GROUP the table reports how many programs it includes (#) and, for both Boogie and Why3 for each choice of SMT solver among ALT-ERGO, CVC3, and Z3: the mean percentage of goals verified in each program (OUTCOME μ), how many programs were completely verified (OUTCOME $\forall$), and how many were not verified at all (OUTCOME $\not\exists$), the mean μ and total Σ verification TIME in seconds, and how many programs timed out.

CVC4 falls somewhere in the middle, in terms both of effectiveness and of running time. Boogie's responsiveness remains excellent if balanced against its effectiveness; a better time-effectiveness of Why3 with Alt-Ergo and Z3 could be achieved by setting tight per-goal timeouts (in most cases, verification attempts that last longer than a few seconds do not eventually succeed).

6 Discussion

The current implementation of the translation $\mathcal{T}$ has some limitations that somewhat restrict its applicability. As we already mentioned in the paper, some features of the Boogie language are not supported (bitvectors, gotos), or only partially supported (polymorphic mappings); and frame specifications are assumed. All of these are, however, limitations of the current prototype implementation only, and we see no fundamental hurdles to extending `b2w` along the lines of the definition of $\mathcal{T}$ in Sec. 4.

Since `b2w` also takes great care to confine the effect of translating Boogie programs that include unsupported features, and to fail when it cannot produce a correct translation, it still largely preserves *correctness* (soundness, in particular). On the other hand, our experiments also demonstrate that the translation $\mathcal{T}$, as implemented by `b2w`, largely meets the other goal of preserving *verifiability*: even if the experimental subjects all are idiomatic Boogie programs written independent of the translation effort, 83% of the translated programs behave in Why3 as they do in Boogie.

In future work, we will address the features of Boogie that are still not satisfactorily supported by `b2w`. We will also devise strategies to take advantage of Why3's multi-prover support. Other possible directions include formalizing the translation to prove that it preserves correctness; and devising a reverse translation from WhyML to Boogie.

References

1. M. Ameri and C. A. Furia. Why just Boogie? Translating between intermediate verification languages. `http://arxiv.org/abs/1601.00516`, January 2016.
2. S. Arlt and M. Schäf. Joogie: Infeasible code detection for Java. In *Proceedings of CAV*, volume 7358 of *LNCS*, pages 767–773. Springer, 2012.
3. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

4. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
5. Z. Cheng, R. Monahan, and J. F. Power. A sound execution semantics for ATL via translation validation – research paper. In *Proceedings of ICMT*, volume 9152 of *LNCS*, pages 133–148. Springer, 2015.
6. J. Filliâtre and A. Paskevich. Why3 – where programs meet provers. In *Proceedings of ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
7. C. A. Furia. Rotation of sequences: Algorithms and proofs. `http://arxiv.org/abs/1406.5453`, June 2014.
8. C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys*, 46(3):Article 34, 2014.
9. D. Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.
10. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *Proceedings of ECOOP*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.
11. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.
12. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Proceedings of POPL*, pages 179–192. ACM, 2014.
13. K. R. M. Leino. Developing verified programs with Dafny. In *Proceedings of ICSE*, pages 1488–1490. ACM, 2013.
14. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
15. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
16. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *Proceedings of FoVeOOS*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.
17. L. Segal and P. Chalin. A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In *Proceedings of VSTTE*, volume 7152 of *LNCS*, pages 130–145. Springer, 2012.
18. P. Stevens. A landscape of bidirectional model transformations. In *Proceedings of GTTSE*, volume 5235 of *LNCS*, pages 408–424. Springer, 2008.
19. M. Trudel, C. A. Furia, M. Nordio, and B. Meyer. Really automatic scalable object-oriented reengineering. In *Proceedings of ECOOP*, volume 7920 of *LNCS*, pages 477–501. Springer, 2013.
20. M. Trudel, C. A. Furia, M. Nordio, B. Meyer, and M. Oriol. C to O-O translation: Beyond the easy stuff. In *Proceedings of WCRE*, pages 19–28. IEEE Computer Society, October 2012.
21. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *Proceedings of TACAS*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015.