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 caf@inf.ethz.ch

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 84% 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 [2] and WhyML [8] 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 or even being familiar with its peculiarities, users can take advantage 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, verifiability, and readability as much as possible. Preserving correctness means that, given a correct Boogie program p, its translation $\mathcal{T}(p)$ is a correct WhyML program with the same semantics as p; if p is incorrect, $\mathcal{T}(p)$ should also be incorrect. 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. Preserving readability means that the translation should not introduce unnecessary changes in the structure of programs.

The differences, outlined above, between Boogie and WhyML and their supporting systems make achieving correctness, verifiability, and readability 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 [11], 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 84% (162) 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. 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 that 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 [25]. This paper describes a translation between intermediate languages—Boogie and Why3—which belong to *similar abstraction levels*. In the context of model transformations [19], so-called bidirectional transformations [24] also target lossless transformations between notations at the same level of abstraction.

Intermediate verification languages. The Spec# project [3] introduced Boogie to add flexibility to the translation between an object-oriented language (a dialect of C#) and the verification conditionsin the logic fragments supported by SMT solvers. An intermediate verification language embodies the idea of intermediate representation—a technique widespread in compiler construction—in the context of verification. Since its introduction for Spec#, Boogie has been adopted as intermediate verification language for numerous other front-ends such as Dafny [16], AutoProof [27], Viper [12], and Joogie [1]; its popularity demonstrates the advantages of using intermediate verification languages.

While Boogie retains some support for different back-end SMT solvers, Z3 [7] remains its fully supported primary target. By contrast, supporting multiple, different back-ends is one of the main design goals behind the Why3 system [8], which does not merely generate verification conditions in different formats but offers techniques to split them into independently verifiable units and to dispatch each unit to a different prover. Why3 also fully supports interactive provers[3] which provides a powerful means of discharging the most complex verification conditions that defy complete automation.

Another element that differentiates Boogie and Why3 is the support for executing programs; this is quite useful for debugging verification attempts and for applying testing-like techniques to the realm of verification. Boogaloo [20] supports symbolic execution of Boogie programs; Symbooglix is a more recent project with the same goal.[4] Thanks to it being a member of the ML family, Why3 directly supports symbolic execution as well as compilation of WhyML programs to OCaml.

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. Since Boogie is overall more popular, in

[3] In comparison, Boogie's support for HOL is restricted and not up-to-date [4].

[4] `http://srg.doc.ic.ac.uk/projects/symbooglix/`

terms of tools that use it as a back-end, the translation from Boogie to WhyML is more practically useful than the one in the opposite direction.

Other intermediate languages for verification are Pilar [23], used in the Sireum framework for SPARK; Silver [12], an intermediate language with native support for permissions in the style of separation logic; and a flavor of dynamic logic for object-oriented languages [22] 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, and 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 `N` $<$ `N`, `N` is a nonnegative constant, and the loop immediately terminates). Nonetheless, recent versions of Boogie and Z3 successfully verify it.[5] More generally, unless the complete toolchain has been formally verified (a monumental effort that has only been performed in few case studies [18,13,14]), 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 backend tools 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 (the power operator $**$ is not properly supported); indeed, only the inequality encoded by `lemma_yes` is correct. Why3 provides a more thorough support for real arithmetic, both by translating to backends such as Alt-Ergo and by providing a more effective encoding in Z3; in fact, it verifies the translated procedure `lemma_yes` but correctly fails to verity `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. Based on this knowledge, there are specification patterns that try to work around such idiosyncrasies; in the example, one could introduce a "witness" ghost variable `k` such that `i` $=$ `2*k` is an invariant. However, if we insist on verifying the program in its original form, Why3 can dispatch verification

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

conditions to *interactive* provers, where the user provides the crucial proof steps. 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. Thus, translating Boogie to Why3 provides another means of exploiting the latter's versatile support for interactive provers and multiple backends.

```
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).

Both Boogie and WhyML provide a typed first-order logic with arithmetic as logic language. Boogie's programming constructs are a simple imperative language with both structured (while loops, procedures, and so on) 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 $\mathcal{T}\colon \mathit{Boogie} \to \mathit{WhyML}$ of Boogie to WhyML as the composition $\mathcal{E} \circ \mathcal{D}$ of two translations: $\mathcal{D}\colon \mathit{Boogie} \to \mathit{Boogie}$ is a desugaring 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, $\mathcal{E}\colon \mathit{Boogie} \to \mathit{WhyML}$ encodes Boogie programs simplified by $\mathcal{D}$ as WhyML programs, while introducing constraints that ensure that the semantics in WhyML mirrors the one in Boogie. 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*, which correspond to mappings between domains of generic type. Why3 does support polymorphic maps through a library, but its type

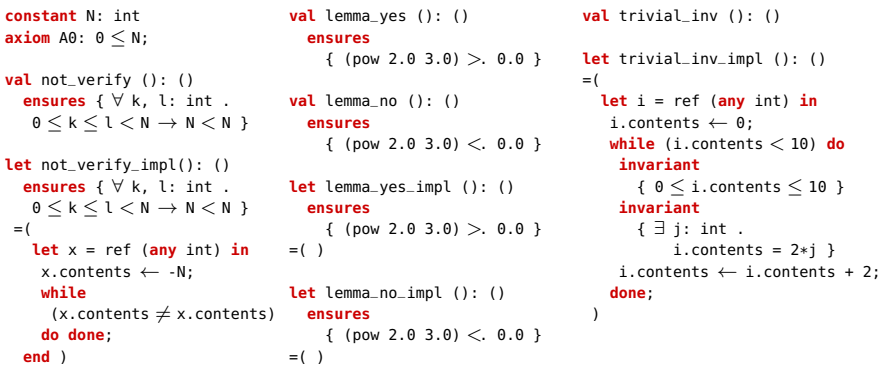

Fig. 2. The translation to WhyML of the three Boogie programs in Fig. 1. (Boilerplate such as general declarations, imports, and frame condition checking are omitted for clarity.)

system is more restrictive and does not allow the same degree of freedom as Boogie's in using variables of polymorphic map types. For clarity, the presentation of the translation initially ignores polymorphic maps. Then, Sec. 4.10 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.

4.1 Types

Boogie types include primitive types, instantiated type constructors, and map types.

Primitive types are `int` (mathematical integers), `real` (mathematical reals), `bool` (Booleans), and `bv`n (n-bit vectors). $\mathcal{T}$ translates primitive types into their Why3 analogues as shown in Tab. 1. Since Why3 offers primitive types and operations on them through libraries, $\mathcal{T}$ also generates import statements for the libraries that provide the same operations that are available in Boogie, such as integer to/from real conversion.

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`
`bv`n	`bv`	`bv.BitVector with constant size =` n

Table 1. Translation of primitive types, and Why3 libraries supplying the necessary operations.

Type constructors. A Boogie type declaration using the *type constructor* syntax [6] introduces a new parametric type T with parameters $\mathtt{a}_1, \ldots, \mathtt{a}_m$. $\mathcal{T}$ translates it to an algebraic

[6] $\mathcal{T}$ ignores the optional type modifier `finite`, since it does not seem fully supported in Boogie.

type with constructor T: $\mathcal{T}$(`type T a`$_1$`...a`$_m$) = `type T 'a`$_1$`... 'a`$_m$ for $m \geq 0$, where ticks ' identify type parameters in WhyML.

Map types. A Boogie *map type* `M` declared as: `type M = [T`$_1$`, ...T`$_n$`] 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`;[7] hence, $\mathcal{T}$(`M`) = `type M = map (`$\mathcal{T}$(`T`$_1$`), ..., `$\mathcal{T}$(`T`$_n$`)) `$\mathcal{T}$(`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:

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

Unique constants. All constants of a type `T` declared with the modifier `unique` have values that are pairwise different. Thus, for m constants `const unique c`$_1$`,..., c`$_m$`:T`, $\mathcal{T}$ encodes the uniqueness properties using $\binom{m}{2}$ axioms`axiom unique_c_i_j: c`$_i \neq$ `c`$_j$, for $1 \leq i \neq j \leq m$.

Orders. Boogie provides the operator <: to express partial order over every type; $\mathcal{T}$ introduces a polymorphic operator <: and axiomatizes its reflexive, antisymmetric, and transitive properties:

```
predicate (<:) (x: 'a) (y: 'a)
axiom ReflexivePO:     ∀ x: 'a     . x <: x
axiom AntisymmetricPO: ∀ x y: 'a   . x <: y ∧ y <: x → x = y
axiom TransitivePO:    ∀ x y z: 'a . x <: y ∧ y <: z → x <: z
```

Boogie supplies special syntax to describe a partial-order relations with a certain structure, which corresponds to a DAG where any two nodes x and y are connected by an edge $x \to y$ iff $x <: y$ and y is a direct successor of x in the order. Let `a`, `b`, `c`, `d`, `e`, `f` be unique[8] constants of the same type `T`. The Boogie syntax to specify ordering between them is in Fig. 3. $\mathcal{D}$ reconstructs the DAG of the order specification, and then formalizes it in axiomatic form. For example, the specifications in Fig. 3 determine the DAG in Fig. 4, which is axiomatized as in Fig. 5.

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:

global variable $\quad \mathcal{T}(\texttt{var v: T}) = \texttt{val v: ref } \mathcal{T}(\texttt{T})$
local variable $\quad \mathcal{T}(\texttt{var l\_v: T}) = \texttt{let l\_v = ref (any } \mathcal{T}(\texttt{T})\texttt{) in}$

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

[7] Why3's maps, like Boogie's, do not satisfy extensionality (`http://lists.gforge.inria.fr/pipermail/why3-club/2013-February/000572.html`).

[8] Uniqueness is not required but makes the order specification easier to present.

BOOGIE SPECIFICATION	SEMANTICS
`const c: T extends a, b;`	`a` and `b` are the only direct successors of `c`
`const a: T extends;`	`a` has no (direct) successors
`const d: T extends c complete;`	`c` has no direct predecessors other than `d` and any others that are explicitly specified
`const e, f: T extends unique d;`	`d` is the only direct successor of both `e` and `f`, and the sub-graphs that originate in `e` and `f` are disjoint

Fig. 3. Ordering specifications in Boogie (older versions of Boogie use <: instead of `extends`).

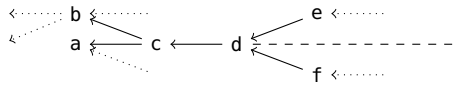

Fig. 4. DAG corresponding to the ordering specification of Fig. 3. Solid edges denote the successor relation; dotted edges denote allowed (but not specified) relations; the dashed line expresses disjointness of the two sub-graphs.

4.4 Functions

Boogie function *declarations* become WhyML function declarations:

$$\mathcal{T}\big(\texttt{function f(x}_1\texttt{: T}_1\texttt{, ..., x}_n\texttt{: T}_n\texttt{) returns (U)}\big) \\ = \texttt{function f (x}_1\texttt{:}\mathcal{T}\texttt{(T}_1\texttt{))} \cdots \texttt{(x}_n\texttt{:}\mathcal{T}\texttt{(T}_n\texttt{)):}\mathcal{T}\texttt{(U)} \quad (1)$$

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: if function `f` in (1) has body `B`, $\mathcal{D}$ replaces the body with the `axiom` $(\forall\ z_1\texttt{: T}_1\texttt{, ..., z}_n\texttt{: T}_n\bullet\ \texttt{f(z}_1\texttt{, ..., z}_n\texttt{) = B)}$.

4.5 Expressions

The translation of Boogie expressions to WhyML expressions is mostly straightforward, given the translation of types described above. We describe the few cases that deserve some detail.

Nondeterministic choice. The special value $*$ represent a nondeterministic Boolean choice (used in loop exit flags and conditionals); we define $\mathcal{T}(*) =$ `any bool`, which provides a nondeterministic Boolean value.

```
axiom (c <: a ∧ c <: b ∧ ∀ x: T • c <: x ⟹ c = x ∨ a <: x ∨ b <: x)
axiom (∀ x: T • ¬(a <: x))
axiom (d <: c ∧ ∀ x: T • x <: c ⟹ c = x ∨ x <: d)
axiom (e <: d ∧ ∀ x: T • e <: x ⟹ e = x ∨ d <: x)
axiom (f <: d ∧ ∀ x: T • f <: x ⟹ f = x ∨ d <: x)
axiom (∀ x: T • x <: e ⟹ ¬( x <: f))
axiom (∀ x: T • x <: f ⟹ ¬( x <: e))
```

Fig. 5. Axiomatization of the ordering specification in Fig. 3.

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

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:[9]

	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. Boogie recently introduced lambda expressions as syntactic sugar for maps. While WhyML has lambda abstractions, they are not allowed as first-order values in programs [6]. Instead, 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]τ(`e`) is axiomatized by `axiom (∀` x_1`:` T_1`,...,`x_n`:` T_n`•` `lmb[`x_1`,...,`x_n`] = e)`.

Old expression. Within a procedure's postcondition or body, the expression `old(e)` refers to the value of `e` in the prestate. WhyML offers a more general construct to refer to an expression's value at any labeled point within a procedure's body. Hence, every WhyML procedure implementation translating a Boogie procedure implementation includes a label `"begin"`, so that $\mathcal{T}$(`old(e)`) is just `old` $\mathcal{T}$(`e`) within postconditions, and is `at` $\mathcal{T}$(`e`) `' "begin"` within bodies.

Bitvectors. Why3's theory `BitVectors` does not provide all operations that are supported by Boogie. In particular, it does not support *extraction expressions* `b[n:m]` (drop the `m` least significant bits and return the next `n` − `m` least significant bits) and *concatenation expressions* `b ++ c` (the bit vector obtained by concatenating `b` and `c`). $\mathcal{T}$ introduces functions `extract (b: bv) (n: int) (m: int): bv` and `cat (b: bv) (c: bv): bv` and uses them to translate applications of these bit vector operators, but leaves them uninterpreted in Why3. $\mathcal{T}$'s implementation currently supports only the bitvectors operations available in Why3's theory `BitVectors`.

4.6 Procedures

Boogie procedures have a declaration (signature and specification) and zero or more implementations. The latter follow the general syntax of Fig. 6 (left), where a procedure `p` with input argument `t` and output argument `u` has one implementation with local variable `l` and body `B`. 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 trivial.

The specification of procedure `p` consists of preconditions `requires`, frame specification `modifies`, and postconditions `ensures`. A precondition is an assertion that callers of `p` must satisfy upon calling, and that every implementation of `p` can assume;

[9] Despite its name, `set` returns a new map rather than changing its argument's value.

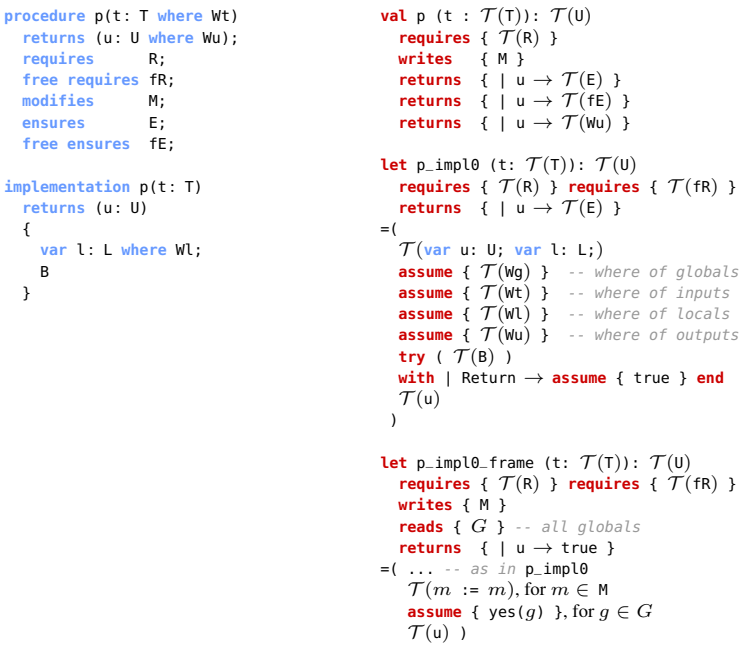

Fig. 6. Translation of a Boogie procedure (left) into WhyML (right).

`free` preconditions need not be satisfied by callers. A postcondition is an assertion that every implementation of `p` must satisfy upon terminating, and that every caller of `p` can assume; `free` postconditions need not be satisfied by implementations. Every implementation of `p` may only modify the global variables listed in `p`'s frame specification.

$\mathcal{T}$ translates a generic procedure `p` as shown in Fig. 6 (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**. Another implementation **let** p_impl0_frame checks the frame condition (**modifies** clause).[10] It relies on the same full precondition as p_impl0 but has postcondition true since E has already been checked; it includes a **writes** clause and a **reads** clause. Why3 checks that a global variable is in the **writes** clause if and only if it is written by the implementation; since Boogie's **modifies** clause only expresses variables that *may* be written, p_impl0_frame includes an assignment of every variable in M to itself so that the requirement that every variable in M is written is vacuously satisfied. When a **writes** clause is present, Why3 also requires a **reads** clause and checks that every variable in it is written, read, or both. The translation builds a **reads** clause with all global variables G, and vacuously reads all of them using function yes 'a: bool, which identically returns true for any input; this makes the reads clause satisfied by any implementation.

4.7 Statements

Axioms and assertions. Boogie's **assert** e, **assume** e, and **axiom** e statements translate to **assert** {$\mathcal{T}$(e)}, **assume** {$\mathcal{T}$(e)}, and **axiom** A: $\mathcal{T}$(e) in WhyML.

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

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

Havoc. An abstract function **val** havoc (): 'a provides a fresh, nondeterministic[11] value of any type 'a. It translates Boogie's **havoc** statements following the scheme:[12]

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

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. 6, and catches an **exception** Return; thus, $\mathcal{T}$(**return**) = **raise** Return.

[10] The tool b2w does not currently implement frame condition checks.
[11] http://lists.gforge.inria.fr/pipermail/why3-club/2013-April/000615.html
[12] Alternatively, we could define $\mathcal{T}$(**havoc** v) = **any** $\mathcal{T}$(T), where T is v's type.

Jumps (branching). In addition to structured `while` loops (discussed below), Boogie provides jump statements of the form `goto` $\mathtt{l}_1,\ldots,\mathtt{l}_n$, which nondeterministically jump to any of the locations labeled by $\mathtt{l}_k$. The translation must remove jump statements in a way that preserves verifiability; this rules out "global" approaches using a program counter [11,26], since they would require new invariants about the counter. Instead, we introduce simple heuristics that replace jumps with structured code; since the usage of jumps in Boogie programs tend to follow well-defined patterns that can be traced back to structured loops, the heuristics may be sufficient in practice.[13]

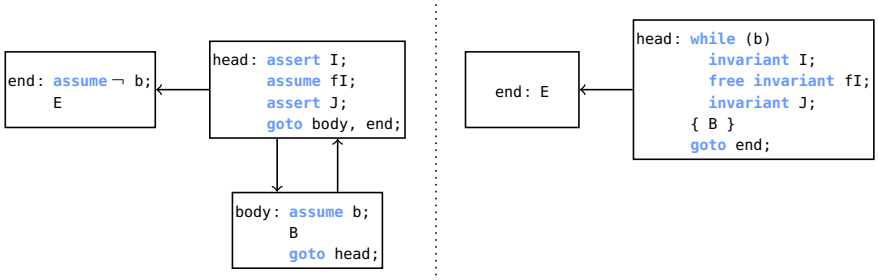

Fig. 7. Transformation of loops from unstructured (left) to structured (right).

Consider the control-flow graph G of a procedure body; each node N of G is a *simple block*: a linear piece of code with a label ℓ_N on the first statement, no labels anywhere else in N, and a `goto` as last statement or no `goto` statements at all; arrows connect N to the locations mentioned in N's `goto` statement (if N has no `goto`, we call it a *terminal* node). We apply three kinds of transformations on G exhaustively.

Sequencing: if $N \to M$ is the only arrow out of N and the only arrow into M, and $M \not\to N$, replace N and M with the single block $N\,;M$ with the `goto` at the end of N and label ℓ_M removed.

Choosing: if $N \to \{M_1,\ldots,M_n\}$ are the only arrows out of N and the only arrows into each $M_1,\ldots,M_n$, and every M_k, for $1 \le k \le n$, is a terminal node, replace $N, M_1,\ldots,M_n$ with the single block:

$$N\texttt{; if (*)\{}M_1\texttt{\} else \{if (*)\{}M_2\texttt{\} else \{}\cdots\texttt{ else \{}M_n\texttt{\}\}}\cdots\texttt{\}}$$

with the `goto` at the end of N and all labels other than ℓ_N removed.[14]

Looping: replace the subgraph of Fig. 7 (left) with the structured loop to its right.

Conditionals. The translation of conditionals is straightforward:

$$\mathcal{T}(\texttt{if (b) then \{BT\} else \{BE\}}) = \texttt{if } \mathcal{T}(\texttt{b}) \texttt{ then \{ } \mathcal{T}(\texttt{BT}) \texttt{ \} else \{ } \mathcal{T}(\texttt{BE}) \texttt{ \}}$$

[13] $\mathcal{T}$'s implementation currently does not support this translation of `goto` statements.

[14] This is after Dafny's calculational proof approach [17].

Loops. Fig. 8 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`: $\mathcal{T}(\texttt{break}) = \texttt{raise Break}$.

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

Fig. 8. Translation of a Boogie loop (left) into WhyML (right).

Procedure calls. The translation of procedure calls is straightforward; for Boogie procedure `p` in Fig. 6: $\mathcal{T}(\texttt{call r := p(e)}) = \mathcal{T}(\texttt{t}) \leftarrow \texttt{p}(\mathcal{T}(\texttt{e}))$. Since WhyML function calls translating Boogie procedures use the `val` style of declaration rather than the recursive function style (`rec`), the modular semantics of procedure calls (where the behavior is entirely determined by the specification) is correctly preserved.

Call-forall. $\mathcal{T}$ translates call-forall statements (supported in older versions of Boogie [15]) by axiomatizing their semantics:

$$\mathcal{D}(\texttt{call forall Lemma(*)}) = \texttt{assume } (\forall\ \texttt{t: T} \bullet \texttt{R(t)} \Longrightarrow \texttt{E(t)})$$

where `Lemma` is declared as `procedure Lemma(t: T) requires R(t); ensures E(t)`.

4.8 Attributes

$\mathcal{T}$ translates *triggers* using WhyML's syntax:

$$\mathcal{T}(\forall \texttt{x:X} \bullet \texttt{\{trig\} E(x)}) = \forall\ \texttt{x:}\mathcal{T}(\texttt{X})\ \texttt{[}\mathcal{T}(\texttt{trig})\texttt{]}.\mathcal{T}(\texttt{E(x)})$$

The translation discards other application-specific attributes, which have no equivalent in Why3.

4.9 Identifiers and Visibility

Boogie is more liberal than WhyML in the range of characters that are allowed in identifier names; therefore, the translation defines an injective renaming of identifiers when necessary.

Boogie allows local declarations to shadow global declarations of entities with the same name. Since WhyML does not allow shadowing, the translation introduces fresh names for local declarations when necessary to avoid name clashes with the shadowed declarations.

While the order of declarations is immaterial in Boogie, in WhyML reference must follow declaration. Thus, the translation reorders declarations to comply with WhyML's requirements; it also introduces a canonical order of declarations: types, global variables, functions, axioms, procedure declarations (**val**), procedure definitions (**let**), other declarations.

4.10 Polymorphic Maps

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

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

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 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.

Boogie vs. WhyML polymorphism. While WhyML also supports generic polymorphism, like every functional language in the ML family to which it belongs, its usage is more restrictive than Boogie's. The first difference is that *mutable* maps cannot be polymorphic in WhyML; therefore, Boogie variables of polymorphic map type require a special translation. The second difference is that, in some contexts, a variable of polymorphic map type in Boogie effectively corresponds to *multiple* maps, one for each possible concrete type, and the different maps can be combined in the same expression. Consider, for example, a $\texttt{type Mix} = \langle \alpha \rangle\, [\alpha]\alpha$ of maps from generic type α to α; Boogie accepts formulas such as $\texttt{axiom } (\forall\ \texttt{m: Mix} \bullet \texttt{m[0]} = \texttt{1} \wedge \texttt{m[true]})$ where `m` acts as a map over `int` in the first conjunct and as a map over `bool` in the secondconjunct. 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 that do not assume anything about the concrete type that will instantiate `'a`. Note that Boogie even allows expressions that introduce inconsistencies, such as $\forall\ \langle \beta \rangle\ \texttt{x: } \beta\texttt{, y: Mix} \bullet \texttt{y[x]} = \texttt{3} \wedge\ \texttt{y[x]} = \texttt{true}$ (where the quantification is also type-generic), which passes typechecking but allows one to derive false.

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

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

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

Precisely, two kinds of polymorphic maps may feature within polymorphic functions and procedures: polymorphic maps generic with respect to *explicitly declared* function or procedure parameters are similar to Why3's, and hence different from those generic with respect to implicit type parameters declared outside the function or procedure. For example, implementations of a procedure `p`$\langle\beta\rangle$`(m: Mix, n: [`β`]`β`)` can select elements of any concrete type from `m`, but only elements of parametric type β from `n`.

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 $types($`pm`$)$ of all actual types `pm` takes in expressions or assignments, as outlined in Tab. 2.[15] This in turn determines the set $types($`pM`$)$ as the union of all sets $types($`p`$)$ for `p` of type `pM`.

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

Table 2. 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.)

The types in $types($`pM`$)$ include in general both concrete and parametric types. For example, the program of Fig. 9 (left) determines $types($`m`$) = \{$`[int]int`, `[`β`]`$\beta\}$, $types($`n`$) = \{$`[bool]bool`$\}$, and $types($`M`$) = types($`m`$) \cup types($`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`$)$.

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` $\wedge$ `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

[15] A parameter's actual type is ambiguous if the parameter appears in the map type's codomain but not in its domain; in this case, Boogie defaults to type `int`.

```
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. 9. An example of how polymorphic maps (left) turn into monomorphic maps (right).

$k = 1, \ldots, n$. If $\mathbb{T}$ is a set of types obtained from the same type expression T, such as $types(\texttt{pM})$ with respect to pM's definition, and id is an identifier, let $(\mathbb{T})$ denote $\mathbb{T}$ as a tuple, and $(\texttt{id\_}\mathbb{T})$ denote the tuple of identifiers $\texttt{id\_}\boldsymbol{t}$ such that $T_{\boldsymbol{t}}$ is the corresponding type in $\mathbb{T}$. In the example of Fig. 9, if $T = [\alpha]\alpha$ then $T_{\texttt{int}} = \texttt{[int]int}$, $(types(\texttt{m})) = (\texttt{[int]int}, [\beta]\beta)$, and $(\texttt{j\_}types(\texttt{m})) = (\texttt{j\_int, j\_}\beta)$. Throughout, we also assume that an uninterpreted type $\texttt{a}_k$ is available for $k = 1, \ldots, n$, that $M_{\texttt{a}}$ denotes the type expression $\texttt{[}\mathsf{T}_1\texttt{, ..., }\mathsf{T}_n\texttt{] U}$ in (3) with each α_k replaced by $\texttt{a}_k$, and that $conc^+(\texttt{pM}) = conc(\texttt{pM}) \cup \{M_{\texttt{a}}\}$.

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

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

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 $(\texttt{g\_}conc^+(\texttt{pM}))$ with argument declared as $(\texttt{x: }(conc^+(\texttt{pM}))$; if g has multiple arguments of this kind, the desugaring is applied recursively to each variant. Fig. 9 (right) shows how the polymorphic map type M and each of the items m and n of type M become three monomorphic types and three 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 $\texttt{x: }X$. If X is a type expression that depends on $\boldsymbol{\beta}$, and there exists a map type $\texttt{[}V_1\texttt{, ..., }V_n\texttt{]}V_0$ in $types(\texttt{pM})$ such that $X = V_k$ for some $k = 0, \ldots, n$, then g becomes $(\texttt{g\_}\mathbb{V}_k)$ and x becomes $\texttt{x: }(\mathbb{V}_k)$, where $\mathbb{V}_k = \{V_k \mid \texttt{[}V_1\texttt{, ..., }V_n\texttt{]}V_0 \in conc^+(\texttt{pM})\}$ 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. 9 (right) shows how argument $\texttt{x: }\beta$ becomes three arguments of concrete types int, bool, and a, since $[\beta]\beta \in types(\texttt{M})$. Since procedure p does not use β elsewhere, we drop it from the procedure's 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 w_c such that $\mathtt{W}_c = \mathtt{C}$; otherwise, w's actual type is a *parametric* type, and we replace w with the tuple (w_X), including all variants of w that have been introduced. In Fig. 9 (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[16], which we modified to support access using the visitor pattern, AST in-place modifications, and the latest syntax of Boogie (e.g., for integer vs. real division[17]). Stages 2) and 3) are implemented by multiple AST visitors, each taking care of a particular aspect of the translation, in the style of [26]; 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.A similar technique is also advocated in [21].

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 as 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. 3 summarizes the sizes of the programs in each group.

The programs in NAT, which we developed in previous work [10,9], 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 [1] and

[16] https://github.com/martinschaef/boogieamp

[17] http://boogie.codeplex.com/discussions/397357

		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 3. 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 program and of their WHYML translation.

AutoProof [27] (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[18] we retained in TES those that mainly exercise features supported by `b2w`. This meant excluding several groups of tests that exercise special options (Houdini, assertion inference, special Z3 encodings and directives, etc.), unsupported language features (bitvectors, gotos, etc.), and the correctness of typechecking (`b2w` assumes well-formed Boogie input). It also meant excluding 4 programs that triggered Boogie errors (a Boogie *error* means here a problem with the input such as a typechecking or parsing error due to a feature not activated; it is not a verification error, which just denotes a failed verification attempt and is fair game for evaluating the translation); and another 35 programs that `b2w` failed to translate because of unsupported features that we identified a posteriori.

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, and Z3,[19] it runs Why3 with command `why3 prove -P p w`, also with a timeout of 180 seconds.[20] 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.[21]

All the experiments ran on a Ubuntu 14.04 LTS GNU/Linux box with AMD A4-5300 CPU at 3.4 GHz and 4 GB of RAM, with the following tools: Alt-Ergo 0.99.1, CVC3 2.4.1, Z3 4.3.2, Mono 4.2.1, OCaml 4.01.0, Boogie 2.3.0.61016, and Why3 0.86.2. To account for noise, we repeated each verification twice and report the mean of the running times.

Results. Tab. 4 shows a summary of the results where we compare Why3's performance with the best SMT solver against Boogie's. The most significant result is that

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

[19] We initially included CVC4 among the SMT solvers, but the version 1.5-prerelease that we tried invariably crashed in our experimental setup.

[20] The timeouts were enforced using the Unix command `timeout`. We also set had a 30-second timeout per procedure (option `/timeLimit` in Boogie) or goal (option `-T` in Why3).

[21] The number of verification goals of each program is the same in Boogie and Why3: the number of procedure implementations.

GROUP	#	B = W	B > W	B < W	0=0	50=50	100=100	SPURIOUS
NAT	29	20	9	0	1	0	19	0
OBJ	6	5	0	1	1	2	2	0
TES	159	137	21	1	71	21	45	0
Total:	194	162	30	2	73	23	66	0

Table 4. 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 these 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 these Boogie and Why3 both verify none (0=0), some but not all (50=50), or all (100=100) of the goals; the last column (SPURIOUS) demonstrates 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).

the WhyML translation produced by `b2w` behaves like the Boogie original in 85% (162, 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. While Tab. 4 does not show this piece of information, we also found one program in NAT (`inv_survey/bst`) where Why3 proves all goals (like Boogie does)only by combining the results of Alt-Ergo and Z3.

Boogie verifies more goals than Why3 in 15% (30, 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 is not fully supported by `b2w` (examples are Z3-specific annotations, which `b2w` simply drops, and `goto`, which `b2w` encodes as `assert false` to guarantee correctness). In 1% (2, B < W) of the experiments, Why3 even verifies more goals than Boogie. One program in OBJ (`rotation_by_copy`) is a genuine example where Why3's Z3 encoding is more effective than Boogie's; the one program in TES (`test2/Quantifiers`) 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. (Procedures `S`, `U0`, and `U1` use regular triggers whose translation to Why3 yields a different behavior, probably because of the triggers generated by Why3 by default differ from those generated by Boogie; procedures `W` and `X2` use negated triggers that `b2w` ignore. As this was the only program in our experiments that introduced spurious behavior, the experiments provide convincing evidence that `b2w`'s translations preserve correctness and verifiability to a large degree.

Tab. 5 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. 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-

		Z3 BOOGIE						ALT-ERGO WHY3						CVC3 WHY3						Z3 WHY3					
		OUTCOME			TIME			OUTCOME			TIME			OUTCOME			TIME			OUTCOME			TIME		
GROUP	#	μ	$\forall$	$\nexists$	μ	Σ	∞	μ	$\forall$	$\nexists$	μ	Σ	∞	μ	$\forall$	$\nexists$	μ	Σ	∞	μ	$\forall$	$\nexists$	μ	Σ	∞
NAT	29	93	25	1	0.7	21	0	63	15	6	30.4	882	1	28	1	12	0.2	7	0	73	16	5	19.1	554	0
OBJ	6	52	2	2	11.0	66	0	46	1	2	45.3	272	0	46	1	2	0.4	2	0	68	3	1	35.8	215	0
TES	159	45	55	71	0.7	104	0	37	45	85	37.7	5995	5	33	39	91	0.2	26	0	37	44	86	38.2	6067	5
Total:	194	58	82	74	1.6	191	0	54	61	93	33.3	7149	6	30	41	105	0.2	35	0	68	63	92	22.5	6836	5

Table 5. 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 $\nexists$), the mean μ and total Σ verification TIME in seconds (including time outs), and how many programs timed out.

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*. For example, a `goto` statement is rendered as `assert false`; therefore, the translated program verifies only if the `goto` is never executed in the original program. 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, 84% 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. S. Arlt and M. Schäf. Joogie: Infeasible code detection for Java. In *Proceedings of CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 767–773. Springer, 2012.
2. 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 *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

3. 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.
4. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie – an interactive prover for the Boogie program-verifier. In *Proceedings of TPHOLs*, volume 5170, pages 150–166. Springer, 2008.
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 *Lecture Notes in Computer Science*, pages 133–148. Springer, 2015.
6. M. Clochard, J. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In *Proceedings of VSTTE*, volume 8471 of *Lecture Notes in Computer Science*, pages 37–51, 2014.
7. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, pages 337–340, 2008.
8. J. Filliâtre and A. Paskevich. Why3 – where programs meet provers. In *Proceedings of ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
9. C. A. Furia. Rotation of sequences: Algorithms and proofs. `http://arxiv.org/abs/1406.5453`, June 2014.
10. C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys*, 46(3):Article 34, 2014.
11. D. Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.
12. 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 *Lecture Notes in Computer Science*, pages 451–476. Springer, 2013.
13. 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.
14. 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.
15. K. R. M. Leino. This is Boogie 2, 2008. `http://goo.gl/QsH6g`.
16. K. R. M. Leino. Developing verified programs with Dafny. In *Proceedings of ICSE*, pages 1488–1490. ACM, 2013.
17. K. R. M. Leino and N. Polikarpova. Verified calculations. In *Proceedings of VSTTE*, pages 170–190, 2013.
18. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
19. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
20. N. Polikarpova, C. A. Furia, and S. West. To run what no one has run before: Executing an intermediate verification language. In *Proceedings of RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 251–268. Springer, 2013.
21. D. Sarkar, O. Waddell, and R. K. Dybvig. Educational pearl: A nanopass framework for compiler education. *J. Funct. Program.*, 15(5):653–667, 2005.
22. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *Proceedings of FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.
23. L. Segal and P. Chalin. A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In *Proceedings of VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2012.
24. P. Stevens. A landscape of bidirectional model transformations. In *Proceedings of GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2008.

25. M. Trudel, C. A. Furia, M. Nordio, and B. Meyer. Really automatic scalable object-oriented reengineering. In *Proceedings of ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 477–501. Springer, 2013.
26. 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.
27. 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 *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.

	Boogie			Why3						
		Z3			Alt-Ergo		CVC3		Z3	
NAME	LOC	% V.	T	LOC	% V.	T	% V.	T	% V.	T
inv_survey/array_partitioning_v1	42	100	0.7	100	100	1.8	50	0.2	50	30.7
inv_survey/array_partitioning_v2	53	100	0.7	125	100	0.6	50	0.2	100	0.3
inv_survey/array_stack_reversal	125	100	0.7	204	100	0.8	86	0.3	86	30.7
inv_survey/bst	153	100	0.7	258	50	61.2	50	0.4	75	30.7
inv_survey/bubble_sort_basic	49	100	0.7	113	100	0.3	50	0.2	100	0.2
inv_survey/bubble_sort_improved	53	100	0.7	118	100	1.6	50	0.2	100	0.2
inv_survey/comb_sort	56	100	0.7	124	100	0.4	50	0.3	100	0.2
inv_survey/dutch_flag	63	100	0.7	133	50	30.2	50	0.2	100	0.2
inv_survey/insertion_sort	47	100	1.1	100	0	30.1	0	0.2	100	0.4
inv_survey/knapsack	50	100	0.6	97	100	17.8	0	0.2	100	0.2
inv_survey/Levenshtein_distance	43	100	0.6	91	100	0.8	0	0.2	100	0.2
inv_survey/max_of_array_v1	20	100	0.8	66	100	0.2	0	0.2	100	0.2
inv_survey/max_of_array_v2	20	100	0.7	66	100	0.4	0	0.2	100	0.2
inv_survey/partition	63	100	0.7	137	100	1.7	50	0.2	100	0.3
inv_survey/plateau	43	100	0.7	84	0	30.1	0	0.2	0	30.6
inv_survey/reverse	68	100	0.6	131	100	0.6	0	0.2	100	0.2
inv_survey/selection_sort	72	100	0.8	160	100	10.8	33	0.3	100	0.3
inv_survey/sequential_search_v1	28	100	0.7	72	0	30.1	0	0.2	0	30.5
inv_survey/sequential_search_v2	23	100	0.7	70	0	30.2	0	0.1	0	30.5
inv_survey/sum_of_array	21	100	0.6	62	100	0.1	100	0.1	100	0.2
inv_survey/welfare_crook	44	100	0.6	86	100	0.3	0	0.2	0	30.5
rotation/rotation_copy	57	100	0.7	128	33	60.2	33	0.2	67	30.6
rotation/rotation_copy_plain	41	100	0.6	80	0	30.1	0	0.2	100	0.3
rotation/rotation_reverse	201	90	0.8	318	40	180.0	10	0.7	80	61.3
rotation/rotation_swap-1_3	48	0	0.7	88	0	30.1	0	0.3	0	30.5
rotation/rotation_swap-2_3	175	60	0.7	201	20	120.4	20	0.3	40	91.6
rotation/rotation_swap-3_3	47	100	0.6	96	67	30.2	67	0.2	100	0.2
rotation/rotation_swap_iterative-1_2	152	100	0.7	184	33	60.2	33	0.3	67	30.6
rotation/rotation_swap_iterative-2_2	253	60	0.7	224	20	120.3	20	0.5	40	91.4
oo/autoproof_account	385	0	0.9	446	0	120.4	0	0.8	0	122.1
oo/binary_search	68	100	0.7	158	67	30.2	67	0.4	100	0.3
oo/joogie_examples	187	60	0.7	277	60	60.3	60	0.4	60	61.1
oo/joogie_helloWorld	142	50	1.3	175	50	30.3	50	0.3	50	30.8
oo/linked_list_max	44	100	0.7	90	100	0.2	100	0.2	100	0.2
oo/rotation_by_copy	52	0	62.0	99	0	30.1	0	0.2	100	0.4

Table 6. Results for the programs in groups NAT (above the horizontal line) and OBJ (below it) in the experiments. For each program (NAME) the Boogie program length in non-comment non-empty lines of code (LOC) and the length of its WHY3 translation; and, for both Boogie and Why3, for each choice of SMT solver among ALT-ERGO, CVC3, and Z3: the percentage of goals verified in each program (% V.) and the verification time (T) in seconds (with a timeout of 180 seconds).

	BOOGIE			WHY3							
	Z3			ALT-ERGO			CVC3			Z3	
NAME	LOC	% V.	T	LOC	% V.	T	% V.	T		% V.	T
doomed/doomdebug	36	0	0.7	86	0	60.2	0	0.2		0	60.9
doomed/doomed	73	43	0.8	185	43	120.4	43	0.4		43	122.4
doomed/notdoomed	43	50	0.5	107	50	60.2	50	0.2		50	60.9
doomed/smoke0	61	67	0.6	148	67	60.3	67	0.3		67	61.0
lock/Lock	86	100	0.6	163	67	30.2	67	0.3		67	30.6
lock/LockIncorrect	34	0	0.6	64	0	30.1	0	0.1		0	30.5
smoke/smoke0	41	100	0.6	108	100	0.2	100	0.2		100	0.2
snapshots/Snapshots0.v0	16	0	0.7	72	0	120.3	0	0.2		0	121.8
snapshots/Snapshots0.v1	16	50	0.6	72	50	60.3	50	0.2		50	61.0
snapshots/Snapshots0.v2	12	67	0.6	60	67	30.2	67	0.1		67	30.6
snapshots/Snapshots1.v0	10	50	0.6	48	50	30.2	50	0.2		50	30.6
snapshots/Snapshots1.v1	10	50	0.6	48	50	30.2	50	0.2		50	30.6
snapshots/Snapshots1.v2	11	50	0.6	50	50	30.2	50	0.2		50	30.5
snapshots/Snapshots10.v0	14	100	0.7	48	100	0.1	100	0.1		100	0.1
snapshots/Snapshots10.v1	14	100	0.5	48	100	0.1	100	0.1		100	0.1
snapshots/Snapshots11.v0	10	0	0.6	43	0	30.1	0	0.1		0	30.6
snapshots/Snapshots11.v1	10	0	0.6	43	0	30.1	0	0.1		0	30.5
snapshots/Snapshots12.v0	12	100	0.6	41	100	0.1	100	0.1		100	0.1
snapshots/Snapshots12.v1	12	0	0.6	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots13.v0	16	100	0.6	43	100	0.1	100	0.1		100	0.1
snapshots/Snapshots13.v1	12	0	0.6	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots14.v0	16	100	0.6	43	100	0.1	100	0.1		100	0.1
snapshots/Snapshots14.v1	16	0	0.6	43	0	30.1	0	0.1		0	30.6
snapshots/Snapshots15.v0	11	100	0.5	42	100	0.1	100	0.1		100	0.1
snapshots/Snapshots15.v1	11	0	0.5	42	0	30.1	0	0.1		0	30.5
snapshots/Snapshots16.v0	11	100	0.6	40	100	0.1	0	0.1		100	0.1
snapshots/Snapshots16.v1	11	0	0.6	40	0	30.1	0	0.1		0	30.6
snapshots/Snapshots17.v0	22	100	0.6	61	100	0.2	100	0.1		100	0.1
snapshots/Snapshots17.v1	22	0	0.6	61	0	30.2	0	0.1		0	30.7
snapshots/Snapshots18.v0	18	100	0.6	53	100	0.1	100	0.1		100	0.1
snapshots/Snapshots18.v1	18	0	0.6	53	0	30.1	0	0.1		0	30.6
snapshots/Snapshots19.v0	8	0	0.6	39	0	30.1	0	0.1		0	30.5
snapshots/Snapshots19.v1	8	0	0.6	39	0	30.1	0	0.1		0	30.5
snapshots/Snapshots2.v0	9	100	0.6	38	100	0.1	100	0.1		100	0.1
snapshots/Snapshots2.v1	9	100	0.6	38	100	0.1	100	0.1		100	0.1
snapshots/Snapshots2.v2	10	100	0.6	40	100	0.1	100	0.1		100	0.1
snapshots/Snapshots2.v3	10	100	0.7	40	100	0.1	100	0.1		100	0.1
snapshots/Snapshots2.v4	10	100	0.6	40	100	0.1	100	0.1		100	0.1
snapshots/Snapshots2.v5	11	100	0.5	42	100	0.1	100	0.1		100	0.1
snapshots/Snapshots20.v0	16	0	0.6	44	0	30.1	0	0.1		0	30.6
snapshots/Snapshots20.v1	16	0	0.6	44	0	30.1	0	0.1		0	30.5
snapshots/Snapshots21.v0	13	0	0.6	41	0	30.1	0	0.1		0	30.6
snapshots/Snapshots21.v1	13	0	0.6	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots22.v0	13	0	0.6	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots22.v1	13	100	0.5	41	100	0.1	100	0.1		100	0.1
snapshots/Snapshots23.v0	17	50	0.7	52	50	30.2	50	0.1		50	30.5
snapshots/Snapshots23.v1	18	50	0.6	53	50	30.2	50	0.2		50	30.5
snapshots/Snapshots23.v2	17	50	0.6	52	50	30.2	50	0.1		50	30.5
snapshots/Snapshots24.v0	23	0	0.6	51	0	30.1	0	0.1		0	30.5
snapshots/Snapshots24.v1	23	0	0.6	51	0	30.1	0	0.1		0	30.6
snapshots/Snapshots25.v0	11	0	0.6	47	0	30.1	0	0.1		0	30.5
snapshots/Snapshots25.v1	11	0	0.6	47	0	30.1	0	0.1		0	30.5
snapshots/Snapshots26.v0	11	0	0.6	47	0	30.1	0	0.1		0	30.5
snapshots/Snapshots26.v1	12	0	0.6	48	0	30.1	0	0.1		0	30.5
snapshots/Snapshots27.v0	11	0	0.6	47	0	30.1	0	0.1		0	30.5
snapshots/Snapshots27.v1	13	0	0.6	51	0	30.1	0	0.1		0	30.5
snapshots/Snapshots28.v0	11	100	0.5	48	100	0.1	100	0.1		100	0.2
snapshots/Snapshots28.v1	12	0	0.6	48	0	30.1	0	0.1		0	30.5
snapshots/Snapshots29.v0	11	100	0.6	47	0	30.1	0	0.1		0	30.6
snapshots/Snapshots29.v1	11	0	0.6	47	0	30.1	0	0.1		0	30.5
snapshots/Snapshots3.v0	13	100	0.6	41	100	0.1	100	0.1		100	0.1
snapshots/Snapshots3.v1	13	0	0.6	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots30.v0	11	0	0.6	42	0	30.1	0	0.1		0	30.5
snapshots/Snapshots30.v1	12	0	0.6	43	0	30.1	0	0.1		0	30.5
snapshots/Snapshots31.v0	12	100	0.5	44	100	0.1	100	0.1		100	0.1

	Boogie			Why3							
	Z3			Alt-Ergo			CVC3			Z3	
NAME	LOC	% V.	T	LOC	% V.	T	% V.	T		% V.	T
snapshots/Snapshots31.v1	11	0	0.7	43	0	30.1	0	0.2		0	30.6
snapshots/Snapshots32.v0	12	100	0.6	44	100	0.2	100	0.1		100	0.1
snapshots/Snapshots32.v1	9	0	0.7	41	0	30.1	0	0.1		0	30.5
snapshots/Snapshots33.v0	12	100	0.6	44	100	0.1	100	0.1		100	0.2
snapshots/Snapshots33.v1	6	100	0.6	37	100	0.1	100	0.1		100	0.1
snapshots/Snapshots34.v0	6	100	0.5	38	100	0.1	100	0.1		100	0.1
snapshots/Snapshots34.v1	5	0	0.6	36	0	30.1	0	0.1		0	30.6
snapshots/Snapshots35.v0	6	100	0.5	38	100	0.1	100	0.1		100	0.1
snapshots/Snapshots35.v1	5	0	0.7	36	0	30.1	0	0.2		0	30.6
snapshots/Snapshots36.v0	11	100	0.6	44	100	0.1	0	0.1		100	0.1
snapshots/Snapshots36.v1	11	0	0.6	44	0	30.1	0	0.1		0	30.5
snapshots/Snapshots37.v0	7	100	0.6	42	100	0.1	0	0.1		100	0.2
snapshots/Snapshots37.v1	7	0	0.6	42	0	30.1	0	0.1		0	30.6
snapshots/Snapshots38.v0	10	100	0.6	43	100	0.1	100	0.1		100	0.1
snapshots/Snapshots38.v1	11	0	0.6	44	0	30.1	0	0.1		0	30.5
snapshots/Snapshots38.v2	11	100	0.7	44	100	0.1	100	0.1		100	0.1
snapshots/Snapshots39.v0	10	100	0.7	43	100	0.1	100	0.1		100	0.1
snapshots/Snapshots39.v1	11	0	0.6	44	0	30.1	0	0.1		0	30.5
snapshots/Snapshots39.v2	11	100	0.7	44	100	0.1	0	0.1		100	0.1
snapshots/Snapshots4.v0	23	100	0.6	64	100	0.2	100	0.1		100	0.2
snapshots/Snapshots4.v1	27	50	0.6	76	50	60.3	50	0.2		50	61.0
snapshots/Snapshots40.v0	11	0	0.6	44	0	30.1	0	0.1		0	30.5
snapshots/Snapshots40.v1	12	0	0.6	45	0	30.1	0	0.1		0	30.5
snapshots/Snapshots40.v2	12	0	0.6	45	0	30.1	0	0.1		0	30.5
snapshots/Snapshots41.v0	31	40	0.6	99	40	90.3	40	0.2		40	91.3
snapshots/Snapshots41.v1	31	40	0.7	100	40	90.3	40	0.3		40	91.4
snapshots/Snapshots5.v0	9	100	0.6	39	100	0.1	100	0.1		100	0.1
snapshots/Snapshots5.v1	9	0	0.6	39	0	30.1	0	0.1		0	30.5
snapshots/Snapshots6.v0	12	100	0.6	43	100	0.1	100	0.1		100	0.1
snapshots/Snapshots6.v1	12	0	0.5	43	0	30.1	0	0.1		0	30.5
snapshots/Snapshots7.v0	14	100	0.5	45	100	0.1	100	0.1		100	0.1
snapshots/Snapshots7.v1	14	100	0.6	45	100	0.1	100	0.1		100	0.2
snapshots/Snapshots8.v0	11	100	0.7	45	100	0.1	100	0.1		100	0.1
snapshots/Snapshots8.v1	11	100	0.6	45	100	0.1	100	0.1		100	0.1
snapshots/Snapshots9.v0	13	100	0.7	47	100	0.1	100	0.1		100	0.1
snapshots/Snapshots9.v1	11	100	0.6	45	100	0.1	100	0.1		100	0.2
test13/ErrorTraceTestLoopInvViolationBPL	19	0	0.6	86	0	90.2	0	0.2		0	91.4
test15/CaptureState	23	0	0.7	62	0	30.2	0	0.2		0	30.5
test15/InterpretedFunctionTests	15	0	0.6	66	0	90.2	0	0.2		0	91.4
test15/IntInModel	3	0	0.6	36	0	30.1	0	0.1		0	30.6
test15/ModelTest	10	0	0.6	49	0	30.1	0	0.2		0	30.7
test15/NullInModel	5	0	0.6	39	0	30.1	0	0.1		0	30.6
test16/LoopUnroll	63	0	0.6	124	0	90.2	0	0.2		0	91.5
test17/contractinfer	21	0	0.8	68	0	60.2	0	0.2		0	60.9
test2/AssertVerifiedUnder0	26	50	0.7	100	0	180.0	0	0.3		0	180.0
test2/AssumeEnsures	53	57	0.8	124	57	90.4	57	0.3		57	91.8
test2/AssumptionVariables0	44	50	0.7	137	0	180.0	0	0.3		0	180.0
test2/Axioms	24	67	0.7	73	67	30.2	67	0.2		67	30.7
test2/B	65	100	0.6	112	0	120.3	0	0.2		0	122.0
test2/Call	49	40	0.7	117	20	120.3	20	0.3		20	122.0
test2/ContractEvaluationOrder	26	25	0.7	101	25	90.2	25	0.2		25	91.7
test2/CutBackEdge	35	20	0.7	96	0	150.3	0	0.2		0	152.3
test2/Ensures	61	50	0.8	168	50	150.5	50	0.5		50	152.5
test2/False	14	100	0.6	54	100	0.2	100	0.2		100	0.2
test2/FormulaTerm2	36	50	0.7	104	50	60.3	50	0.2		50	61.1
test2/FreeCall	59	64	0.8	185	27	180.0	27	0.3		27	180.0
test2/Implies	28	0	0.8	97	0	150.3	0	0.3		0	152.6
test2/InvariantVerifiedUnder0	42	17	0.8	146	0	180.0	0	0.2		0	180.0
test2/LoopInvAssume	15	0	0.7	44	0	30.1	0	0.1		0	30.6
test2/Passification	155	64	0.7	290	18	180.0	18	0.5		18	180.0
test2/Quantifiers	122	57	0.9	254	86	60.7	64	0.5		93	31.0
test2/SelectiveChecking	31	25	0.6	121	0	120.3	0	0.3		0	122.0
test2/sk_hack	17	100	0.6	44	0	30.1	0	0.1		0	30.6
test2/Timeouts0	71	0	3.4	156	0	90.2	0	0.3		0	91.7
test2/TypeEncodingM	19	0	0.6	60	0	30.1	0	0.2		0	30.6

	BOOGIE			WHY3							
	Z3			ALT-ERGO			CVC3			Z3	
NAME	LOC	% V.	T	LOC	% V.	T	% V.	T		% V.	T
test21/BooleanQuantification2	9	0	0.6	46	0	30.1	0	0.1		0	30.6
test21/Boxing	15	0	0.8	49	0	30.1	0	0.1		0	30.6
test21/Casts	7	0	0.6	48	0	30.1	0	0.1		0	30.6
test21/Colors	13	0	0.7	62	0	60.2	0	0.2		0	61.2
test21/DisjointDomains	21	0	0.8	81	0	90.2	0	0.3		0	91.6
test21/EmptySetBug	18	0	0.9	54	0	30.1	0	0.2		0	30.7
test21/FunAxioms	24	50	0.8	81	50	30.2	50	0.2		50	30.7
test21/FunAxioms2	13	0	0.7	49	0	30.1	0	0.1		0	30.6
test21/InterestingExamples3	17	67	0.7	71	33	60.2	33	0.2		33	61.2
test21/InterestingExamples5	9	100	0.7	45	100	0.1	0	0.1		100	0.2
test21/Keywords	5	100	0.7	38	100	0.1	100	0.1		100	0.1
test21/LargeLiterals0	12	0	0.7	46	0	30.1	0	0.1		0	30.7
test21/LetSorting	11	100	0.6	43	0	30.1	0	0.1		0	30.6
test21/Maps2	14	100	0.6	52	100	0.1	0	0.1		0	30.7
test21/Orderings	13	50	0.7	59	0	60.2	0	0.2		0	61.0
test21/Orderings2	11	0	0.8	49	0	30.1	0	0.1		0	30.6
test21/Orderings3	22	0	0.7	78	0	60.2	0	0.2		0	61.2
test21/Orderings4	7	0	0.8	47	0	30.1	0	0.1		0	30.7
test21/PolyList	35	0	0.8	91	0	60.2	0	0.2		0	61.1
test21/Triggers0	34	50	0.7	92	50	30.2	50	0.2		50	30.7
test21/Triggers1	12	0	0.8	50	0	30.1	0	0.1		0	30.6
test7/MultipleErrors	14	0	0.6	42	0	30.1	0	0.1		0	30.6
test7/NestedVC	20	50	0.5	61	0	60.2	0	0.2		0	61.0
test7/UnreachableBlocks	34	100	0.6	79	50	60.3	50	0.2		50	61.2
textbook/Bubble	47	100	0.7	110	0	30.2	0	0.3		0	30.7
textbook/DutchFlag	47	100	0.8	92	0	30.1	0	0.2		0	30.6
textbook/Find	27	100	0.8	72	50	30.2	50	0.2		50	30.6
textbook/McCarthy-91	11	100	0.6	47	100	0.2	100	0.1		100	0.1
textbook/TuringFactorial	27	100	0.7	81	0	30.1	0	0.2		0	30.6

Table 7: Results for the programs in group TES in the experiments. The measures are the same as in Tab. 6.