Chair of Software Engineering

Supporting multiple proof engines by translating between intermediate verification languages

Master Thesis

Michael Salar Ameri
ETH Zurich
mameri@student.ethz.ch

March 1, 2015 - September 1, 2015

Supervised by:
Dr. Carlo Alberto Furia
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Automatic verification of code against functional properties is an important part of software engineering. A widely used approach are intermediate verification languages, or IVLs. Programs from high-level languages such as Eiffel, Java or C are encoded using IVLs. Verifiers exist that generate verification conditions from IVLs; the verification condtitions can in turn be checked by back-end provers - either SMT solvers such as Z3 or interactive solvers such as Isabelle. The two most widely used IVLs are Boogie and Why. While Boogie relies mainly on SMT solvers (Z3 in particular), Why is geared towards a variety of different back-end solvers, including SMT and interactive ones. This thesis presents the design and implementation of a translator which takes Boogie programs and produces semantically equivalent Why programs. This allows programmers to use Why and its powerful verification system as a back-end alternative to Boogie. Even if the translation does not currently fully support a small number of features of the Boogie language, it is usable in practice on a variety of examples. To demonstrate, we evaluated the translator on 19 benchmark programs, including bubble sort, linear search and binary search trees. The translator produces semantically equivalent Why programs for 18 of the benchmarks programs, which can be verified automatically using the Why prover.

Acknowledgments

First I would like to thank Dr. Carlo A. Furia for supervising my master's thesis and providing helpful discussions and advice throughout. My gratitude also goes to Prof. Dr. Bertrand Meyer for providing me the opportunity to work on this thesis.

Contents

Chapter 1

Introduction

Static verification of software systems is a complex task within software engineering. To make verification of programs written in high level languages such as Java, C, and Eiffel feasible, several approaches have been proposed and implemented [20][39][7][8]. To separate such programming languages from logic used by theorem provers, intermediate verification languages (henceforth IVL) are often used to encode programs [24][40][23]. Verifiers can then generate verification conditions from IVLs, which can in turn be checked by theorem provers. Two classes of theorem provers are *interactive provers* [3] and *satisfiability modulo theories (SMT) solvers*. As the name suggests, interactive provers rely on human input and aid in checking verification conditions. Prominent examples of such provers are Isabelle-HOL [37], PVS [30] and Coq [3]. SMT solvers on the other hand work automatically by trying to solve decision problems in first order logic. Widely used examples are Z3 [9], CVC4 [2] and Alt-Ergo [4].
Two prevalent IVLs are Boogie2 [33] and Why3 [14]. Boogie relies mainly on SMT solvers, and is particularly geared towards Z3. The only available interactive prover is HOL-Boogie [7], which is supported only when Boogie is used with a specific C front-end. On the other hand many systems have been developed to translate different programming languages into Boogie, such as Dafny [22] or AutoProof [40] for Eiffel. Why currently supports at least 16 SMT solvers and 3 interactive provers as back-ends [18]. While systems such as Krakatoa for Java [13] translate high-level programming languages into Why, their number is not as high as for Boogie.
The aim of this thesis is to design and implement an automatic translation mechanism from Boogie programs to Why programs. Our translation mechanism will make it possible to use the Why system as a back-end for verifiers that output Boogie code. Therefore systems which currently translate into Boogie could make use of Why's extensive list of back-end provers, including interactive ones, without the need of any alterations. Furthermore, each verification system may generate different verification conditions, which can again make a program more amenable to verification.
The two main challenges of the translation are:

1. **Semantic equivalence.** Boogie is imperative, while WhyML is a func-

tional language with some imperative features. We must be certain that our translation mechanism produces semantically equivalent code, even if a construct from Boogie is not available in WhyML. In other words, our translation must be sound: Proven correctness of the translated program must imply correctness of the original Boogie program.

2. **Verifiability.** Given a Boogie program which is correct and sufficiently annotated with specifications, our translator should produce WhyML programs which are amenable to verification.

The first point is very important, as an unsound translation would be of no real use. Some features –such as axioms– can be translated one to one, while others –such as recursive functions, frame clauses or procedure declarations– must be axiomatized or encoded with different structures. To ensure the second point, our translator must be able to include all important specifications at the right points. These include pre- and postconditions, invariants, and triggers.

The translator in its current state supports most Boogie features, such as procedures with contracts, functions, and constant declarations with parent relations. Currently not implemented in our translator is support for the following constructs: attributes, bitvectors, gotos, and polymorphic maps. Because of these limitations, the tool cannot handle Boogie programs generated from verification systems such as AutoProof [28], as they rely on polymorphic maps to encode heaps. Nevertheless, we show how these constructs can be translated, so support can be added in the future.

To demonstrate the usability of our translator, we translate 19 different benchmark programs. The benchmarks consist of annotated implementations of well known algorithms such as bubble sort, an automatically generated program from Autoproof, and some programs which contain different Boogie features. Of all the programs we tested 18 are translated into semantically equivalent WhyML programs and 17 are automatically verified using the Why3 system.

Outline. Chapter 2 gives a short introduction into Boogie2 and Why3. The next chapter 3 demonstrates the translator design and most important implementation details. Subsequently, chapter 4 explains the important steps of the translation in detail. It serves as a reference of how the translation operates, and argues that the resulting WhyML code is semantically equivalent to the input code and therefore the translation is correct. Chapter 5 serves as a manual for the translators API and CLI interfaces with usage examples, while also explaining how the API may be extended in the future. Chapter 6 first evaluates the translator based on benchmarks to show the translated code is also amenable to verification, and then demonstrates potential use cases in practice. Finally, chapter 7 draws conclusions on the design and implementation, before stating possible future work.

Chapter 2

Background

This chapter serves as a quick introduction into Boogie and Why. Some more refined details of both are presented in chapter 4 as we present the translation and reason about its correctness.

2.1 Boogie

Boogie, currently in its second version, is an intermediate verification language. Boogie is also the name of the verification tool which takes Boogie language programs as input and generates verification conditions, which are passed to a backend prover, the SMT solver Z3 by default [25].
In this section we introduce the main features of the Boogie language, which is a combination of imperative constructs, and specifications in first-order logic. A detailed description of the language is available at [33], while shorter introductions can be found at [40] and [41].
Boogie has 4 inbuilt types (**`int`**, **`real`**, **`bool`**, and bit–vectors), a map type, and the ability for a user to define types. Bit–vectors are written as x`bv`y, where x is the decimal value of the bit–vector and y is its length.
Figure 2.1 demonstrates a newly defined type `building`, and the usage of inbuilt and map types for the declared variable `inhabited`.
The imperative part of the language consists of procedures, implementations, local and global mutable variables, assignments, if-then-else structures, while loops, and gotos. Procedure declarations can be annotated with specifications such as preconditions (**`requires`**), postconditions (**`ensures`**), and a list of global variables which the procedure might modify (**`modifies`**). Pre- and postconditions have the option to be marked as **`free`** , in which case they may be assumed when appropriate, but do not have to be checked when otherwise necessary. Loops may be annotated with invariants, which also have the possibility to be **`free`**.
`havoc` is a construct which introduces non-determinism into programs, by assigning arbitrary, blindly chosen values to variables. These values may be bound or checked using **`assume`** and **`assert`** statements, respectively. Figure 2.2 demonstrates their usage.

```
type building;
const white_house: building;
function size(b: building): real;
axiom (∀ b:building • size(b) ≥ 0.0);
axiom (size(white_house) = 5100.0)
var inhabited: [building]bool;
procedure move_in(b:building)
   requires ¬(inhabited[b]);
   ensures inhabited[b];
   modifies inhabited;
{
      inhabited[b] := true;
}
```

Figure 2.1: A simple Boogie program.

```
...
var i,j: int;
//assign values
i := −1;
j := −2;
//randomize values
havoc i,j;
//bound values
assume i ≥ 0;
assume j > i;
//check values
assert j > 0;
...
```

Figure 2.2: Usage of non-determinism within Boogie programs.

2.2 Why

Why3 [17] is another platform for program verification. Its language is called WhyML. The verification tool offers a rich API, so it can support many different back-end provers. A detailed description of Why3 and WhyML is given at [5]. Here we present the most important features of the platform and language which are needed for the translation.
WhyML does not contain any inbuilt types. Instead, the Why3 platform has a standard library which contains theories. These theories introduce and axiomatize the most widely used types, such as `map`, `int`, `real`, `bool`, `list`. A WhyML program is called a module. An example module is presented in figure 2.3. Just as in Boogie, WhyML contains specification structures. In contrast to Boogie, WhyML is mainly a functional programming language with some imperative structures.
The specification structures precondition (**`requires`**), postcondition (**`ensures`** or **`returns`**), loop **`invariant`**, and **`assert`** and **`assume`** statements behave just as their Boogie counterparts. Frame clauses are specified differently, namely using **`writes`** and **`reads`** clauses. Addiotionally, loops and recursive functions may contain a **`variant`** to prove their termination.
`let` expressions, while technically being functions, behave similarly to Boogie procedures. They contain input and output arguments (which may be an empty tuple `()`) and can be annotated with specifications. Their body may also contain imperative structures such as loops and if-then-else statements.
Figure 2.4 shows the difference between a global variable (`counter`) and an abstract function. Abstract functions return a new non-deterministic value each time they are invoked. The non-determinism may be bounded with pre- and postconditions. In essence, they behave like procedure declarations without a body in Boogie. For global variables, we use the imported `ref` library, which is simply a wrapper for mutable types. The contents of such a type may be accessed with an exclamation mark (!), and assigned to with the usual syntax (:=), as seen in figure 2.3.

```
module Building_example
   use import map.Map
   use import real.Real
   use import bool.Bool
   use import ref.Ref
   type building
   constant white_house: building
   function size(b: building): real
   axiom A1: forall b:building . (size b) ≥ 0.0
   axiom A2: (size white_house) = 5100.00
   val inhabited: ref (map building bool)
   let move_in (b:building) : ()
      requires {not !inhabited[b]}
      ensures {!inhabited[b]}
      writes {inhabited} (*optional in WhyML*)
   =(
      inhabited := (set !inhabited b true)
   )
end
```

Figure 2.3: A manual translation of the Boogie program in figure 2.1.

```
...
val counter: ref int
val havocBool(): bool
...
```

Figure 2.4: Difference between global variable declaration and an abstract function.

Chapter 3

Translator Design

This chapter first presents an overview of how the translator from Boogie to WhyML is designed. Then the most important technical choices of the implementation are demostrated.

3.1 Translator Overview

The Boogie to WhyML translator operates in four main steps.

1. Parse Boogie code and generate an abstract syntax tree (AST).
2. Manipulate and rewrite parts of the AST based on predefined rules.
3. Translate the AST into a semantically equivalent WhyML AST.
4. Output WhyML code based on the translated AST.

3.1.1 Step 1 - Parser and AST generation

In the first step an abstract syntax tree must be created of the Boogie program. As a grammar reference, the paper "This is Boogie 2" [33] is used. Some syntax has changed with respect to the paper, and we adjusted the parser to take these changes into account. Several (open source) tools already exist which produce such an AST in different programming languages [31][32][36]. For the Boogie parsing part of this project, most of the code from [36] is reused. Some adjustments were necessary to support some newer Boogie syntax. BoogieAmp and these changes are presented and detailed in section 3.2.3.

3.1.2 Step 2 - Boogie AST

Boogie has several constructs without a direct equivalent in WhyML. In this step, these constructs are rewritten based on predefined rules, using only con-

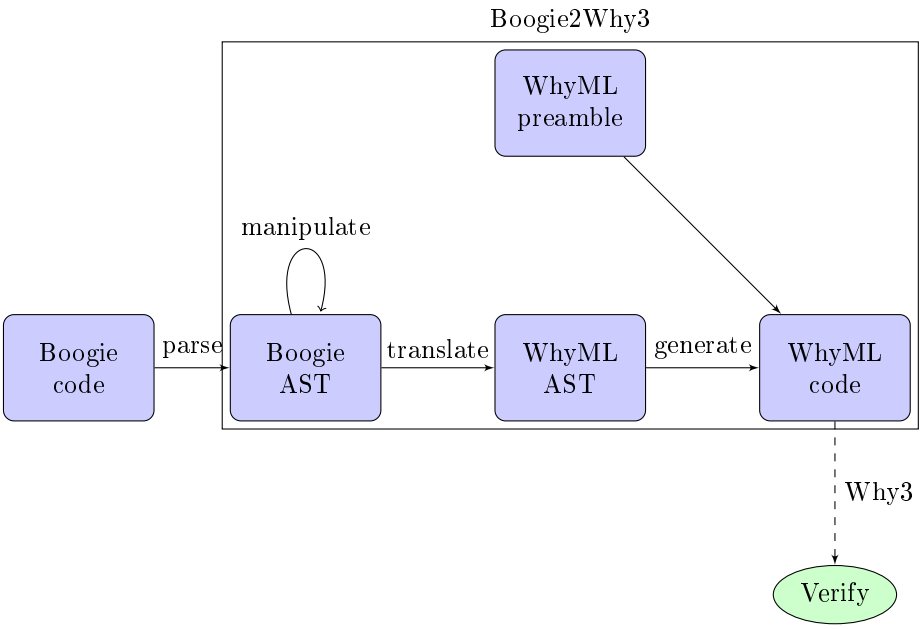

Figure 3.1: Overview of the translation mechanism.

structs which are available in WhyML. This Boogie to Boogie transformation step facilitates the subsequent steps by producing Boogie statements with a more direct correspondence to WhyML statements. It is important that the semantics of the Boogie program remain the same after this step.

3.1.3 Step 3 - WhyML AST

This step generates a new WhyML AST from the Boogie AST. Chapter 4 describes in detail how each construct is translated. The goal is not only for the translation to be semantically equivalent, but also verifiable using the Why3 platform.

3.1.4 Step 4 - WhyML code

In this last step the generated WhyML AST is traversed and corresponding WhyML code is produced. The outputted code can be stored in a file and subsequently verified using the Why3 system and its back-end provers.

3.2 Implementation Details

The translation tool is implemented in Java using the Visitor Pattern [27] for most interesting parts.

3.2.1 Visitor Pattern

Both the Boogie and WhyML abstract syntax tree can be accessed using visitors. The implementation makes heavy use of this. The Boogie AST is traversed n consecutive times (where currently $n = 8$), each time completing a further step of the translation. A similar approach is used by Trudel et al. [38] to automatically translate C into Eiffel. This has the benefit of keeping the translation very modular, as a step can easily be added (or replaced) by creating a new visitor or (adjusting an existing one) - see section 5.2.3 for more details and examples. A concern might be added run time, however in practice the time of the translation seems to remain within a couple seconds - tested with programs with up to 1000 lines of code.

3.2.2 Formula, Term, Expression

Boogie expressions are split into three groups in the WhyML syntax: formula, term, and expression [17]. When the translator reaches a Boogie expression in the AST, it must know which of the three possibilities it should generate for the WhyML AST. This information relies on the context and is stored in the private field `formulaTermExpression` of the class `BoogieTranslator`.

3.2.3 BoogieAmp

To parse Boogie files and generate abstract syntax trees the open source project BoogieAmp [36] is used. Next to an AST generator, it also offers a `TypeChecker` class which adds information about the type to each node.We made several modifications to the source code of BoogieAmp to meet our requirements. The new code can be found at [1]. Boogie2Why3 uses BoogieAmp as a precompiled library.

Visitor Pattern. The generated AST is extended to accept visitors, such that accessing it matches the style of the rest of the implementation.

New Syntax. Boogie has introduced several changes to its syntax since compared to the initial description in [33], such as the handling of integer vs real division [26]. Since BoogieAmp uses the original syntax, we made several adjustements to be compatible with the new syntax.

AST mutability. Several *setter* methods were added to the AST nodes. This allows the nodes to be modified after creation, such as renaming variables or replacing nodes altogether.

Chapter 4

Translation Scheme

This chapter explains the important steps of the translation in detail. It serves as a reference of how the translation operates, and demonstrates why the resulting code is semantically equivalent to the input code and therefore the translation is correct[1].
Once a Boogie file is parsed and a corresponding AST is generated using BoogieAmp's API, the AST is modified in several steps in preparation of the translation to WhyML. Then the actual translation into a WhyML AST is performed in several steps, before the resulting code is finally printed to a file. All these steps are described in detail in this chapter.

4.1 Identifier renaming

The first step is to rename identifiers based on two criteria:

- Using characters which are allowed in WhyML. The allowed characters represent a subset of the ones which are allowed by Boogie. [17][33]
- Explicitly differentiating shadowed Identifiers. Boogie lets procedures shadow (among other things) global variables with input arguments, which is not allowed in WhyML [33]. Therefore such identifiers are given a new and unique name.

This part of the translation is handled by the class `BoogieIdentRenaming` which implements Boogie's `ASTVisitor`. An example can be found in figures 4.1 and 4.2.
Correctness. Identifiers are renamed consistently, and variable shadowing is made explicit. Therefore the semantics of the code is not altered in any way.

[1] Modulo bugs in the implementation

```
50 //shadowed x, unsupported char '.'
51 var x: bool;
52 procedure p (x: int) returns (y.:int)
53 ensures x = y.;
54 {
55   y. := x;
56 }
```

Figure 4.1: Shadowed variable x and unsupported character ('.').

```
57    ...
58    val __x : (ref bool )
59
60    let __p_IMPL0 (____x: int ) : int
61       returns{|____y_DOT_  → (( ____x ) = ( ____y_DOT_ )) }
62    =(
63       let ____y_DOT_ = ( ref (( havoc ((
      _GLOBAL_LOCAL___p_____y_DOT_ ( () ))))))in
64       assume { true };
65       try(
66           ____y_DOT_ .contents ←  ____x ;
67          assume { true }
68       )with
69          |Return  → assume { true }
70       end ;
71        ____y_DOT_.contents
72    )
73    ...
```

Figure 4.2: Shadowed variable x is renamed and unsupported character ('.') is replaced.

```
74 type T;
75 const unique a,b,c:T;
76 const d:T;
```

Figure 4.3: Four declared constants of the same type `T`, whereof 3 are unique.

4.2 Type checking

The next step is to use BoogieAmp's `TypeChecker` class to fill out information about the concrete types in the nodes. It is important to perform this step after all identifier names are final. Any application that modifies the Boogie AST should re-run the type checker before feeding the AST to the translator.
In the current implementation, this is the last step before a WhyML abstract syntax tree is created. The following sections describe how each Boogie construct is translated into a corresponding node in the WhyML AST. Unless otherwise noted, the implementation can be found in the `BoogieTranslator` class.

4.3 Constants

Constant declarations in Boogie have two properties which must be considered during translation:

1. Uniqueness
2. Order specification

A constant declaration which contains the keyword **`unique`** potentially leads to new axioms in the translation, as seen in figures 4.3 and 4.4. For each pair of unique constants of the same type, an inequality clause is axiomatized.
Correctness. "Declaring a constant with **`unique`** makes manifest that the constant has a value that is different from the values of other unique constants of the same type." [33] Therefore a new axiom must be introduced which contains an inequality clause for each pair of declared unique constants of the same type.

When translating the order specification, four cases must be handled:
Empty parent–edge. A constant `c` of type `T` declared with an empty parent–edge[2] introduces one new axiom into the translation, depicted in figure 4.5.

[2]Note that a non-existent parent–edge is not the same as an empty one. A non–existent parent–edge means nothing further is known about the parents, which is the default case also for Why3.

```
77 ...
78    type __T
79    constant __a: __T
80    constant __b: __T
81    constant __c: __T
82    constant __d: __T
83    axiom C0: (( __a ) ≠ ( __b ))
84    axiom C1: (( __a ) ≠ ( __c ))
85    axiom C2: (( __b ) ≠ ( __c ))
86 ...
```

Figure 4.4: Translation of code in figure 4.3. For n unique constants of the same type, the translation introduces $\binom{n}{2}$ axioms, each containing one inequality clause. Note that constant `d` was not declared as unique, and is therefore not compared to the other constant, even though it is of the same type.

```
87 constant c
88 axiom C1: forall a:T. (a≠c) → not c <: a
```

Figure 4.5: Axiom to describe an empty parent edge for constant `c`.

Non–empty parent–edge. A constant `c` of type `T` declared with n parents p_1, ..., p_n introduces one new axiom, depicted in figure 4.6.

Complete parent relation. The **`complete`** keyword is analogous to the parent relation, but for children. Axiomatization is presented in figures 4.7 and 4.8.

Unique parent. The **`unique`** parent relation states that the subtrees of two different children of a node are distinct. Given n different constants which have the same parent constant declared as unique, the translation introduces $\binom{n}{2}$ new axioms to produce equivalent semantics. The translation mechanism is illustrated in figures 4.9 and 4.4.

Correctness. The correctness of the axiomatization follows by definition of the keywords.

```
89 constant c, p1, ..., pn: T
90 axiom A1 : forall t : T. c <: t → c=t || p1 <: t || ... || pn <: t
```

Figure 4.6: Axiom to describe a non-empty parent edge for constant `c`.

```
91 const c: T <: complete;
92 const p_1: T <:c;
93 ...
94 const p_n: T <:c;
```

Figure 4.7: Example usage of the **complete** keyword within a parent relation.

```
95 constant c, p_1, ... , p_n: T
96 axiom A1: forall t : T. t <: c → c=t || t <: p_1 || ... || t <: p_n
```

Figure 4.8: Axiomatization of the constant declaration in figure 4.7.

4.4 Functions

Boogie allows recursive functions to be non-terminating, whereas in WhyML each defined function must provably terminate [6]. Therefore functions cannot be translated one to one, but are instead axiomatized as demonstrated in figures 4.11 and 4.12. If-then-else structures are turned into implications while other expressions are turned into equalities within the axioms.

Correctness. The axiomatization ensures that each use of a function complies with the same constraints as in the original code. Whatever is provable within the translation must therefore also hold true in the original.

```
97 const p: T <: complete;
98 const c_1: T <:unique p;
99 ...
100 const c_n: T <:unique p;
```

Figure 4.9: Example usage of the **unique** keyword within a parent relation.

```
constant p, c_1, ..., c_n: T
axiom A1 : forall x,y: T. x <: c_1 && y <: c_2 && c_1 != c_2 → x != y
...
axiom An2 : forall x,y: T. x <: c_{n-1} && y <: c_n && c_{n-1} != c_n →
    x != y
```

Figure 4.10: Axiomatization of the constant declaration in figure 4.9.

```
function f(args:argsType): returnType
  {if cond then exp0 else exp1}
```

Figure 4.11: Possibly non-terminating function in Boogie.

4.5 Procedure and Implementation Declarations

When translating procedures and implementations, several points are of importance:

4.5.1 Separation of signature and bodies

In Boogie, each procedure consists of an optional body, a signature and m contracts ($m \geq 0$). Additionally, $n \geq 0$ implementations can reference a procedure declaration. Each implementation acts as a separate and independent body of the procedure. Implementation declarations are allowed to make some syntactic changes to the signature, but must keep the semantics unchanged [33]:

- **Allowed syntactic changes**
  - Reorder type arguments.
  - Consistently rename type arguments.
  - Rename in- and out-parameters.
- **Semantics inherited from procedure declaration**
  - Specifications (i.e. pre- and post-conditions and modifies clauses).
  - Where-clauses.

The translation explicitly separates a procedure and all its implementations as seen in figures 4.13 and 4.14. Any renamed type arguments or in- and out-parameters are given their original name from the procedure declaration in

```
axiom A0: forall args:argsType.
  (cond → (f(args) = exp0)) &&
  ((not cond) → (f(args) = exp1))
```

Figure 4.12: Axiomatized translation method of possibly non-terminating function.

```
110 procedure p (xin: int) returns (result: int)
111 ensures xin ≤ result;//verified by each implementation
112 {
113   assume xin ≤ result;
114 }
115
116 implementation p(xin2:int) returns (r:int)
117 {
118   assume xin2 ≤ 4;
119   r := 5;
120 }
121
122 implementation p(xin3:int) returns (res:int)
123 {
124   res := xin3;
125 }
```

Figure 4.13: Procedure and implementation declarations in Boogie. Each implementation must fulfill the same contracts, but may rename parameters.

order to be consistent with the contracts[3]. For each procedure declaration, a new abstract function is generated with the corresponding signature and contracts (contract translation is described in more detail in section 4.5.3). These declarations do not need to verify their contracts, but instead are used when a procedure must be called. On the other hand, a new **let** declaration is introduced for each procedure body and implementation. These are never called, but are instead used to verify that the implementation fulfills the contracts.

Correctness. Renaming type arguments does not change the semantics, and is therefore allowed. In Boogie all calls, including recursive calls, use the modular semantics where contracts are substituted for the body of the callee. This separation of body and signature emulates that behavior.

4.5.2 Procedure Calls

A procedure call is replaced by a call to the generated corresponding abstract function in WhyML. This separates the caller and callee traces just like in Boogie, as discussed in [33]. In Boogie all calls, including recursive calls, use the modular semantics where contracts are substituted for the body of the callee. In WhyML, in contrast, recursive calls with the non-modular semantics where the actual body of the callee is used can be implemented using the keyword **rec**. The translation, however, does not use this feature of the WhyML language since it would alter the Boogie semantics.

[3] An alternative approach would have been to rename parameters within the contracts to match each new implementation. Both approaches are valid, as long as they are consistent.

```
...
   val __p (____xin: int ): int
      returns{|____result  → (( ____xin ) ≤ ( ____result )) }

   val _GLOBAL_LOCAL___p_____result (): int

   let __p_IMPL0 (____xin: int ) : int
      returns{|____result  → (( ____xin ) ≤ ( ____result )) }
   =(
      ...
      assume {(( ____xin ) ≤ ( ____result.contents )) };
      ...
      ____result.contents
   )

   let __p_IMPL1 (____xin: int ) : int
      returns{|____result  → (( ____xin ) ≤ ( ____result )) }
   =(
      ...
      assume {(( ____xin ) ≤ ( 4 )) };
      ____result .contents ←  5 ;
      ...
      ____result.contents
   )

   let __p_IMPL2 (____xin: int ) : int
      returns{|____result  → (( ____xin ) ≤ ( ____result )) }
   =(
      ...
      ____result .contents ←  ____xin ;
      ...
      ____result.contents
   )
...
```

Figure 4.14: Automatic translation of procedure and implementation declarations. The **val** declaration can be used by calls, the **let** declarations need to be verified by Why3.

```
procedure p()
free requires e1;
free ensures e2;
{
   //do something
}
```

(a) Boogie version.

```
val p ():  ()
   ensures { e2 }

let p_IMPL0 () :  ()
   requires { e1 }
=(
   (*do something*)
)
```

(b) Translated WhyML version.

Figure 4.15: Translation of free pre- and postconditions.

4.5.3 Procedure Contracts

When translating procedure contracts, one must pay attention to *free* pre- and postconditions. In contrast to standard contracts, these do not have to be checked, but may be assumed where appropriate. The method of translation is illustrated in figure 4.15.
Free preconditions are only attached to the **let** declarations, while free postconditions are only attached to the abstract functions. Non-free conditions are attached to both.
Correctness. As noted before, a procedure call in Boogie is replaced by a call to the translated abstract function in WhyML. In both cases, the caller must guarantee that the preconditions hold, and can in return assume the postconditions to be true. Free precondtions do not have to be verified by the caller, therefore they are not in the abstract function. However, free precondtions can be assumed by the callee, therefore they appear in the **let** declarations. On the other hand, free postconditions can be assumed to hold by the caller, therefore they can be found in the abstract function. Free postconditions do not have to be proven by the implementation (callee), therefore they must not appear in the **let** declarations.

4.5.4 Multiple outputs

Boogie has the option for procedures to return multiple values of potentially different types. To emulate this, the translation uses WhyML's tuple type.
Correctness. Tuples are used to pass values after a function call, after which the single values can be separated again. Contracts may also reference each value of a tuple individually. Therefore the behavior is equivalent to Boogie's.

4.6 Procedure and Implementation Bodies

Bodies of **procedure** and **implementation** declarations are translated using a fixed structure. It is presented in figure 4.16 and described in this section.

```
174   let implementation_body (inputArgs ) : outputType
175      (*...contracts...*)
176   =(
177      (*...local bindings...*)
178      (*...where clauses...*)
179      try(
180         (*...translated body...*)
181      )with
182         |Return  → assume { true }
183      end ;
184      (*return result*)
185   )
```

Figure 4.16: General structure of a procedure or implementation body translation.

4.6.1 Local Variables

WhyML does not have local variables as they are known from Boogie. Instead it has a construct called local binding. The translation of a local variable declaration `var i: T;` is of the form `let i = ref (global_i()) in...`, where `global_i` must be defined as a global abstract function `val global_i():T`. The return parameter(s) are handled as if they were local variables.
Correctness. First we assume no where-clause is present (see next section for the other case). When a local variable is declared in Boogie, it can have any value which matches its type. This same effect is achieved by our translation, because abstract functions (without postconditions) can return any value of the given type, regardless how many times the function has been called. Finally, using `ref` as a wrapper makes the contents of the binding mutable, just like a local variable in Boogie.

4.6.2 Where-clause Statements

The second part of the body consists of several **assume** statements, namely one for each of the following where-clauses:

- input parameters of the current procedure signature
- output parameters of the current procedure signature
- local variables of the current body
- global variables

Notably, where-clauses from output parameters of other procedures are not included, as they become part of the corresponding **val** declaration's contracts.
Correctness. Where-clauses come into effect at two occasions: When a variable or parameter is initialized, and when **havoc** is called on them. Each where-clause can restrict (or bound) the possible values a variable or parameter can attain. Since all of the above mentioned values are initialized at the

beginning of a procedure, the bounds of the where-clause can be enforced with assume statements. Assignments to variables overwrite where-clauses, which is also the case for assume statements[4].

4.6.3 Return Emulation

The next part of the structure is the translation of all the statements within the body (excluding variable declarations), encapsulated by an exception handling. This is illustrated in lines 179 through 182 of 4.16. Finally, the last line returns the value of the output parameter, which was earlier defined as a local binding.
Correctness. If the original body contains no return statement, the translated body executes all other statements, and finally returns the correct output parameter. Assuming all the intermediate statements are translated correctly, then the returned values are equivalent in the original Boogie program and translated WhyML program.
If the original body contains one (or more) return statement(s), the translated program will throw an exception at the equivalent positions (see section 4.9). Such an exception is caught by the surrounding exception handling clause, and immediately afterward, the current value of the output parameter is returned. Since WhyML does not roll back any changes when an exception is thrown, the returned value is equivalent to the value returned in the Boogie program (again assuming the other statements are translated correctly).

4.7 Loops

Boogie supports the usual while loops with invariants, which can be translated one to one into WhyML's while loop. Special care must only be taken when translating loops with the following properties:

- **Wildcard $*$.** If the exit condition is a wildcard expression, the translation replaces it by a call to `havoc_bool()`, which is introduced in the preamble, see section 4.19.
- **Free invariant.** A free invariant is translated into an equivalent invariant, and two assume statements containing the same formula. One of these statements is placed directly before the loop, the other is placed as the last statement of the loop body. Additionally, for each non-free invariant, we must generate two assert statements at the same locations. If there are multiple invariants, the generated assume and assert statements retain the same order as the declared invariants.

Figures 4.17 and 4.18 demonstrate how free invariants may be translated.
Since WhyML supports loop variants, invariant attributes could potentially be used to declare variants which are included in the translated programs. Section 6.3.1 discusses this further.

[4] I.e. `assume i = 0;i := 1;` does not lead to a contradiction, whereas `assume i = 0;assume i = 1;` is tantamount to `false`

```
while (*)
invariant inv0;
free invariant inv_free;
{
   s;
}
```

Figure 4.17: Boogie loop with a free invariant.

```
   assert {inv0};
   assume {inv_free};
   while havoc_bool() do
      invariant {inv0}
      invariant {inv_free}
      s;
      assert {inv0};
      assume {inv_free};
   done;

```

Figure 4.18: Translation of free invariants.

Correctness. The wildcard expression states that the loop may execute any number of times, non-deterministically. The usage of the abstract function `havoc_bool` has the same effect since it non-deterministically returns either true or false each time it is called.
A free invariant may be assumed to hold true when appropriate, but does not have to be checked. Adding our two assume statements lets the invariant easily be checked in the translated program. The assert statements of non-free invariants must be added because the order of invariants matter when they are checked: An invariant `inv0` declared before a free invariant `inv1` must be checked without assuming `inv1`. Note that assume statements are needed for the generated WhyML programs to be amenable to verification. The assert statements on the other hand are needed for soundness - without them, the translation of the program in 4.19 would be verifiable, but the original is not.

```
while (*)
invariant false;
free invariant false;
{
   //do something
}
```

Figure 4.19: Unsound translation example without assert statements..

4.8 If-then-else

If-then-else structures can be translated one to one to the equivalent structure in WhyML. Wildcard expressions in the condition can be translated by calling the `havoc_bool` abstract function.
Correctness. The structures in both languages behave the same. `havoc_bool` introduces the same non-determinism as the wildcard expression.

4.9 Break and return statements

Boogie's break and return statements can be simulated in WhyML using exceptions. To that extent, the two exceptions which are defined in the preamble are used.
While statements which contain a break in Boogie are translated by surrounding the resulting while statement with a try, followed by a clause which catches `Break` exceptions. As soon as a break would occur, we can instead throw such an exception.
Return statements are translated similarly: The entire body of a procedure is surrounded by an exception handler, which catches and handles `Return` exceptions. **Correctness.** If the original program contains one (or more) **`return`** or **`break`** statement(s), the translated program will throw an exception at the equivalent position(s). Such an exception is caught by the surrounding exception handling clause, and all variables retain their values. The exception handlers around while loops also don't catch `Return` exceptions, so throwing a `Return` exception inside a loop body is translated correctly. Since WhyML does not roll back any changes when an exception is thrown, the returned value is equivalent to the value returned in the Boogie program.

4.10 Havoc

The havoc statement in Boogie sets a variable `v` to an arbitrarily chosen value. To emulate this behavior, the preamble of the translation contains an abstract function `havoc(x:` α`):`α, which takes an argument of any type, and sets the result to the same type.
Correctness. By definition, the returned value of an abstract function with no post-condition is non-deterministic, which is equivalent to Boogie's havoc feature. Section 4.12 discusses the case when variables contain where-clauses.

4.11 Call–forall

Call–forall statements are described in detail in [33]. Basically, they can be used as locally visible lemmas, without the burden of proof. I.e. the proof is carried out by the callee at some other point, and the caller may assume that the lemma is true. Figures 4.20 and 4.21 illustrate an example of how call–forall statements are translated. Assume we have **`procedure`** `Lemma(`$x_1 : T_1, ..., x_n : T_n$`)` with n

```
208 procedure Lemma(x:X, y:Y);
209 requires P(x,y);
210 ensures Q(x,y);
211
212 [...]
213 var y0:Y
214 call ∀ Lemma(∗,y0);
215 [...]
```

Figure 4.20: Example call–forall statement.

```
216 val Lemma(x:X) (y:Y)
217 requires {P(x,y)}
218 ensures {Q(x,y)}
219
220 [...]
221 var y0:Y
222 assume {forall x:X. P(x,y0) → Q(x,y0)}
223 [...]
```

Figure 4.21: Translation of call–forall statement in figure 4.20.

arguments, where argument i is named x_i and is of type T_i, for $1 \leq i \leq n$. Additionally, let the procedure contain m preconditions $P_j, 1 \leq j \leq m$ and l postconditions $Q_k, 1 \leq k \leq l$. Each input argument may of course appear in any pre- and postcondition. A call-forall statement may either use concrete variables, or the wild-card expression for each input argument of `Lemma`. The translation produces a statement of the form:
assume{**forall** x_{h_0}:T_{h_0}, ..., x_{h_g}:T_{h_g}.($P_1 \wedge \ldots \wedge P_m$)→($Q_1 \wedge \ldots \wedge Q_l$) }

Each occurrence of a wildcard expression in the pre- and postconditions is substituted by the corresponding quantified variable of the correct type. Occurrences of concrete variables are not substituted.
Correctness. The translation directly applies the definition of call-forall statements, and is therefore correct.

4.12 Where-clauses

A variable which contains a where-clause introduces statements of the form **assume** e, where e is the expression taken form the where-clause, at the following positions in the translation:

- After the variable is initialized. This includes the beginning of each **let** declaration.

- After `havoc` is called on the variable.

Additionally, for in- and out-parameters of procedure declarations, `e` is added as a post-condition of the corresponding abstract function and in the beginning of the procedure's body in the `let` declaration, if such a body exists.
Correctness. Follows from the definition of where-clauses given in [33]. Where-clauses bound the possible ranges of variables and parameters. These bounds can also be enforced using assume statements. Therefore the assume statements must be added at the aforementioned positions.

4.13 Goto statements

Several approaches exist to translate goto structures into semantically equivalent while- and if-statements [38][10]. Although all these approaches are sound, the resulting code is not particularly amenable to verification. However, in practice gotos are mainly used for one of the following cases:

- Introduce non-determinism
- Encode loops with variants and invariants

Using an example of programs generated by AutoProof [28], we show how such structured gotos can be translated to retain verifiability:
Figure 4.22 shows a code snippet taken from a Boogie program generated by AutoProof. It shows how gotos can be used to introduce non-determinism. Since all jumps are performed forwards, the structure can easily be rewritten using only if-then-else statements with wildcard expressions as conditions(figure 4.23), and therefore retain verifiability.

Figure 4.24 schematically shows how AutoProof uses gotos to encode loops from Eiffel. Using the general translation mechanism for gotos from [38], we get the structure presented in figure 4.25. Note that the invariants at lines 276 and 277 must be added to enable verification. Using this translation mechanism allows us to translate any goto structure into semantically equivalent while-loops, and add invariants needed for verification automatically for known cases.

```
implementation SUM_AND_MAX.invariant_admissibility_check(Current: ref)
{
  entry:
  goto pre, a2, a3;
  pre:
  return;
  a2:
  assume user_inv(Heap, Current);
  assert admissibility2(Heap, Current); // type:A2
  return;
  a3:
  assume user_inv(Heap, Current);
  assert admissibility3(Heap, Current); // type:A3
  return;
}
```

Figure 4.22: Non-determinism by goto.

```
let SUM_AND_MAX.invariant_admissibility_check(Current: ref)
=(
   [...]
   if(havoc_bool()) then(
      raise Return
   )else(
      if(havoc_bool()) then(
         assume {user_inv heap current};
         assert {admissibility2 heap current};
         raise Return
      )else(
         assume {user_inv Heap Current};
         assert {admissibility3 Heap, Current};
         raise Return
      )
   )
   [...]
)
```

Figure 4.23: Translation of figure 4.22 which retains verifiability.

```
implementation SUM_AND_MAX.sum_and_max(Current: ref, a: ref)
    returns (Result: ref)
{
[...]
   //stmtList1;
   goto loop_head_1;
loop_head_1:
   assert invList;
   goto loop_body_2, loop_end_3;
Loop_body_2:
   assume ¬ u_cond;
   //stmtList2;
   assert variant_decrease ∧ variant_positive;
   goto Loop_head_1;
loop_end_3:
   assume u_cond;
[...]
}
```

Figure 4.24: Loop encoding using gotos.

4.14 Polymorphic Maps

Although WhyML also has polymorphic maps, a one to one translation does not work, as they are not as powerful as their Boogie counterparts. As an example, a global variable in Boogie of the form `var heap: <α>[ref,Field α]α;` is not directly translatable into WhyML. Instead, the translation of such a variable (or constant of such a type) can be performed in several steps:

1. Find all instantiations $I_0, I_1, ..., I_n$ of the type parameter α of the variable.
2. For each instantiation $I_i, 0 \leq i \leq n$ create a new variable as follows in the translated program: `val heap_i: ref (map (ref, field` I_i`)` I_i`)`. Additionally create a variable with a newly introduced dummy type I_D.
3. Whenever `heap` is used in the original program, replace it with the corresponding new variable `heap_i` in the translated program.
4. Axioms and functions which use a type parameter for α are translated into $n + 1$ new equivalent structures, one for each instantiated type of `heap`, and for the dummy type.

Correctness. We may view polymorphic maps as a set of maps. This set contains n maps, one for each instantiation. With this translation method we explicitly separate the maps of the set, while still axiomatizing them correctly. Finally, the dummy type covers the corner case where a variable is never instantiated.

```
274 let pc = ref 1 in
275 while not (¬pc = −1) do
276    invariant {invList} @@
277    invariant {pc = −1 -> u_cond} @@
278    if (¬pc = 1) then (
279       assert {invList};
280       let gotoChooser = someInt() in
281       assume {0≤ gotoChooser ∧  gotoChooser ≤ 1};
282       if gotoChooser = 0 then(
283          pc := 2;
284       ) else (
285          if gotoChooser = 1 then(
286             pc := 3;
287          ) else (
288             absurd;
289          );
290       );
291    ) ;
292    if ¬pc = 2 then (
293       assume {not u_cond};
294       stmtList2;
295       assert { variant_decrease ∧  variant_positive};
296       pc := 1;
297    );
298    if ¬pc = 3 then (
299       assume {u_cond};
300       pc := −1;
301    );
302 done;
```

Figure 4.25: Translation of figure 4.24.

4.15 Frame clause

A procedure's **modifies** clause contains a list of global variables which the current procedure may modify. The translator must add each variable from this list to a **writes** clause of the corresponding abstract function, but **not** to the **let** declaration.
Correctness. In Boogie, a global variable v_i which is assigned to in the body of a procedure p_1 **must** be added to the **modifies** clause, while other global variables **may** be added. When a procedure p_2 calls p_1, p_2 must assume that the value of v_i may have changed, with respect to p_1's postconditions. By adding the same variable to the **writes** clause of the corresponding abstract function p_1_val, we make sure that callers in the translated program also know that v_i might be modified. Note: We do not add it to the corresponding **let** declaration, because once the clause is added, Why3 enforces that any global variable which is read must be added to a **reads** clause. Since in our translation scheme **let** declarations are never called, this method is valid.

4.16 Triggers

Triggers can be translated one to one into WhyML triggers.
Correctness. Triggers do not influence correctness, only verifiability. Why3 passes triggers on to back-end provers which support them [12].

4.17 Axiom, Assume, Assert

These may all be translated directly into the corresponding WhyML structures.

4.18 Declaration order

Programs in both verification languages consist of several top level declarations such as axioms, procedures, functions etc. However a main difference of the two is that in Boogie, any declaration may be called or accessed at any point, whereas in WhyML only elements which were defined previously are visible. Therefore the automatic translation is required to reorder declarations and explicitly split up procedure contracts and bodies. The translated declarations are always reordered as follows:

1. **Type Declarations.** Type declarations which reference other type declarations must be inserted later. Circular type declarations are not allowed in Boogie.
2. **Global Variable Declarations.**
3. **Functions / Predicates.**

4. **Axioms.**
5. **Abstract Functions.** These are declared using the keyword `val`. They roughly correspond to procedure declarations without a body in Boogie.
6. `let` **declarations.** Each procedure body ist translated into one of these.
7. **Rest.**

The class `DeclarationReordering` is used to perform this second-to-last step of the translation on the WhyML AST, before the code is written to a file.
Correctness. Declarations which we add first cannot reference declarations which are added later. Therefore we ensure that anything which is referenced within the translated program is declared first. Note that it is important for the translation to axiomatize functions, as described in section 4.4. Otherwise two functions which reference each-other could not be declared, as each would have to be defined before the other.

4.19 Preamble

The preamble is predefined code which is introduced at the beginning of every translation, presented in figure 4.26. Lines 304 through 314 import useful theories from Why3's standard library [19]. `bool`, `int` and `real` are self-explanatory counterparts of Boogie's primitive types. `map` is used to emulate Boogie's (possibly polymorphic) map type denoted by `[]`. `ref` is used as a wrapper for mutable variables. Lines 316 and 317 introduce functions which offer the ability to emulate Boogie's havoc statement, described further in section 4.10. Boogies partial order operator $<:$ is introduced and axiomatized as such in lines 320 through 326. Next, lines 328 and 329 introduce two exceptions which are used in place of Boogies break and return statements. Further details on how exactly can be found in section 4.9. Finally the last two lines provide syntactic sugar to rename conversion functions from `int` to `real` and vice versa, to match the equivalent functions from Boogie [26].

```
303  (* Preamble Start *)
304  use import bool.Bool
305  use import map.Map
306  use import ref.Ref
307  use import int.Int
308  use import int.EuclideanDivision
309  use import real.RealInfix
310  (* int → real *)
311  use import real.FromInt
312  (* real → int *)
313  use import real.Truncate
314  use import real.PowerReal
315
316  val havoc (_x: α ): α
317  val havoc_bool (): bool
318
319  (* define and axiomatize Boogie's partial order operator. *)
320  predicate (<:) (_x: α ) (_y: α )
321  (* reflexive *)
322  axiom ReflexivePartialOrder: forall _a: α   . (( _a ) <: ( _a ))
323  (* transitive *)
324  axiom TransitivePartialOrder: forall _a _b _c: α   . ((((( _a
      ) <: ( _b )) )&&((( _b ) <: ( _c )) )) )→((( _a ) <: ( _c ))
      ))
325  (* antisymmetric *)
326  axiom AntiymmetricPartialorder: forall _a _b: α   . ((((( _a )
      <: ( _b )) )&&((( _b ) <: ( _a )) )) )→((( _a ) = ( _b )) ))
327
328  exception Return
329  exception Break
330
331  function real (_x: int ) : real  =
332     (from_int ( _x ) )
333  function int (_x: real ) : int  =
334     (floor ( _x ) )
335
336  (* Preamble End *)
```

Figure 4.26: Predefined code at the beginning of each translation.

Chapter 5

Translator API & CLI

5.1 CLI

5.1.1 Requirements

In order to execute the translation program, a Java runtime with version 8 or higher is required. Boogie2Why3 – the program to perform the translation – can be downloaded as a jar file from [1].

5.1.2 Documentation

Boogie2Why3 takes one mandatory input argument:

1. Input file name. Can be given as a path to a file, or a file name in the working directory. This file must contain legal Boogie code.

Afterwards, it takes the following optional arguments:

1. Output file name. Can be given as a path to a file, or a file name in the working directory. If no such file exists, it is created. If the specified file exists, the output is appended to it.
2. The parameter "-debug". In this case the program goes into debug mode and outputs information and warnings on the console. This parameter does not affect the generated WhyML translation.
3. The parameter "-h" to display the cli documentation on the console.

If no output file is specified, the translation is printed to the console. If there are no arguments at all, the cli documentation is displayed.

5.1.3 Examples

Quiet translation

To execute a translation, the command from figure 5.1 can be executed in a shell which has access to the executable jar file.

```
337 > java -jar Boogie2Why3.jar input.bpl output.mlw
```

Figure 5.1: CLI example in standard mode.

Since this is **not** executed in debug mode ("-debug"), no information is displayed on the console during the translation.

Debug translation

To execute a translation with information, the command must be adjusted as as seen in figure 5.2:

```
338 > java -jar Boogie2Why3.jar input.bpl output.mlw -debug
```

Figure 5.2: CLI example in debug mode.

Help

In order for the program to display a documentation the "-h" command can be used as seen in figure 5.3

```
339 > java -jar Boogie2Why3.jar -h
```

Figure 5.3: CLI example to display documentation.

5.2 API

5.2.1 Requirements

The source code of the project can be obtained from [1]. It requires Java SDK version 8 or higher. The only additionally needed library is our modified version of BoogieAmp, which is included in the project. The project can easily be

compiled and executed using the Java IDE Intellij [21]. The source code of our modified BoogieAmp library is also available at [1].

5.2.2 Documentation - Main translation

As seen in chapter 3, the translation operates in several consecutive steps. Each step can be accessed with specific API calls which are documented in this section.

Step 1 - Parser and AST generation

To parse a Boogie file and produce an AST we use BoogieAmp's API, namely the `ProgramFactory` class, as seen in figure 5.4. This gives access to an array of all the declarations defined in the Boogie file through the following command:
`pf.getASTRoot().getDeclarations();`

Steps 2 & 3 - Translate Boogie AST into WhyML AST

The code in 5.5 demonstrates how to execute the translation of the generated Boogie AST. The result is a WhyML module. Lines 352 and 353 create a new translator and execute all the necessary steps. This includes attaching the preamble, renaming variables where necessary and reordering the declarations. Line 355 creates a new WhyML module with the given name and comment, while the last line fills this module with the translated declarations.

Step 4 - Outputting WhyML code

The last step - writing code to a file based on the AST - is performed by the class `ModulePrinterVisitor`, as seen in listing 5.6. Using the visitor pattern, each node of the abstract syntax tree is accessed and corresponding code is generated. The code is stored as a list of strings, where each new element of the list represents a new line. Finally this list is traversed and written to a specified file (359).

```
340 //parse input
341 String boogieFileName = "input.bpl"
342 ProgramFactory pf;
343 try {
344     pf = new ProgramFactory(boogieFileName);
345 } catch (Exception e) {
346     Log.error(e);
347     return;
348 }
349 //access AST
350 pf.getASTRoot();
```

Figure 5.4: Parsing a Boogie file and creating the corresponding AST.

```
351 //execute translation
352 BoogieTranslator bt = new BoogieTranslator();
353 List<Declaration> translatedDeclarations = bt.translate(pf);
354 //fill new WhyML module with translation
355 Module translationModule = new Module("Translation", "This translation was
        automatically generated.");
356 translationModule.addDeclerations(translatedDeclarations);
```

Figure 5.5: Translating a Boogie AST into a semantically equivalent WhyML AST.

```
357 ModulePrinterVisitor mpv = new ModulePrinterVisitor(translationModule);
358 List<String> printableModule = mpv.printModuleToString();
359 ModulePrinterVisitor.printToFile(printableModule,"output.mlw");
```

Figure 5.6: Writing WhyML code to a file.

5.2.3 API Extensions

In this section we explain how a programmer can write extensions for our tool. The first step in writing an extension for the translator is to find out on what part the extension should operate:

1. Boogie AST
2. WhyML AST
3. Both

For the first two cases one can simply provide a new implementation for one of the interfaces `boogie.ast.ASTVisitor` and `whyMl.ASTVisitor`, respectively. Examples include the classes `ProcedureBodyFactory` and `ModulePrinterVisitor`. Once such a visitor exists, it can be added to the workflow where appropriate. For the third case, either a combination of the two visitors can be used, or the class `BoogieTranslator` can be modified. `BoogieTranslator` visits all nodes of the Boogie AST and generates a corresponding WhyML AST. A programmer therefore has access to the Boogie AST, and can influence the creation of the WhyML AST.

Chapter 6

Evaluation

This chapter evaluates the current state of our translator Boogie2Why3. The first section presents the different benchmark programs with which we tested the translator. Section 6.2 discusses which benchmarks produced a valid translation which we could verify. The last section 6.3 presents which benefits a programmer gains from the translator, and what limits the translator has in its current state.

6.1 Benchmark Programs

Table 6.1 lists all programs which we used as benchmarks for the translator. Column LOC contains the lines of code. The next three columns display the amount of procedures, functions and axioms in each Boogie program. The last column shows the amount of pre- and postconditions, invariants, and assert statements in the Boogie programs, i.e. all constructs which must be checked, added up in that order. The entries of the table are sorted by increasing LOC of the input programs.
Table 6.2 describes the translated benchmarks into WhyML. Columns "auto LOC" and "manual LOC" contain lines of code of the automatically and manually translated programs. For the automatically translated programs, the preamble always accounts for 35 lines. The last two columns describe how many functions and axioms are in the automatically translated programs. The preamble contains 2 functions and 3 axioms.
We collected the source codes of all benchmarks at [1]. Each program was either written by us for this thesis, or obtained from one of the following opensource repositories: [16][11]. We now briefly present each program and the most challenging aspects regarding their translations.

Program	LOC	#procedures	#functions	#axioms	#specification items
Linear Search	17	1	0	0	1+3+2+0 =6
Rotation By Copy	52	1	3	4	1+1+4+0 =6
Welfare Crook	60	1	3	7	0+2+2+0 = 4
Binary Search	66	3	1	5	3+7+2+0 =12
Integer & Real Ops	66	10	0	0	0+15+0+0 =15
Bubble Sort	87	2	3	4	1+4+10+0 =15
Binary Search Tree	212	4	9	25	11+8+18+0 =37
Misc	227	16	2	7	8+15+0+12 =35
Rotation By Reversal	228	10	5	7	21+18+12+9 =60
AutoProof	1843	18	272	442	45+42+0+23 =110

Table 6.1: Boogie Benchmark programs.

6.1.1 Linear Search

Iterates through an array represented as a map and searches for a value. Figure 6.1 shows the program.
Demonstrated features.

- `return` statement
- local variable

6.1.2 Rotation By Copy

Left rotates an array represented as a map by r positions. The elements of the array are copied to their new positions. The source code is presented in figure 6.2.
Demonstrated features.

- `return` statement
- axiomatized functions within postconditions which need to be verified

Program	auto LOC	manual LOC	#functions	#axioms
Linear Search	73	33	2	3
Rotation By Copy	95	82	5	7
Welfare Crook	152	NA	5	9
Binary Search	150	97	3	8
Integer & Real Ops	201	102	2	3
Bubble Sort	135	108	5	9
Binary Search Tree	313	NA	11	30
Misc	569	NA	4	11
Rotation By Reversal	353	270	7	10

Table 6.2: Translations of benchmark programs in table 6.1.

```
360 // 'arr' is an "array" of size 'size'. If it contains the element
      'val', its index is returned. Otherwise -1 is returned.
361 procedure LinearSearch(arr: [int]int, size: int, val:int) returns
      (index: int)
362     requires size ≥ 0;
363     ensures (index = -1 ∨ (arr[index] = val ∧ index ≥ 0 ∧ index <
          size));
364     ensures (∃ i:int • 0 ≤ i ∧ i < size ∧ arr[i] = val) ⟹
          (index ≥ 0 ∧ index < size ∧ arr[index] = val); //if the
          array contians val, then the correct index is returned.
365     ensures (¬(∃ i:int • 0 ≤ i ∧ i < size ∧ arr[i] = val) ⟹
          index = -1); //if the array does not contian val, then -1
          is returned.
366 {
367     index := 0;
368     while(index < size)
369        invariant index ≤ size;
370        invariant (∀ i:int • 0 ≤ i ∧ i < index ⟹ arr[i] ≠ val);
371     {
372        if (arr[index] = val){ return; }
373        index := index + 1;
374     }
375     index := -1;
376 }
```

Figure 6.1: Linear Search benchmark.

```
377 function seq(a: [int]int, low: int, high: int) returns([int]int);
378 axiom (∀ a: [int]int, low: int, high: int, i: int •
379       0 ≤ i ∧ i < high − low
380       ⟹
381       seq(a, low, high)[i] = a[low + i]);
382
383 // i mod N
384 function wrap(i: int, N: int) returns(int);
385 axiom (∀ i, N: int • 0 ≤ i ∧ i < N ⟹ wrap(i, N) = i);
386 axiom (∀ i, N: int • 0 < N ∧ N ≤ i ⟹ wrap(i, N) = wrap(i − N,
        N));
387
388 // Left−rotated sequence of a[low..high) at r  (directly defined
        using wrap)
389 function rot(a: [int]int, low: int, high: int, r: int)
        returns([int]int);
390 axiom (∀ a: [int]int, low: int, high: int, i, r: int •
391       0 ≤ r ∧ r < high − low ∧ 0 ≤ i ∧ i < high − low
392       ⟹
393       rot(a, low, high, r)[i] = seq(a, low, high)[wrap(i + r, high
              − low)]);
394
395 // Left−rotate a by r by copying
396 procedure rotate_copy(a: [int]int, N: int, r: int) returns(b:
        [int]int)
397   requires 0 ≤ r ∧ r < N;
398   ensures (∀ i: int • 0 ≤ i ∧ i < N ⟹ seq(b, 0, N)[i] = rot(a,
        0, N, r)[i]);
399 {
400    var s, d: int;
401    if (r = 0) {  // In this case, rotation coincide with identity
402       b := a;
403       return;
404    }
405    s := 0;
406    d := N − r;
407    while (s < N)
408    invariant (0 ≤ s ∧ s ≤ N);
409    invariant (d = wrap(s + N − r, N));
410    invariant (0 ≤ d ∧ d < N);
411    invariant (∀ i: int •
412              0 ≤ i ∧ i < s
413              ⟹
414              seq(a, 0, N)[i] = seq(b, 0, N)[wrap(i + N − r, N)]);
415    {
416       b[d] := a[s];
417       s := s + 1;
418       d := d + 1;
419       if (d = N) {
420          d := 0;
421       }
422    }
423 }
```

Figure 6.2: Rotation By Copy Benchmark.

```
424 procedure realPower() returns (res: real)
425 ensures res = 8.0; //NOT verifiable in boogie.
426 {
427    res := 2.0**3.0;
428 }
```

Figure 6.3: Power operator which is not verifiable by Boogie.

6.1.3 Welfare Crook

Annotated algorithm to find the smallest equal entry of three arrays.
Demonstrated features.

- Unique constants
- Multiple return parameters
- Type declaration

6.1.4 Binary Search

Verified implementation of the well known searching algorithm binary search.
Demonstrated features.

- User defined types
- Procedure calls

6.1.5 Integer & Real Ops

Contains operations on integers and reals. Boogie has a power operator on reals: $real ** real \rightarrow real$, but cannot verify calculations on such operators. The automatically translated program can be verified by some back-end provers automatically, see section 6.2. Figure 6.3 shows the procedure which contains the mentioned operator.
Demonstrated features.

- Operators on integers, such as `mod` and `div`
- Power operator on reals(**)
- Conversion functions and operators from `int` to `real` and vice-versa.

6.1.6 Bubble Sort

Verified implementation of the well known sorting algorithm bubble sort [42]. The most important parts are presented in figure 6.4.
Demonstrated features.

- User defined types
- Array assignment
- Procedure call
- Multiple loop invariants
- Free postcondition
- Functions with a body

6.1.7 Binary Search Tree

Representation of a binary search tree with the following operations implemented as procedures and proven correct: Searching for a node, Finding the node with minimum value in a subtree, and adding nodes to the tree. The most important code snippets are in figure 6.5.
Demonstrated features.

- User defined types
- Constants
- Global Variables
- Frame clause

```
429 ...
430 procedure swap (a: array T, i, j: int) returns(b: array T)
431    // elements in positions i,j are swapped
432    ensures ( b[i] = a[j]  ∧  b[j] = a[i] );
433    // all other elements are unchanged
434    ensures ( ∀ k: int • k ≠ i ∧ k ≠ j  ⟹  b[k] = a[k] );
435    // the output is a permutation of the input (not proved)
436    free ensures ( perm (a, b) );
437 {
438    var tmp: T;
439    b := a;
440    tmp := b[i];
441    b[i] := b[j];
442    b[j] := tmp;
443 }
444 procedure bubble_sort_improved (old_a: array T, n: int)
445    returns(a: array T)
446    requires n ≥ 1;
447    ensures perm (a, old_a);
448    ensures sorted (a, 1, n);
449 {
450    var i, j: int;
451    a := old_a;
452    i := n;
453    while ( i ≠ 1 )
454    invariant ( 1 ≤ i ∧ i ≤ n );
455    invariant ( perm (a, old_a) );
456    invariant ( sorted (a, i, n) );
457    invariant ( i < n ⟹ less_equal_pivot (a[i + 1], a, 1, i) );
458    {
459       j := 1;
460       while ( j ≠ i )
461       invariant ( 1 ≤ i ∧ i ≤ n );
462       invariant ( 1 ≤ j ∧ j ≤ i );
463       invariant ( perm (a, old_a) );
464       invariant ( sorted (a, i, n) );
465       invariant ( i < n ⟹ less_equal_pivot (a[i + 1], a, 1, i)
              );
466       invariant ( less_equal_pivot (a[j], a, 1, j) );
467       {
468          if ( ¬(a[j] <: a[j + 1]) ) {
469             call a := swap (a, j, j + 1);
470          }
471          j := j + 1;
472       }
473    }
474 }
```

Figure 6.4: Most important parts of Bubble Sort benchmark.

```
// Node type, Value (key) type, Node --> Node, Node --> Value,
    children, parent, values, root
type ref; type G = int; type LINK = [ref] ref; type VAL = [ref] G;
const Void: ref;
var left,right,parent: LINK; var value: VAL; var root: ref;
[...]
// binary tree invariant
function is_tree(l: LINK, r: LINK, p: LINK, v: VAL) returns(bool);
axiom (∀ l, r, p: LINK, v: VAL, n: ref • is_tree(l, r, p, v) ∧ n ≠
    Void ∧ p[n] ≠ Void ⟹ n = r[p[n]] ∨ n = l[p[n]] );
[...]
// Add 'node' to the tree
procedure put_bst (node: ref) returns (Result: ref)
   requires is_bst(left, right, parent, value);
   [...]
   modifies left, right, parent, root;
   ensures parent[root] = Void;
   ensures in(left, right, parent, value, root, node);
   ensures is_bst(left, right, parent, value);
{
   var x, y: ref;
   y := Void;
   x := root;
   while (x ≠ Void)
      invariant is_bst(left, right, parent, value);
      invariant in(left, right, parent, value, root, x);
      invariant y ≠ Void ⟹ left[y] = x ∨ right[y] = x;
      invariant y ≠ Void ⟹ in(left, right, parent, value, root,
          y);
      invariant root ≠ Void ∧ x = Void ⟹ y ≠ Void;
      invariant x = Void ⟹ inode(left, right, parent, value,
          root, value[node]) = y;
      invariant x ≠ Void ⟹ inode(left, right, parent, value,
          root, value[node]) = inode(left, right, parent, value,
          x, value[node]);
   {
      y := x;
      if (value[node] < value[x]) {x := left[x];}
      else {x := right[x];}
   }
   parent[node] := y;
   if (y = Void) {root := node;}
   else {
      if (value[node] < value[y]) {left[y] := node;}
      else {right[y] := node;}
   }
}
[...]
```

Figure 6.5: Important snippets from the Binary Search Tree Snippet.

6.1.8 Miscellaneous

This benchmark is a collection of 10 programs. It tests the correctness of our implementation for different constructs and corner cases. These programs are not meant to be particularly difficult to verify, but to make use of features of Boogie.
Demonstrated features.

- Arithmetic and boolean operations
- Map operations
- Assignments
- Recursive function definitions
- Global variable declarations
- Declaration reordering
- Where-clauses for variables, and input and output parameters
- Procedures with multiple implementations
- Frame clause
- Shadowed variables
- Type declarations

6.1.9 Rotation By Reversal

Rotation of an array by r positions by reversing sequences. [15] discusses the algorithm in detail. The most important part of the code is presented in figure 6.6.
Demonstrated features.

- `call` $\forall$

6.1.10 AutoProof

This benchmark program was generated by AutoProof [28]. It encodes an Eiffel program which calculates the sum of all elements of an array, and also finds the maximum value of the array. The Eiffel program and generated Boogie program can be found at [1]. The current version of our tool cannot translate this Boogie program, because the support for polymorphic maps hasn't been implemented yet. AutoProof uses polymorphic maps to encode the heap.

```
517 [...]
518 // Left-rotate a by r by performing three reversals.
519 // Key correctness argument: if |X| = r and |Y| = N - r,
520 //                           then rot(X Y, r) = rev(rev(X) rev(Y))
        = Y X
521 procedure rotate_reverse(a: [int]int, N: int, r: int) returns(b:
        [int]int)
522   requires 0 ≤ r ∧ r < N;
523   ensures (∀ i: int • 0 ≤ i ∧ i < N ⟹ seq(b, 0, N)[i] = rot(a,
          0, N, r)[i]);
524 {
525     b := a;
526     call b := reverse(b, 0, r);
527     call b := reverse(b, r, N);
528     assert (∀ i: int • 0 ≤ i ∧ i < r ⟹ seq(b, 0, r)[i] = rev(a,
          0, r)[i]);
529     assert (∀ i: int • 0 ≤ i ∧ i < N - r ⟹ seq(b, r, N)[i] =
          rev(a, r, N)[i]);
530     call ∀ lemma_rev_cat_2(a, 0, r, a, r, N, b, *);
531     assert (∀ i: int • 0 ≤ i ∧ i < r ⟹ rev(b, 0, r)[i] = seq(a,
          0, r)[i]);
532     assert (∀ i: int • 0 ≤ i ∧ i < N - r ⟹ rev(b, r, N)[i] =
          seq(a, r, N)[i]);
533     call b := reverse(b, 0, N);
534     assert (∀ i: int • 0 ≤ i ∧ i < N - r ⟹ seq(b, 0, N)[i] =
          seq(a, r, N)[i]);
535     assert (∀ i: int • 0 ≤ i ∧ i < r ⟹ seq(b, 0, N)[i + N - r] =
           seq(a, 0, r)[i]);
536     //call ∀ lemma_rot(a, 0, N, r, *);
537     //replaces call-∀ statement
538     assume (
539          ∀ p:int• (
540              (0 ≤ N) ∧ (0 ≤ r ∧ r < N - 0) ∧ (0 ≤ p ∧ p < N - 0)
541              ⟹
542              rot(a, 0, N, r)[p] = seq(a, 0, N)[wrap(r + p, N - 0)]
543          )
544     );
545     assert (∀ i: int • 0 ≤ i ∧ i < N - r ⟹ rot(a, 0, N, r)[i] =
          seq(b, 0, N)[i]);
546 }
```

Figure 6.6: Rotation By Reversal Benchmark.

6.2 Verifiability

This sections discusses how amenable the automatically translated programs are to verification. Table 6.3 summarizes the results of our evaluation. The first column states how long Boogie needed to prove a program correct. The next four columns refer to the verification time of the automatically translated programs in Why3, using different back-end provers. All times are in seconds, rounded to two decimal places. To time Boogie, we used a script which queries the current time before boogie starts and after boogie terminates. Why3 offers a timing service integrated within the IDE. However, since that measures system time, and our Boogie script measures clock time, we also measure clock time for Why3 using the unix `time` command. Cells containing only a number indicate the running time of a successful back-end prover. Cells with an f followed by two numbers of the form x/y indicate a failed attempt[1], where x out of y generated goals could not be proven. Finally, the last column states whether Why3 could successfully prove the program, using either a single or a combination of back-ends.
For our evaluation we used Boogie version 2.2.30705.1126 with Z3 version 4.3.2, running on a 64 bit Windows 8.1 notebook with Intel i7-5500U CPU@2.4GHz and 12GB ram. All of Boogie's settings were set to default. We executed Why3 version 0.86.1 on the same machine, inside a VirtualBox VM [29] emulating Ubuntu 15.04. Settings were set to 10 second time-limit and 2000MB available memory per goal.
Some interesting notes on the verification process:

- By default, Boogie infers some invariants [25]. This is the case for our "Linear Search" program, where Boogie automatically infers the invariant `i` $\geq$ `0` and is able to verify the program. For the translated program, we needed to add the invariant manually for the provers to succeed.
- The translation of the binary search tree could be proven by Alt-Ergo alone. The stated time of 12.08s does not violate the time-limit, because Why3 generates three goals for the program. Also, while neither Z3 nor CVC4 could prove the program to be correct, together they can handle the task: each is able to prove the missing goal of the other.
- From the misc benchmark, the failed goal of CVC3 and Alt-Ergo is `(fib 8)=21`, where fib is the axiomatized Fibonacci function.
- CVC3 exited with failure messages in several instances. The other provers simply ran into a timelimit for the cases where they didn't succeed.
- Rotation by reversal can be proven by Boogie version 4.1 with the included call-forall statements. Since newer versions of Boogie have dropped support for call-forall, we manually replaced all occurences by an equivalent assume statement according to the definition of call-forall [33]. However, after replacing these statements, Boogie is not able to verify the program anymore. This might explain why the translated program is not verifiable either.

[1] either by timeout or exit code

Program	Boogie Z3	Why Z3	Why CVC3	Why CVC4	Why Alt-Ergo	Why combo
Linear Search	0.85	0.26	0.24	0.37	0.26	ok
Rotation By Copy	0.91	0.90	*f:*1/1	*f:*1/1	9.99	ok
Welfare Crook	0.88	0.42	*f:*1/2	*f:*1/2	7.48	ok
Binary Search	0.85	0.44	*f:*1/3	0.56	*f:*1/3	ok
Integer & Real Op	*f:*1/10	7.68	*f:*6/10	*f:*7/10	9.37	ok
Bubble Sort	0.91	0.38	*f:*1/2	*f:*1/2	2.58	ok
Binary Search Tree	0.90	*f:*1/4	*f:*2/4	*f:*1/4	12.08	ok
Misc	0.93	1.19	*f:*1/18	13.23	*f:*1/18	ok
Rotation By Reversal	NA	*f:*2/10	*f:*9/10	*f:*5/10	*f:*6/10	*f:*1/10

Table 6.3: Verification times of automatically translated benchmark programs.

- Boogie defines a power operator on reals, but cannot verify any procedures which contain such operations. The translated program can be verified by back-ends such as Z3 and Alt-ergo. While Z3 takes relatively long for the task (4.68 seconds), Alt-ergo can prove it almost instantaneously (0.34 seconds). In return, Alt-ergo takes longer to verify conversion functions between int and real.

6.3 Benefits and limits of our translator

In this section we present how a programmer might benefit from our tool, and in which cases the tool reaches its limits.

6.3.1 Use Cases

Writing tests.

The Why3 system has the ability to execute programs natively. When a program cannot be verified, it might indicate a faulty implementation. Being able to execute programs on some test cases might help find the mistake. Figure 6.7 shows an example test case for the Linear Search program. It can be executed with the command `why3 execute Linear_search.mlw LinearSearch.testcase` Although not native to Boogie, a tool which can execute Boogie programs is introduced in [32].

Back-end provers.

As mentioned before, Why3 can generate verification conditions for an extensive list of back-end provers, including interactive ones. As seen in section 6.2, these provers can be used together to discharge different proof obligations of a

```
547  let testcase () =
548        let n = 5 in
549     let _val = 9 in
550     let a = const 0 in
551     let a = set a 0 3 in
552     let a = set a 1 7 in
553     let a = set a 2 1 in
554     let a = set a 3 9 in
555     let a = set a 4 11 in
556        linear_search a n _val
```

Figure 6.7: Executable test case for the linear search benchmark.

program. When SMT solvers fail, an experienced user may use an interactive prover for the remaining goals.

Triggers

Related to back-end provers are triggers. Why3 passes triggers on to back-end SMT solvers when possible [12]. Different solvers might show better performance with different triggers. Our tool is able to translate triggers automatically.

Prove termination.

Why3 issues a warning when a loop may diverge. In such a case a user can add a **`variant`** to the loop to prove its termination. The importance of this can be illustrated using the bubble sort benchmark from section 6.1.6: The local variable `i` of the last procedure is never decreased within the loop, therefore the loop never terminates (for $n > 1$). Nevertheless, Boogie can discharge all proof obligations and state that all postconditions are valid. Why3 does the same, but warns us that the loop may diverge. Once we add the line `i:=i-1` as the last statement of the outermost loop to the Boogie program, we are also able to add a variant to the translation and prove its termination.

Call forall.

In its latest version, Boogie has dropped support for call forall statements. Our tool is able to replace these statements by semantically equivalent **`assume`** statements. Using our API, a Boogie program with these statements replaced can also be generated and printed, using the command `pf.toFile()`.

Real operations.

The inbuilt type **`real`** has been added recently to the Boogie language [26]. However, verification of certain operators such as the power operator (**) is not

```
557 procedure bitVector()
558 {
559     assert 3bv3 = (13bv6 ++ 4bv3)[5:2];
560 }
```

Figure 6.8: Example usage of bit-vectors in Boogie.

possible yet. The automatically translated program can handle the verification, as seen in section 6.2.

6.3.2 Limitations of the implementation

Bit-vectors

Support for bit-vectors has not been implemented yet. The newest version of Why3 contains a theory for bit-vectors, although it doesn't support conversion from integers to bit-vectors yet, and therefore cannot yet handle all of Boogies operations. An example program performing concatenation and extraction operations on bit-vectors is presented in figure 6.8.

Attributes

Attributes have no fixed meaning in Boogie. In the current implementation, all attributes are ignored. Support for specific commands can easily be added to the translator, e.g. using a new visitor. An example usage might be to add support for loop variants, as we discuss in section 7.2.

Goto

Goto structures are not supported in the current version of the tool. Translating generally nested goto structures into semantically equivalent structures available in WhyML while maintaining provability is a difficult task. Several translation algorithms exist which preserve the semantics, such as the one presented in [38]. For specific applications, of which the goto structures are known beforehand, the translation can be adapted to maintain provability, see section 4.13. Figure 6.9 presents an example usage of **goto**s to introduce uncertainty into a program.

Polymorphic maps

Boogie supports polymorphic maps of the form `var Heap: <alpha>[Ref , Field alpha] alpha;` [34] [35]. WhyML also has a kind of polymorphic map, however it is more limited, as e.g. global variables cannot be of such a type. A one-to-one

translation is therefore not possible. Although support isn't implemented yet, we show how such maps may be translated in section 4.14.

```
561 procedure Goto()
562 {
563   //chose a path non-deterministially
564   goto label1, label2;
565   label1:
566   //do something
567   return;
568   label2:
569   //do something else
570   return;
571 }
```

Figure 6.9: Example usage of gotos in Boogie.

Chapter 7

Conclusions

7.1 Conclusions

With this thesis we designed and implemented an automatic translation mechanism between two intermediate verification languages, namely from Boogie to WhyML. Our tool can handle most of Boogie's constructs to produce semantically equivalent WhyML programs. For the constructs which are not implemented yet, we stated how the translation mechanism can be extended. Using different benchmark programs, we showed that the generated programs retain their provability. Finally, we demonstrated several scenarios in which our tool is beneficial to a programmer.

7.2 Future Work

The most obvious first step in extending the tool is implementing support for the missing constructs, namely `goto` and polymorphic maps. Once that is added, the tool can be tested with output from verification systems that use Boogie as an IVL, such as AutoProof. Such systems may then additionally use Why3 as an IVL, without having to modify their output.

To further improve provability of programs, we could perhaps make use of WhyML's functional programming style. The current translation method keeps the general structure of programs intact: (Boogie) loops are translated into (WhyML) loops for example. This new method would replace imperative constructs by functional ones - a loop with invariants would be replaced by a recursive `let` declaration with pre- and postconditions. Figure 7.1 demonstrates how we manually translated the loop from the linear search program using a functional programming style. It would be interesting to evaluate whether such a translation produces better results regarding provability.

Furthermore, variants could be added as attributes to Boogie programs, e.g.

as follows: `invariant {variant:n-i} true;`. This would allow the translator to automatically add them to the generated WhyML programs, and subsequently prove termination of Boogie programs.

```
572   predicate invariant1 (tuple: (int, (map int int), int, int))
573   =
574      let (_index, _arr, _size, _val) = tuple in
575      (0≤_index)
576   predicate invariant2 (tuple: (int, (map int int), int, int))
577   =
578      let (_index, _arr, _size, _val) = tuple in
579      (_index ≤ _size)
580   predicate invariant3 (tuple: (int, (map int int), int, int))
581   =
582      let (_index, _arr, _size, _val) = tuple in
583      (forall i:int. 0 ≤ i && i < _index → _arr[i] ≠ _val)
584
585   let rec whileLoop (_index: int) (_arr: map int int) (_size:
       int) (_val: int)
586   : ((int), (map int int), (int), (int))
587   requires {invariant1 (_index, _arr, _size, _val)}
588   requires {invariant2 (_index, _arr, _size, _val)}
589   requires {invariant3 (_index, _arr, _size, _val)}
590   ensures {invariant1 result}
591   ensures {invariant2 result}
592   ensures {invariant3 result}
593   ensures {
594      (exists i:int. 0 ≤ i && i < _size && _arr[i] = _val) →
       false}
595   ensures {
596      let (_indexB, _arrB, _sizeB, _valB) = result in
597      _arr = _arrB && _val = _valB && _size = _sizeB
598      }
599   raises {Return i → _arr[i] = _val}
600   raises {Return i → i ≥ 0}
601   raises {Return i → i < _size}
602   variant {_size - _index - 1}
603   =
604      if (_index < _size) then (*loop condition*)
605      (
606         (
607         if (_arr[_index] = _val) then
608            raise (Return _index);
609         );
610
611         let _index2 = _index + 1 in
612         let (res_index, res_arr, res_size, res_val) = (whileLoop
       _index2 _arr _size _val) in
613         (res_index, res_arr, res_size, res_val)
614
615      )
616      else
617         (_index, _arr, _size, _val)
```

Figure 7.1: Manual translation of the linear search benchmark using a functional programming style.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Supporting multiple proof engines by translating between intermediate verification languages

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Ameri	Michael Salar

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 01.09.2015	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

List of Figures

Bibliography

[1] Michael Ameri. *Thesis repository*, accessed August 1, 2015. https://bitbucket.org/michael_ameri/thesis.

[2] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[3] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[4] François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The alt-ergo automated theorem prover, 2008, 2013.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. The why3 platform. *LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition*, 2011.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[7] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. Hol-boogie - an interactive prover-backend for the verifying c compiler. *Journal of Automated Reasoning*, 44(1-2):111–144, 2010.

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 229–240. IEEE, 1994.

[11] ETH Zurich, Chair of Software Engineering. *Verified Boogie Programs*, accessed August 1, 2015. https://bitbucket.org/sechairethz.

[12] Jean-Christophe Filliâtre. *Triggers in Why3*, accessed August 1, 2015. http://lists.gforge.inria.fr/pipermail/why3-club/2012-February/000191.html.

[13] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Computer Aided Verification*, pages 173–177. Springer, 2007.

[14] Jean-Christophe Filliâtre and Andrei Paskevich. Why3-where programs meet provers. In *Programming Languages and Systems*, pages 125–128. Springer, 2013.

[15] Carlo A Furia. Rotation of sequences: Algorithms and proofs. *arXiv preprint arXiv:1406.5453*, 2014.

[16] Carlo A. Furia. *Verified Boogie Programs*, accessed August 1, 2015. https://bitbucket.org/caf/verified/.

[17] Inria Saclay-Ile-de-France / LRI Univ Paris-Sud 11 / CNRS. *Why3*, accessed August 1, 2015. http://why3.lri.fr/.

[18] Inria Saclay-Ile-de-France / LRI Univ Paris-Sud 11 / CNRS. *Why3 Provers*, accessed August 1, 2015. http://why3.lri.fr/#provers.

[19] Inria Saclay-Ile-de-France / LRI Univ Paris-Sud 11 / CNRS. *Why3 Standard Library*, accessed August 1, 2015. http://why3.lri.fr/stdlib-0.86.1/.

[20] F Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *Computer Aided Verification*, pages 301–306. Springer, 2005.

[21] JetBrains. *IntelliJ IDEA*, accessed August 1, 2015. https://www.jetbrains.com/idea/.

[22] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[23] K Rustan M Leino. Program proving using intermediate verification languages (ivls) like boogie and why3. *ACM SIGAda Ada Letters*, 32(3):25–26, 2012.

[24] K Rustan M Leino and Philipp RÃ$\frac{1}{4}$mmer. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer, 2010.

[25] Rustan Leino. *Boogie*, accessed August 1, 2015. https://github.com/boogie-org/boogie.

[26] Rustan Leino. *Boogie now has real, and now interprets integer div/mod*, accessed August 1, 2015. http://boogie.codeplex.com/discussions/397357.

[27] Bertrand Meyer and Karine Arnout. Componentization: the visitor example. *Computer*, (7):23–30, 2006.

[28] Martin Nordio, Carlo A. Furia, and Bertrand Meyer. *AutoProof*, accessed August 1, 2015. http://se.inf.ethz.ch/research/autoproof/.

[29] Oracle. *VirtualBox*, accessed August 1, 2015. https://www.virtualbox.org/.

[30] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated Deduction-CADE-11*, pages 748–752. Springer, 1992.

[31] Nadia Polikarpova. *Boogaloo*, accessed August 1, 2015. https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/Home.

[32] Nadia Polikarpova, Carlo A Furia, and Scott West. To run what no one has run before: Executing an intermediate verification language. In *Runtime Verification*, pages 251–268. Springer, 2013.

[33] K. Rustan and M. Leino. This is boogie 2. Technical report, Microsoft Research, Redmond, WA, USA, 2008.

[34] K. Rustan, M. Leino, and Philipp RÃ¼mmer. The boogie 2 type system: Design and verification condition generation. Technical report, Microsoft Research, Redmond, WA, USA, 2008.

[35] K. Rustan, M. Leino, and Philipp RÃ¼mmer. A polymorphic intermediate verification language:design and logical encoding. Technical report, Microsoft Research, Redmond, WA, USA, 2008.

[36] Martin Schäf. *BoogieAmp*, accessed August 1, 2015. https://github.com/martinschaef/boogieamp.

[37] Norbert Schirmer et al. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, 2006.

[38] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. C to O-O translation: Beyond the easy stuff. In Rocco Oliveto, Denys Poshyvanyk, James Cordy, and Thomas Dean, editors, *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, pages 19–28. IEEE Computer Society, October 2012.

[39] Julian Tschannen, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Verifying eiffel programs with boogie. *arXiv preprint arXiv:1106.4700*, 2011.

[40] Julian Tschannen, Carlo Alberto Furia, Martin Nordio, and Bertrand Meyer. Automatic verification of advanced object-oriented features: The autoproof approach. In *Tools for Practical Software Verification*, pages 133–155. Springer, 2012.

[41] Julian Tschannen and Bertrand Meyer. *Automatic verification of Eiffel programs*. PhD thesis, ETH, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Chair of Sotware Engineering, 2009.

[42] Wikipedia. *Bubble Sort*, accessed August 1, 2015. https://en.wikipedia.org/wiki/Bubble_sort.