Semester Thesis

Design of a *Java/Jml* Frontend for *BoogiePL*

Samuel Burri
saem [at] student.ethz.ch

supervised by:

Prof. Peter Müller
peter.mueller [at] inf.ethz.ch

October 2005

Abstract

This report describes a translation of a subset of the language *Jml*, extend with a few additional constructs, to the intermediate language *BoogiePL* for program analysis and program verification. The suggested translation has been implemented and may be used to reason about specifications of object-oriented software components.

Contents

1 Introduction and Idea

The Java Modeling Language (JML) [3] allows one to formally specify the behavior and interfaces of *Java* classes and methods. Proving correctness of a given class or method according to its specification is somewhat difficult since the the object-oriented structure of such components has very rich semantics.

However, we follow the approach to translate a given set of *JML* classes to *BoogiePL* [1] which is an intermediate language for program analysis and program verification. *BoogiePL* programs can be verified using the static verifier *Boogie* which is part of the *Microsoft Spec#* [7, 8] programming framework and may (at this time) be downloaded for free.

Nevertheless statically proving correctness or an arbitrarily property of a software component is still a semi-decidable problem. Consequently we won't be able to proof the absolute absence of errors but we might be able to find some or state that there are no errors with a high probability of success.

1.1 Outline of Semester Thesis

First, we decided on a subset of *Jml* that we will support in our translation throughout this thesis. All the supported constructs can be found in section 2 on page 5. The most important language constructs are taken into account. Anyhow, some concepts are missing. The most important ones are listed below:

Binary Classes	Only classes that appear in regular source files (`*.java` or `*.jml`) are supported. For *Java SDK* classes (e.g. `String`) one can use the annotated *Jml* classes provided by the *Jml* framework in the `specs` folder.
Interfaces	Only classes are supported.
Overloading	We don't consider overloading. Methods can be renamed to support the same functionality.
Shadowing	All fields of a class (including inherited fields from superclasses) should have different names.
Access Modifiers	We don't consider visibility and access control of classes, methods and fiels. i.e. packages, and modifiers such as `public` or `private` are not translated.
Exceptions	Exceptions are not supported either.
For-Loops	The only loop we consider is the annotated While-Loop. But all For-Loops can be rewritten as as While-Loops.
Case-Statements	Only If-Then-Else statements are supported. One may model a case statement using several If-Then-Else statements.
Full Jml Support	Pre- and postconditions, modifies clauses, invariants, assume and assert statements and loop invariants are considered. However, model fields and other more complex specification statements are not supported.

Additionally we introduced a couple of constructs which are currently not part of the *Jml* language. Following one finds the most interesting ones:

Pack and Unpack Statement	A way to kind of unwrap an object instance, moving its `inv` flag one step up respectively down the class hierarchy and asserting all invariants.
Ownership	When creating a new object or array instance one may specify an owner of this new reference.
Ownership-Expressions	Using this expressions one can access the specified owner object or owner type of a given reference.
One-Expression	This expression helps to reference the type of an object from the callers and the callees perspective.

Second, we formulated a translation in a pattern-matching fashion which fits the structure of abstract syntax trees provided by the *Jml* compiler. One can find all translation rules in section 3 on page 11. The modeling of the heap, the used functions and quiet a few translation rules are inspired by the translation suggested in *BoogieOOL: An object-oriented language for program verification* [2].

We start out with the definition of some functions, predicates and axioms and introduce a couple of global variables in order to model the heap and arrays. Then we go trough every class in the provided set of compilation units and translate its methods and fields and state some axioms to model inheritance and object access. Class methods are being translated to *BoogiePL* procedures where we can use the procedure specifications provided by *BoogiePL*. We translate only one constructor per class which also becomes a procedure in the *BoogiePL* code and will be called when a new object instance is created.

Third, we implemented the suggested translation in *Java*. We extended the existing *MultiJava* [5] and *Jml* [4] framework and called our tool *JtoBPL* (packages `org.jtobpl.*`). Details about our implementation can be found in section 4 on page 37. Basically we extended the *Jml* compiler using the *ANTLR* tool [9] in order to integrate our new language constructs in the supported language. In a second step we inspect the abstract syntax tree we get from the parser and make sure that only constructs that we support have been parsed. In the end the given *Jml* compilation units are translated to *BoogiePL* according to our translation rules.

Our source code and Jar files can be downloaded from `http://n.ethz.ch/student/burrisa/jtobpl/`.

2 Subset of supported *JML* Grammar

As we have pointed out in section 1.1, *Outline of Semester Thesis* on page 3, only a subset of the *Jml* language is considered in our translation. We attempted to find a reasonable subset, not to small and imposing to many restrictions and not to big so that our translation has still a feasible size. We tried to stick to the grammar rules documented in the *JML Reference Manual* [3] in *Appendix A, Grammar Summary*. However, you may note that we had to make some changes to the grammar in order to make the structure easier to translate. More details will be mentioned throughout the grammar description.

Although this section deals only with the theoretical definition of the extended *Jml* subset, we also include comments and restrictions that are rather concerned with the implementation of the translation.

2.1 Notations

The common *EBNF* formalism is employed.

Prod	Exactly one occurrence of production *Prod*.
(*Prod*)*	Zero or arbitrarily many repetitions of production *Prod*.
(*Prod*)$^+$	Arbitrarily many repetitions of production *Prod* - at least one.
[*Prod*]	Production *Prod* might occur or not.
Prod1 | *Prod2*	Either production *Prod1* or production *Prod2* occurs.
`term`	Terminal.
Class	Name of *Java* Class that represents one ore more productions in the syntax tree. This is very helpful if you try to follow the implementation in the visitor classes. If a class name starts with a J it's usually a *MultiJava* class, if it with Jml it's normally a *Jml* class and classes starting with B indicate that this class was introduced in *JtoBPL*.

2.2 Types, Programs and Classes

The usual naming convention is applyed.

Ident	::=	*Letter* (*Letter*| *Digit*)*	String, JLocalVariableExpression

The only basic numeric type we support is `int`.

Type	::=	`void`	CVoidType
	|	`boolean`	CBooleanType
	|	`int`	CNumericType
	|	*ReferenceType*	CClassNameType

Only class types that appear in a source file (`*.java` or `*.jml`) can be used. `Object` is the only exception of a built in class type that is supported. However, one may want to use the special fields `inv` or `commit` which are usually not defined in `Object`. In this case one can use the provided `Object.jml` file in the folder `specs/java/lang` and compile it together with the individual *Java* and/or *Jml* files. Arrays are only allowed to have one dimension.

ReferenceType	::=	`Object`	
	|	*Ident*	
	|	*Type* `[]`	CArrayType

A compilation unit - basically a `*.java` or `*.jml` file - is a set of class definitions. Note that neither interfaces nor package declarations nor import declarations are accepted.

CompilationUnit ::= (*ClassDefinition*)* JmlCompilationUnit

Note that no access modifiers such as `public` or `private` are considered in the class definition. As usual a class can be subclass of another class (again: every class, except `Object`, has to appear in the source code) or implicitly extend `Object`.

ClassDefinition ::= `class` *Ident* [`extends` *Ident*] `{` (*Member*)* `}` BClassDeclaration `<:` JmlClassDeclaration

A class member is either a method declaration, a field declaration or an invariant. None of them is allowed to have access modifiers.

Member ::= *MethodDeclaration*
| *FieldDeclaration*
| `/* invariant` *Expression* `; */` JmlInvariant

Shadowing is not supported: The name of a field has to be unique, i.e. no field is allowed to have the same name as another field in a superclass.

FieldDeclaration ::= *Type Ident* [`=` *Expression*] `;` JmlFieldDeclaration

2.3 Method Declarations and Specifications

A method declaration may either be a definition of a regular method or a definition of a constructor. Only one constructor per class will be allowed. Overloading is not supported but this limitation can be overcome using different method names. Overriding is allowed.

MethodDeclaration ::= *MethodSpecification Type Ident Formals CompoundStatement* BMethodDeclaration `<:` JmlMethodDeclaration
| *MethodSpecification Ident Formals ConstructorBlock* BConstructorDeclaration `<:` JmlConstructorrDeclaration

Formals ::= `(` [*ParameterDeclaration* (`,` *ParameterDeclaration*)*] `)`

ParameterDeclaration ::= *Type Ident* JmlFormalParameter

A constructor block is a special instance of a regular block (see *CompoundStatement*) where the first statement may be a super constructor call. Sometimes *Jml* adds implicitly such a super call although it doesn't appear in the source code. Calling another constructor in the same class (e.g. `this();`) is not allowed since there is only one constructor per class expected.

ConstructorBlock ::= `{` [*SuperConstructorInvocation*] (*Statement*)* `}` JConstructorBlock

SuperConstructorInvocation ::= `super(` [*ExpressionList*] `);` JExplicitConstructorInvocation

Methods may have specification clauses. If we add specifications to an overriding method the specification block has to start with `also`. Only a small subset of all the possible *Jml* specification constructs is supported. Basically we only support preconditions, postconditions and Modifies clauses. The ordering of the clauses is crucial. [1]

MethodSpecification	::=	(*RequiresClause*)* (*EnsuresClause* | *ModifiesClause*)*	JmlSpecification
		`/*@ also */` (*RequiresClause*)* (*EnsuresClause* | *ModifiesClause*)*	JmlExtendingSpecification

Requires and ensures clauses expect an expression with boolean return value. Since we don't support access modifiers there should be no visibility conflicts. There is a special requires case which is discussed on in section 3.4.1, *Translation of the One-Expression* on page 17.

RequiresClause ::= `/*@ requires` *Expression* `; */` JmlRequiresClause

EnsuresClause ::= `/*@ ensures` *Expression* `; */` JmlEnsuresClause

Modifies clauses consist of a list of store reference expressions denoting different memory locations which may be altered during execution of the method. Since the *Jml* class JmlName offers a rather simple interface, we don't support a very flexible grammar for store reference expressions. Every expression is supposed to start with `this`. Then one can continue with a certain field of the class or simply end with `.*` which basically means that the entire object instance can be changed. If the denoted field is an array we can also specify which elements of this array are altered. Either we specify a certain index using `[` *Expression* `]` or we allow every array element to be changed using `[*]`.

ModifiesClause ::= `/*@ modifies` *StoreRefExpression* (, *StoreRefExpression*)* `; */` JmlAssignableClause

StoreRefExpression ::= `this` [. *Ident*] *StoreRefNameSuffix* JmlConditionalStoreRef and JmlStoreRefExpression

StoreRefNameSuffix	::=	. *Ident*	JmlName (isIdent())
	|	`.*`	JmlName (isFields())
	|	[*Expression*]	JmlName (isPos())
	|	`[*]`	JmlName (isAll())

2.4 Statements

A compound statement is simply a sequence of statements.

CompoundStatement ::= `{` (*Statement*)$^+$ `}` JCompoundStatement or JBlock

[1] Internally *Jml* has the following grammar for this subset of specification clause:

JmlSpecification	::=	(JmlGenericSpecCase ⊂ JmlSpecCase)*
JmlGenericSpecCase	::=	(JmlRequiresClause)* [JmlGenericSpecBody, JmlNormalSpecBody ⊂ JmlSpecBody]
JmlGenericSpecBody	::=	(JmlRequiresClause, JmlEnsuresClause, JmlAssignableClause ⊂ JmlSpecBodyClause)*
JmlNormalSpecBody	::=	(JmlNormalSpecClause)*

One of the strongest limitations in this subset is the differentiation between a *RightHandSide* and an *Expression*. In *Jml* method calls [2] and the creation of new object or array instances [3] are allowed in every expression. Here we restrict this constructs to appear only on the right hand side of an assignment [4] or a method may be called as a simple statement. Using some additional helper variables this limitation can be overcome quiet easily.

The only supported loop statement is a while-loop. For-loops can be simulated using a while loop.

At this point we introduce two new statements, namely `pack` and `unpack`. The provided identifier indicates *as* what type the referenced object should be *packed* or *from* which type the object should be *unpacked*. Packing or unpacking as `Object` is not possible.

Statement	::=	*CompoundStatement*	
	|	*VariableDeclarations* `;`	
	|	*MethodCall* `;`	JExpressionStatement and BMethodCallExpression
	|	*Expression* = *RightHandSide* `;`	JExpressionStatement and BAssignmentExpression
	|	`if` `(` *Expression* `)` *Statement* [`else` *Statement*]	JIfStatement
	|	(*Maintaining*)* `while` `(` *Expression* `)` *Statement*	JmlLoopStatement and JWhileStatement
	|	`return` [*Expression*] `;`	JReturnStatement
	|	`;`	JEmptyStatement
	|	`/*@ assert` *Expression* `; */`	JmlAssertStatement
	|	`/*@ assume` *Expression* `; */`	JmlAssumeStatement
	|	`/*@ pack` *Expression* `:` *Ident* `; */`	BPackStatement
	|	`/*@ unpack` *Expression* `:` *Ident* `; */`	BUnpackStatement

As for fields, we don't allow a variable to hide another variable in an outer block or a field of the class, i.e. every variable has to have a unique name.

VariableDeclarations	::=	*Type VariableDeclarator* (`,` *VariableDeclarator*)* `;`	JVariableDeclarationStatement

VariableDeclarator	::=	*Ident* [= *Expression*]	BVariableDefinition <: JmlVariableDefinition <: JVariableDefinition <: JLocalVariable

As mentioned above all right hand side expressions are a special subset of expressions which are only allowed to appear in a special context.

RightHandSide	::=	*Expression*
	|	*NewExpression*
	|	*MethodCall*

MethodCall	::=	*DereferenceExpression* `(` [*ExpressionList*] `)`	BMethodCallExpression <: JMethodCallExpression

In the new-expression we made a little change so that one can specify an owner of the newly created object or array instance. Nevertheless, declaring an owner is optional. An owner consists of an owner object/reference and an owner type.

NewExpression	::=	`new` [*Owner*] *Type* `(` [*ExpressionList*] `)`	BNewObjectExpression
	|	`new` [*Owner*] *Type* `[` *Expression* `]`	BNewArrayExpression and JArrayDimsAndInits (1 dim.)

[2] *Jml* grammar derivation of a method call: *Expression* ::= ... ::= *DereferenceExpression* | ... ::= *DereferenceExpression* `(` [*ExpressionList*] `)` | ...
[3] *Jml* grammar derivation of a new-expresssion: *Expression* ::= ... ::= *DereferenceExpression* | ... ::= *PrimaryExpression* | ... ::= *NewExpression* | ...
[4] *Jml* grammar derivation of an assignment: *Statement* ::= *ExpressionStatement* | ... ::= *Expression* = *Expression* | *Expression*

Owner ::= < *DereferenceExpression* , *ReferenceType* > BOwner

In the maintaining-clause one can declare a condition that holds throughout every execution of a loop.

Maintaining ::= `/*@ maintaining` *Expression* `; */` JmlLoopInvariant

2.5 Expressions

Expressions are rather straightforward. Besides the limitations mentioned above dealing with method calls and new-statements, basically all major expressions are supported. We didn't include shifting operators, bit operators and unary increment and decrement operators.

ExpressionList ::= *Expression* [, *ExpressionList*] JExpressionListStatement

Expression ::= *ImpliesExpression*
| *ImpliesExpression EquivalenceOp EqualityExpression* JmlRelationalExpression

EquivalenceOp ::= `<==>` | `<=!=>`

ImpliesExpression ::= *LogicalOrExpression*
| *LogicalOrExpression* (`==>` *LogicalOrExpression* $)^{+}$ JmlRelationalExpression
| *LogicalOrExpression* (`<==` *LogicalOrExpression* $)^{+}$ JmlRelationalExpression

LogicalOrExpression ::= *LogicalAndExpression*
| *LogicalAndExpression* `||` *LogicalOrExpression* JConditionalOrExpression

LogicalAndExpression ::= *EqualityExpression*
| *EqualityExpression* `&&` *LogicalAndExpression* JConditionalAndExpression

EqualityExpression ::= *RelationExpression*
| *RelationExpression EqualityOp RelationExpression* JEqualityExpression
| *RelationExpression* `instanceof` *Type* JInstanceofExpression

EqualityOp ::= `==` | `!=`

RelationExpression ::= *AdditiveExpression*
| *AdditiveExpression RelationOp AdditiveExpression* JmlRelationalExpression

RelationOp ::= `<` | `<=` | `>=` | `>` | `<:`

AdditiveExpression ::= *MultExpression*
| *MultExpression* `+` *AdditiveExpression* JAddExpression
| *MultExpression* `-` *AdditiveExpression* JMinusExpression

MultExpression	::=	*UnaryExpression*	
	|	*UnaryExpression* * *MultExpression*	JMultExpression
	|	*UnaryExpression* / *MultExpression*	JDivideExpression
	|	*UnaryExpression* % *MultExpression*	JModuloExpression

UnaryExpression	::=	*CastExpression*	
	|	- *UnaryExpression*	JUnaryExpression
	|	! *UnaryExpression*	JUnaryExpression

CastExpression	::=	*DereferenceExpression*	
	|	(*Type*) *UnaryExpression*	JCastExpression

DereferenceExpression	::=	*PrimaryExpression*	
	|	*DereferenceExpression* . *Ident*	JClassFieldExpression and JArrayLengthExpression
	|	*DereferenceExpression* [*Expression*]	JArrayAccessExpression

PrimaryExpression	::=	*Ident*	
	|	*Literal*	
	|	`super`	JSuperExpression only allowed for method calls - not for field access
	|	`true` | `false`	JBooleanLiteral
	|	`this`	JThisExpression
	|	`null`	JNullLiteral
	|	(*Expression*)	JParenthesedExpression
	|	*JmlPrimary*	

As mentioned earlier on, the only basic numeric type we support is `int`. Therefore the only supported numeric literal is one that represents an integer number. Other literals, e.g. strings, are not expected.

Literal	::=	0 | 1 ...	Integer: JOrdinalLiteral iff type is *int*

Below one can find the specification specific expressions that we will support. We added three new expressions. The one-expression helps to reference the type of an object from the callers and the callees perspective. It is only allowed to appear in requires clauses in the form: `requires inv == \one;`. Section 3.4.1, *Translation of the One-Expression*, on page 17 explains the use and translation of `\one` in more detail. The other two new expressions, `\ownerobject(`*Expression* `)` and `\ownertype(`*Expression* `)`, help to access the owner fields introduced in the new-statements.

JmlPrimary	::=	`\old(` *Expression* `)`	JmlOldExpression
	|	`\typeof(` *Expression* `)`	JmlTypeOfExpression
	|	`\type(` *Type* `)`	JmlTypeExpression
	|	`\result`	JmlResultExpression
	|	`\one`	BOneExpression
	|	`\ownerobject(`*Expression* `)`	BOwnerObjectExpression
	|	`\ownertype(`*Expression* `)`	BOwnerTypeExpression

3 Translation to *BoogiePL*

3.1 Setup

We start every translation with the definition and introduction of some uninterpreted functions to model inheritance, heap and field access and to simplify some predicates used during translation later on. Furthermore we define types, constants and global variables.

This definitions are written to the output file by the method `writeSetup()` implemented in class `org.jtobpl.translation.Translator`.

We start with the definition of type `elements` which will be used for arrays.

```
type elements;
```

The heap is modeled using several arrays. For each type we define a special heap.

```
var ObjectHeap: [ref, name]ref;
var BoolHeap: [ref, name]bool;
var IntHeap: [ref, name]int;
var ElemementsHeap: [ref]elements;
var NameHeap: [ref, name]name;
```

Literal names for special fields.

```
const alloc: name;
const inv: name;
const commit: name;
```

Using the following functions we get access to the readonly fields.

```
function length(ref) returns (int);
function ownerObject(ref) returns (ref);
function ownerType(ref) returns (name);
```

Next, we introduce a couple of functions to access and modify arrays.

```
function boolElementSelect(elements, int) returns (bool);
function intElementSelect(elements, int) returns (int);
function refElementSelect(elements, int) returns (ref);
function boolElementStore(elements, int, bool) returns (elements);
function intElementStore(elements, int, int) returns (elements);
function refElementStore(elements, int, ref) returns (elements);
```

The array access and modification functions have to obey the following axioms:

```
axiom( forall e: elements, i: int, j: int, val: bool ::
   ( (i == j) ==> (boolElementSelect(boolElementStore(e, i, val), j) == val) ) &&
   ( (i != j) ==> (boolElementSelect(boolElementStore(e, i, val), j) == boolElementSelect(e, j)) ) );

axiom( forall e: elements, i: int, j: int, val: int ::
   ( (i == j) ==> (intElementSelect(intElementStore(e, i, val), j) == val) ) &&
   ( (i != j) ==> (intElementSelect(intElementStore(e, i, val), j) == intElementSelect(e, j)) ) );

axiom( forall e: elements, i: int, j: int, val: ref ::
   ( (i == j) ==> (refElementSelect(refElementStore(e, i, val), j) == val) ) &&
   ( (i != j) ==> (refElementSelect(refElementStore(e, i, val), j) == refElementSelect(e, j)) ) );
```

3.2 Types

We will only use the following types of *BoogiePL* throughout the translation:

- `bool` — This type is corresponding to the *Java/Jml* type `boolean` and represents the boolean values `true` and `false`.
- `int` — Corresponding to the *Java/Jml* type `int`, representing the integer numbers.
- `ref` — A reference type is comparable to basic pointers and will be used for all sorts of *Java/Jml* object and array instances. A variable of type `ref` might have the value `null` which is a built-in literal. Equality and inequality are the supported operations on type `ref`.
- `name` — Type `name` represents different kinds of defined names such as variable names, field names, method names or names of *Java/Jml* types. The language supports equality, inequality and the partial order (`<:`) operation.
- `elements` — As introduced in section 3.1, *Setup*, on page 11 we use the user-defined type `elements` for all sorts arrays.

First, we need a mapping of *Java/Jml* types to *BoogiePL* types. This translation is implemented by the following method: `BPLType translateCoarseType(CType type)` which can be found in class `org.jtobpl.translation.TranslatorOrInspector`.

CoarseType ⟦ `boolean` ⟧ = `bool`
CoarseType ⟦ `int` ⟧ = `int`
CoarseType ⟦ *ReferenceType* ⟧ = `ref`

Note: Type `void` will not appear in a *CoarseType* translation.

Since *BoogiePL* doesn't offer a very rich type system we model *Java/Jml* types as *BoogiePL* names and therefore we need the following mapping of *Java/Jml* types to *BoogiePL* names. Method `String translateType(CType type)` in class `org.jtobpl.translation.TranslatorOrInspector` implements the translation.

Type ⟦ `boolean` ⟧	=	basicBool	
Type ⟦ `int` ⟧	=	basicInt	
Type ⟦ `Object` ⟧	=	Object	
Type ⟦ *Ident:I* ⟧	=	*I*	for all user defined classes
Type ⟦ *Type:T* `[]` ⟧	=	*arrayType* (*Type* ⟦ *T* ⟧)	

Note: Type `void` will not appear in a *Type* translation.

Throughout the translation we will use the following notation: ↪ (`var` *varName*: `BPLType` *type*, `BPLName` *name*;) to indicate that we introduce a new variable with name *varName* , *BoogiePL* type *type* and *BoogiePL* name *name*. Normally, if *type* equals `ref` we don't need to care about *name*. Since local variables can only be introduced in the beginning of a *BoogiePL* procedure we have to collect them temporarily as we introduce them during translation and place them in front of all statements later on. See also section 3.4.4, *Method Body Translation*, on page 22.

Constant literals denote the built-in type `Object` and the basic types `int` and `boolean`. All other names of class types will be introduced as we translate every class.

```
const basicBool: name;
const basicInt: name;
const Object: name;
```

Uninterpreted function *isProperSubClass* helps to specify a direct subclass.

```
function isProperSubClass(subclass: name, superclass: name) returns (bool);
axiom( forall n: name, o: name :: isProperSubClass(n, o) <==> (n <: o && n != o) );
```

The function *arrayType* maps a type *T* to an array type with elements of type *T*.

```
function arrayType(name) returns (name);
```

Following three axioms for array types:

```
axiom( forall n: name :: isProperSubClass(arrayType(n), Object) );
axiom( forall n: name, o: name :: (arrayType(n) == arrayType(o)) <==> (n == o) );
axiom( forall n: name, o: name :: (n <: o) ==> (arrayType(n) <: arrayType(o)) );
```

Function *isOfType* is used as predicate to model the type system and inheritance for references.

```
function isOfType(ref, name) returns (bool);
axiom( forall r: ref, n: name :: isOfType(r, n) <==> (r == null || typeOf(r) <: n) );
```

Function *typeOf* maps a reference to its name (if it was defined by *isOfType* earlier on).

```
function typeOf(ref) returns (name);
```

The following two functions assign class members to their classes and types.

```
function fieldType(name) returns (name);
function fieldHome(name) returns (name);
```

And finally function *isAllocated* to model allocation on the heap.

```
function isAllocated(ref, [ref, name]bool) returns (bool);
axiom( forall r: ref, BoolHeap:[ref, name]bool ::
   isAllocated(r, BoolHeap) <==> (r == null || BoolHeap[r, alloc]) );
```

3.3 Programs and Classes

Now we start with the real translation which is predominately implemented by the two visitor classes `Inspector` and `Translator` both part of package `org.jtobpl.translation`.

We go through every compilation unit and translate class by class, ignoring `java.lang.Object` if it is one of the encountered classes.

```
Tr ⟦ (ClassDefinition)*:CDs ⟧  =  # for all c ∈ CDs
                                       Tr ⟦ c ⟧
                                   # end for all
```

Throughout the following translations $\mathfrak{C}$ will be used to denote the name of the currently translated class. Initialization expressions of class fields will be handled in the constructor. Class invariants are only needed in `pack` and `unpack` statements. If a dummy constructor is needed we invoke method `void translateDummyConstructor(BDummyConstructor dummyConstructor)` in class `org.jtobpl.translation.Translator`.

Following the translation rules for a class that implicitly extends `java.lang.Object`.

```
Tr ⟦ class Ident:C { (Member)*:Ms } ⟧  =

        const C: name;
        axiom( C <: Object );
        # for all ( Type:T Ident:I [ = Expression ] ; ∈ Ms
            const C.I: name;
            # if T is ReferenceType then
                axiom( fieldHome(C.I) == Type ⟦ C ⟧ );
                axiom( fieldType(C.I) == Type ⟦ T ⟧ );
            # end if
        # end for all
        # for all MethodDeclaration:MD ∈ Ms
            Tr ⟦ MD ⟧
        # end for all
        # if there is no constructor defined within class C
            Tr ⟦ C () {;} ⟧
        # end if
```

And the translation rules for a class that explicitly extends another class.

Tr ⟦ `class` *Ident:C* `extends` *Ident:E* `{` (*Member*)*:*Ms* `}` ⟧ =

```
const C: name;
axiom( C <: E );
# for all ( Type:T Ident:I [ = Expression ] ; ∈ Ms
    const C.I: name;
    # if T is ReferenceType then
        axiom( fieldHome(C.I) == Type⟦ C ⟧ );
        axiom( fieldType(C.I) == Type⟦ T ⟧ );
    # end if
# end for all
# for all MethodDeclaration:MD ∈ Ms
    Tr⟦ MD ⟧
# end for all
# for all MethodDeclaration:MDsuper in superclasses of C which are not overridden
    TrDummyMethod⟦ MDsuper ⟧
# end for all
# if there is no constructor defined within class C
    Tr⟦ C () {;} ⟧
# end if
```

3.4 Method and Constructor Declarations and Specifications

Translation of a regular method. Since overloading is not supported $\mathfrak{C}.N$ will unambiguously identify the *BoogiePL* procedure that we want to call. If we would like to support overloading too, we had to introduce a different naming rule for procedures at it is done in the *Spec#* translation to *BoogiePL*.

Tr ⟦ *MethodSpecification:MS Type:RT Ident:N Formals:F CompoundStatement:Body* ⟧ =

```
procedure ℭ.N TrSig⟦ N , F , RT ⟧ ; TrSpec⟦ N , MS ⟧
implementation ℭ.N TrSig⟦ N , F , RT ⟧ TrMethodBody⟦ N , F , RT , Body ⟧
```

Translation of a constructor ($N = \mathfrak{C}$). Because we only allow one constructor per class (and implement a dummy constructor if there is no constructor given in the source code) we always know which constructor we have to call when we create a new object instance. All class fields will be initialized in the constructor although they may not be mentioned explicitly in the constructor implementation. That's why we collect all field declarations in *FieldDecList* and have a constructor specific body translation *TrConstructorBody*.

Tr ⟦ *MethodSpecification:MS Ident:N Formals:F CompoundStatement:Body* ⟧ =

```
procedure ℭ.N TrSig⟦ N , F , void ⟧ ; TrSpec⟦ N , MS ⟧
# var FieldDecList = ∅ ;
# for all FieldDeclaration:FD in class ℭ and all its superclasses
    # FieldDecList = FieldDecList ∪ FD ;
# end for all
implementation ℭ.N TrSig⟦ N , F , void ⟧ TrConstructorBody⟦ N , F , FieldDecList , Body ⟧
```

A dummy method is added if class $\mathfrak{C}$ inherits this method from its superclass $\mathfrak{B}$ but doesn't overwrite it. This is only done for regular methods - not for constructors. Section 3.4.1, *Translation of the One-Expression*, on page 17 explains in more detail why we include an assert and assume statement if `requires inv == \one` appears in the method specification *MS*.

This translation is implemented by the method `void translateDummyMethod(BDummyMethod dummyMethod)` in class `org.jtobpl.translation.Translator`.

```
TrDummyMethod ⟦ MethodSpecification:MS  Type:RT  Ident:N  Formals:F ⟧  =

        procedure ℭ.N TrSig ⟦ N , F , RT ⟧ ; TrSpec ⟦ N , MS ⟧
        implementation ℭ.N TrSig ⟦ N , F , RT ⟧
          # VDs = variables introduced in pack and unpack statement
          # PMs = variables introduced in signature translation
          # generate symbol start
          {
            VDs
            start:
              # if (requires inv == \one) appears in method specs MS
                assume NameHeap[this, inv] == ℭ;
              # end if
              AssumeTypes ⟦ VDs ∪ PMs ⟧
              # if (requires inv == \one) appears in method specs of N
                assert NameHeap[this, inv] == typeOf(this);
              # end if
              Tr ⟦ unpack this :  ℭ ; ⟧
              # if RT is void
                call 𝔅.N ( this
              # else
                call N.result := 𝔅.N ( this
              # end if
              # for all ( Type Ident:I ) ∈ F
                , I
              # end for all
              );
              Tr ⟦ pack this :  ℭ ; ⟧
              return;
          }
```

Translation of a method or constructor signature. For every parameter we introduce a new variable which will only be used in the *AssumeTypes* function that we will call inside the procedure body to make sure that all variables and parameters are allocated and a *BoogiePL* name is defined for references.

This translation is implemented by the method
`void translateSignature(BMethodOrConstructorDeclarationInterface methodOrConstructor)`
in class `org.jtobpl.translation.Translator`.

TrSig ⟦ *Ident:N* , *Formals:F* , *Type:RT* ⟧ = `(this: ref`
 # for all (*Type:T Ident:I*) ∈ *F*
`,` *I*`:` *CoarseType* ⟦ *T* ⟧
↪ `( var` *I*`: BPLType` *CoarseType* ⟦ *T* ⟧`, BPLName` *Type* ⟦ *T* ⟧`; )`
 # end for all
 # if *RT* **is** `void`
`)`
 # else
`) returns (` *N*`.return:` *CoarseType* ⟦ *RT* ⟧ `)`
 # end if

The *TrSpec* function is called for every method and constructor, regardless whether there is a specification or not.

The implementation of this translation can be found in method
`void translateSpecification(BMethodDeclaration methodNode)` in class `org.jtobpl.translation.Translator`.

TrSpec ⟦ *Ident:M* , *MethodSpecification:Specs* ⟧ =

`requires` *TrRequires* ⟦ ℭ , *M* ⟧ ;
`ensures` *TrEnsures* ⟦ ℭ , *M* ⟧ ;
ModifiesContribution ⟦ ℭ , *M* ⟧

3.4.1 Translation of the One-Expression

Expression `\one` is only allowed to appear in preconditions or regular methods in the form `requires inv == \one;`. If there is such a requires clause in method specification *MS* of method *M*, defined in class ℭ𝔩𝔞𝔰𝔰, we don't take this clause into account when translating the specifications of *M*. However, we insert the following assertion code before calling *M* in every caller:

```
...
receiver := Tr ⟦ DereferenceExpression ⟧;
assert receiver != null;
assert NameHeap[receiver, 𝔦𝔫𝔳] == typeOf(receiver);
call ℭ𝔩𝔞𝔰𝔰.M ( receiver, ...
...
```

Inside the body of the *BoogiePL* procedure 𝔆𝔩𝔞𝔰𝔰.*M* corresponding to method *M*, we make the following assumption:

```
...
{
   start:
      assert NameHeap[this, inv] == typeOf(this);
      ...
```

3.4.2 Pre- and Postconditions

Note, the following translation of pre- and postconditions is not equivalent with the semantics of pre- and postconditions in *Jml*.

For every method *Method* we build the conjunction of its requires predicates (*req*). For method *Method* and all the methods that *Method* has overridden we build the disjunction of this conjunctions (... `||` *req′* `||` *req*).

This translation is implemented by the method
`void translateRequires(BClassDeclaration classNode, BMethodDeclaration methodNode)`
in class `org.jtobpl.translation.Translator`.

TrRequires ⟦ *Ident:Class* , *Ident:Method* ⟧ = **# if method** *Method* **is not defined in class** *Class*
  `false`
 # else
  # *SupClass* **is superclass of** *Class*
  # *MS* **is method specification of** *Method* **in** *Class*
  # var *req* **=** `true`
  # for all (`/*@ requires` *Expression:R*`; */` **)** ∈ *MS*
    # (`/*@ requires inv == \one */` **) will be ignored here**
    # *req* `&&` *Tr* ⟦ *R* ⟧
  # end for all
  TrRequires ⟦ *SupClass* , *Method* ⟧ `||` (*req*)
 # end if

Similarly for every method *Method* we build the conjunction of its requires predicates (*req*) and the conjunction of its ensures predicates (*ens*). Next, we build an implication, letting the precondition imply the postcondition (`old`(*req*)`==>` *ens*). For method *Method* and all the methods that *Method* has overridden we build a conjunction of this implications (... `&&` (`old`(*req'*)`==>` *ens'*) `&&` (`old`(*req*)`==>` *ens*)).

The implementation of this translation can be found in method
`void translateEnsures(BClassDeclaration classNode, BMethodDeclaration methodNode)`
in class `org.jtobpl.translation.Translator`.

```
TrEnsures ⟦ Ident:Class , Ident:Method ⟧  =  # if method Method is not defined in class Class
                                                 true
                                             # else
                                                 # SupClass is superclass of Class
                                                 # MS is method specification of Method in Class
                                                 # var req = true;
                                                 # for all ( /*@ requires Expression:R; */ ) ∈ MS
                                                     # ( /*@ requires inv == \one */ ) will be ignored here
                                                     # req && Tr ⟦ R ⟧
                                                 # end for all
                                                 # var ens = true;
                                                 # for all ( /*@ ensures Expression:E; */ ) ∈ MS
                                                     # ens && Tr ⟦ E ⟧
                                                 # end for all
                                                 TrEnsures ⟦ SupClass , Method ⟧ && ( old(req) ==> ens )
                                             # end if
```

3.4.3 Modifies Clauses

Note, the following translation of modifies clauses is not equivalent with the semantics of modifies clauses in *Jml*.

First, we state that our procedure may change all possible heaps using *BoogiePL*'s own modifies statement. Then we ensure that for all objects o and fields f:

$o.f$ `==` `old`($o.f$)	The field is unchanged or ...
$o.f \in m$	... f appears in the modifies list m or ...
`!old(`o`.alloc)`	... o wasn't allocated when entering the method or ...
`old(`o`.commit)`	... o was committed when we entered the method.

A modifies statement is only valid if the precondition is satisfied. Therefore we build a similar conjunction of implications as we did for postconditions (... `&&` (`old`(*req'*) `==>` *mod'*) `&&` (`old`(*req*) `==>` *mod*)).

This following three translations are implemented by the functions
`void modifiesContribution(...)`, `void translateModifiesFields(...)` and `void translateModifiesArrays(...)`.
all in class `org.jtobpl.translation.Translator`.

```
ModifiesContribution [[ Ident:Class , Ident:Method ]] =

    modifies BoolHeap , IntHeap , ObjectHeap , ElemementsHeap , NameHeap ;
    # for all h ∈ { BoolHeap, IntHeap, ObjectHeap, ElemementsHeap, NameHeap }
      ensures( forall x: ref, n: name ::
        # if h == ElemementsHeap
          ( h[x] == old(h)[x] ) ||
        # else
          ( h[x, n] == old(h)[x, n] ) ||
        # end if
        ( TrModifiesFields [[ Class , Method , x , n ]] )
        || ( ! old(BoolHeap)[x,alloc] ) || ( old(BoolHeap)[x,commit] )
      );
    # end for all
    # for all selectFunc ∈ { boolElementSelect, intElementSelect, refElementSelect }
      ensures( forall x: ref, i: int ::
        ( selectFunc(ElemementsHeap[x], i) == selectFunc(old(ElemementsHeap)[x], i) ) ||
        ( TrModifiesArrays [[ Class , Method , x , i ]] )
        ( ! old(BoolHeap)[x,alloc] ) || ( old(BoolHeap)[x,commit] )
      );
    # end for all


TrModifiesFields [[ Ident:Class , Ident:Method , Ref , Name ]] =

    # if method Method is not defined in Class
      true
    # else
      # SupClass is superclass of Class
      # MS is method specification of Method in Class
      # var req = true;
      # for all ( /*@ requires Expression:R; */ ) ∈ MS
        # ( /*@ requires inv == \one */ ) will be ignored here
        # req && Tr [[ R ]]
      # end for all
      # var m = ∅ ;
      # for all ( /*@ modifies ( StoreRefExpression ( , StoreRefExpression )* ):Mlist ; */ ) ∈ Specs
        # m = m ∪ Mlist;
      # end for all
      # var mod = false ;
      # for all (StoreRefName:Pref StoreRefNameSuffix:Suf ) ∈ m
        # if Suf == ( . Ident:I ) then
          # mod || ( Ref == old(Tr [[ Pref ]]) && Name == I )
        # if Suf = ( .* ) then
          # mod || ( Ref == old(Tr [[ Pref ]]) )
        # end if
      # end for all
      TrModifiesFields [[ SupClass , Method , Ref , Name ]] && ( old(req) ==> mod )
    # end if
```

TrModifiesArrays ⟦ *Ident:Class* , *Ident:Method* , *Ref* , *Index* ⟧ =

 # if method *Method* **is not defined in** *Class*
    `true`
 # else
    # *SupClass* **is superclass of** *Class*
    # *MS* **is method specification of** *Method* **in** *Class*
    # var *req* = `true`;
    # for all (`/*@ requires` *Expression:R*`; */` **)** ∈ *MS*
        # (`/*@ requires inv == \one */` **) will be ignored here**
        # *req* `&&` *Tr* ⟦ *R* ⟧
    # end for all
    # var *m* = ∅ `;`
    # for all (`/*@ modifies` (*StoreRefExpression* (`,` *StoreRefExpression*)*):*Mlist* `; */`) ∈ *Specs*
        # *m* = *m* ∪ *Mlist*;
    # end for all
    # var *mod* = `false ;`
    # for all (*StoreRefName:Pref* *StoreRefNameSuffix:Suf*) ∈ *m*
        # if *Suf* = (`[` *Expression:E* `]`) **then**
            # *mod* `||` `(` *Ref* `==` `old`(*Tr* ⟦ *Pref* ⟧) `&&` *Index* `==` `old`(*Tr* ⟦ *E* ⟧) `)`
        # if *Suf* = (`[*]`) **then**
            # *mod* `||` `(` *Ref* `==` `old`(*Tr* ⟦ *Pref* ⟧) `)`
        # end if
    # end for all
    TrModifiesArrays ⟦ *SupClass* , *Method* , *Ref* , *Name* ⟧ `&&` `(` `old`(*req*) `==>` *mod* `)`
 # end if

3.4.4 Method Body Translation

Following we state the translation rules for methods and constructors. The only difference is, that within the constructor all fields of the newly created object instance are implicitly initialized. Either the given initialization value is used or we use the function *Zero* to create an initial value for a given type. Inside the body, we list all local variables that have been introduced during translation or did appear as variable declarations in the original method code. Next, we invoke *AssumeTypes* to make sure that all references are marked as allocated and the *BoogiePL* name representing the original class type is known. Finally we place a return statement if the method has no return value and we leave the procedure.

The implementation of the next two translations can be found in the methods
`void translateMethodBody(BMethodDeclaration method)` and
`void translateConstructorBody(BConstructorDeclaration constructor)`
both in class `org.jtobpl.translation.Translator`.

TrMethodBody ⟦ *Ident:N* , *Formals:F* , *Type:RT* , *CompoundStatement:Body* ⟧ =

```
# VDs = variables introduced during translation of Body.
# PMs = variables introduced in signature translation of N
# generate symbol start
  {
    VDs
    start:
      # if (requires inv == \one) appears in method specs of N
        assume NameHeap[this, inv] == ℭ;
      # end if
      AssumeTypes ⟦ VDs ∪ PMs ⟧
      Tr ⟦ Body ⟧
      # if RT is void
        return;
      # end if
  }
```

TrConstructorBody ⟦ *Ident:N* , *Formals:F* , (*VariableDeclarations*)*:*Members* , *CompoundStatement:Body* ⟧ =

```
# VDs = variables introduced during translation of Body.
# PMs = variables introduced in signature translation of constructor
# generate symbol start
  {
    VDs
    start:
      AssumeTypes ⟦ VDs ∪ PMs ⟧
      # for all FieldDeclaration:FD ∈ Members
        # if FD = ( Type:T Ident:I = Expression:E ; )
          Assign ⟦ this.I , E ⟧
        # if FD = ( Type:T Ident:I ) ;
          Assign ⟦ this.I , Zero ⟦ T ⟧ ⟧
        # end if
      # end for all
      Tr ⟦ Body ⟧
      return;
  }
```

As we have mentioned them quiet a few times above, functions *AssumeTypes* and *GlobalTypes* are used to make sure that every reference is marked as allocated and that the *BoogiePL* names of references are known. *AssumeTypes* is implemented by method `void assumeTypes(BPLVariable[] variableBindings)` in class `org.jtobpl.translation.Translator`.

```
AssumeTypes ⟦ variableBindings ⟧  =  # for all ( var x: BPLType T, BPLName N; ) ∈ variableBindings
                                         # if T = ref  then
                                             assume isOfType(x, N) && isAllocated(x, BoolHeap);
                                         # end if
                                     # end for all
                                     GlobalTypes ⟦ ⟧
```

GlobalTypes is implemented by method `void globalTypes()` in class `org.jtobpl.translation.Translator`.

```
GlobalTypes ⟦ ⟧  =  assume( forall x: ref, n: name :: isOfType(ObjectHeap[x, n], fieldType(n)) );

                    assume( forall x: ref, n: name ::
                        BoolHeap[x, alloc] ==> isAllocated(ObjectHeap[x, n], BoolHeap) );

                    assume( forall a: ref, n: name, i: int :: isOfType(a, arrayType(n)) ==>
                        isOfType(refElementSelect(ElemementsHeap[a], i), n) );

                    assume( forall a: ref, i: int :: BoolHeap[a, alloc] ==>
                        isAllocated(refElementSelect(ElemementsHeap[a], i), BoolHeap) );
```

3.5 Statements

Next, we describe the translation of statements. Translating compound statements and the empty statement is straightforward.

```
Tr ⟦ CompoundStatement:CS ⟧  =  # for all s∈ CS
                                    Tr ⟦ s ⟧
                                # end for all
```

```
Tr ⟦ ; ⟧  =  ∅
```

3.5.1 Variable Declaration

As we have mentioned earlier, every local variable that is needed in a *BoogiePL* procedure needs to be declared in the beginning of the procedure. Therefore we collect all local variables and assign an initial value, either given as an expression or by the *Zero* translation, to them.

```
Tr ⟦ Type:T ( VariableDeclarator ( , VariableDeclarator)*):VDs ; ⟧ =

                    # for all VD ∈ VDs
                      # if VD = ( Ident:I = Expression:E )
                        # ↪ ( var I: BPLType CoarseType ⟦ T ⟧, BPLName Type ⟦ T ⟧; )
                        Assign ⟦ I , E ⟧
                      # else VD = ( Ident:I )
                        # ↪ ( var I: BPLType CoarseType ⟦ T ⟧, BPLName Type ⟦ T ⟧; )
                        Assign ⟦ I , Zero ⟦ T ⟧ ⟧
                      # end if
                    # end for all
```

Zero is implemented by method `public JExpression zeroValue(CType type, JPhylum surroundingNode)` in class `org.jtobpl.translation.Translator`.

```
     Zero ⟦ boolean ⟧ = false
         Zero ⟦ int ⟧ = 0
Zero ⟦ ReferenceType ⟧ = null
```

3.5.2 Method Call

If a method within the given class is called, then *JML* will automatically add `this` as *DereferenceExpression*. Let *T* denote the type of *DereferenceExpression:DE*. See also section 3.5.3, *Assignments*, on page 25 for method calls with return values.

```
Tr ⟦ DereferenceExpression:DE . Ident:N ( ExpressionList:EL ) ⟧ =

                    # generate symbol receiver
                    # ↪ ( var receiver: BPLType ref, BPLName Type ⟦ T ⟧; )
                    # if DE != super
                      assert DefCk ⟦ DE ⟧ ;
                      receiver := Tr ⟦ DE ⟧ ;
                      assert receiver != null;
                    # else
                      receiver := this ;
                    # end if
                    # for all e ∈ EL
                      assert DefCk ⟦ e ⟧ ;
                    # end for all
                    # let MS be the method specifications of N
                    # if ( requires inv == \one; ) ∈ MS
                      assert NameHeap [receiver, inv] == typeOf (receiver);
                    # end if
                    # if DE == super and Super is superclass of ℭ
                      call Super.N ( receiver
                    # else - N is defined in Class C
                      call C.N ( receiver
                    # end if
                    # for all e ∈ EL
                      , Tr ⟦ e ⟧
                    # end for all
                    );
```

3.5.3 Assignments

First, we differentiate between different kinds of left hand sides of the given assignment. This can be an object field, an array element or a local variable. Attention: Writing to parameters may not be allowed in *BoogiePL*.

For a field I of type T in a class C:

```
Tr ⟦ DereferenceExpression:E . Ident:I = Expression:Rhs ; ⟧ =

    assert DefCk ⟦ E ⟧;
    # generate symbol e, i
    # ↪ ( var e: BPLType ref, BPLName Type ⟦ C ⟧; )
    # ↪ ( var i: BPLType CoarseType ⟦ T ⟧, BPLName Type ⟦ T ⟧; )
    e := Tr ⟦ E ⟧;
    assert e != null;
    assert isProperSubClass(C, NameHeap[e, inv]);
    Assign ⟦ i , Rhs ⟧
    # if type T is boolean
        BoolHeap[e, C.I] := i;
    # if type T is int
        IntHeap[e, C.I] := i;
    # else
        ObjectHeap[e, C.I] := i;
    # end if
```

For an dereference expression E whose type AT is an array with elements of type T:

```
Tr ⟦ DereferenceExpression:E [ Expression:N ] = Expression:Rhs; ⟧ =

    assert DefCk ⟦ E ⟧ && DefCk ⟦ N ⟧;
    # generate symbol e, f, n
    # ↪ ( var e: BPLType ref, BPLName Type ⟦ AT ⟧; )
    # ↪ ( var n: BPLType int, BPLName basicInt; )
    # ↪ ( var f: BPLType CoarseType ⟦ T ⟧, BPLName Type ⟦ T ⟧; )
    e := Tr ⟦ E ⟧;
    n := Tr ⟦ N ⟧;
    assert e != null;
    assert 0 <= n && n < length(e);
    Assign ⟦ f , Rhs ⟧
    # if type T is boolean
        ElemementsHeap[e] := boolElementStore(ElemementsHeap[e], n, f);
    # if type T is int
        ElemementsHeap[e] := intElementStore(ElemementsHeap[e], n, f);
    # else
        ElemementsHeap[e] := refElementStore(ElemementsHeap[e], n, f);
    # end if
```

Assignment to a local variable:

```
Tr ⟦ Ident:I = Expression:Rhs ; ⟧ = Assign ⟦ I , Rhs ⟧
```

Next, we distinguish between the four possible right hand sides of an assignment. We may deal with a method call, a new object instance expression, a new array instance expression or a regular expression. Depending on the type of assignment we call a different *Assign* function.

Assigning a regular expression. This translation is implemented by method
`void assignExpression(BPLVariable I, JExpression Rhs)` in class `org.jtobpl.translation.Translator`.

```
Assign ⟦ Ident:I , Expression:E ⟧  =  assert DefCk ⟦ E ⟧;
                                      I := Tr ⟦ E ⟧;
```

Assigning a new class instance with an owner declaration to a local identifier *I*. Let *OOT* denote the type of *DereferenceExpression:OO*.

The next two translations are implemented by method
`void assignNewClassInstance(BPLVariable I, BNewObjectExpression Rhs)` in class `org.jtobpl.translation.Translator`.

```
Assign ⟦ Ident:I , new < DereferenceExpression:OO, ReferenceType:OT > Type:T ( ExpressionList:EL ) ⟧  =

    havoc I;
    assume I != null;
    assume typeOf(I) == Type ⟦ T ⟧;
    # generate symbol oobj
    # ↪ ( var oobj: BPLType ref, BPLName Type ⟦ OOT ⟧; )
    assert DefCk ⟦ OO ⟧;
    oobj := Tr ⟦ OO ⟧;
    assume ownerObject(I) == oobj;
    assume ownerType(I) == Type ⟦ OT ⟧;
    assume NameHeap[I, inv] == Object;
    assume (forall n: name :: ObjectHeap[I, n] == null);
    assume (forall n: name :: BoolHeap[I, n] == false);
    assume (forall n: name :: IntHeap[I, n] == 0);
    BoolHeap[I, alloc] := true;
    # for all e ∈ EL
        assert DefCk ⟦ e ⟧;
    # end for all
    call T.T ( I
    # for all e ∈ EL
        , Tr ⟦ e ⟧
    # end for all
    );
```

Assigning a new class instance without an owner declaration to a local identifier *I*.

Assign ⟦ *Ident:I* , `new` *Type:T* (*ExpressionList:EL*) ⟧ =

```
havoc I;
assume I != null;
assume typeOf(I) == Type⟦ T ⟧;
assume ownerObject(I) == null;
assume NameHeap[I, inv] == Object;
assume (forall n: name :: ObjectHeap[I, n] == null);
assume (forall n: name :: BoolHeap[I, n] == false);
assume (forall n: name :: IntHeap[I, n] == 0);
BoolHeap[I, alloc] := true;
```

 # for all $e \in EL$

```
  assert DefCk⟦ e ⟧;
```

 # end for all

```
call T.T ( I
```

 # for all $e \in EL$

```
  , Tr⟦ e ⟧
```

 # end for all

```
);
```

Assigning a new array instance with an owner declaration to a local identifier *I*. Let *OOT* denote the type of *DereferenceExpression:OO*.

The next two translations are implemented by method
`void assignNewArrayInstance(BPLVariable I, BNewArrayExpression Rhs)` in class `org.jtobpl.translation.Translator`.

Assign ⟦ *Ident:I* , `new` < *DereferenceExpression:OO* , *ReferenceType:OT* > *Type:T* [*Expression:E*] ⟧ =

```
assert DefCk ⟦ E ⟧;
havoc I;
# generate symbol oobj, n
# ↪ ( var oobj:  BPLType ref, BPLName Type ⟦ OOT ⟧; )
# ↪ ( var n:  BPLType int, BPLName basicInt; )
assert DefCk ⟦ OO ⟧;
oobj  := Tr ⟦ OO ⟧;
assume ownerObject(I) == oobj;
assume ownerType(I) == Type ⟦ OT ⟧;
n  := Tr ⟦ E ⟧;
assert 0 <= n;
assume I  != null;
assume typeOf(I) == arrayType(Type ⟦ T ⟧);
assume length(I) == n;
assume NameHeap[I, inv] == Object;
assume BoolHeap[I, commit] == false;
# if type T is boolean
   assume (forall i:  int ::  boolElementSelect(ElemementsHeap[I], i) == false);
# if type T is int
   assume (forall i:  int ::  intElementSelect(ElemementsHeap[I], i) == 0);
# else
   assume (forall i:  int ::  refElementSelect(ElemementsHeap[I], i) == null);
# end if
BoolHeap[I, alloc] := true;
```

Assigning a new array instance without an owner declaration to a local identifier *I*.

```
Assign ⟦ Ident:I , new Type:T [ Expression:E ] ⟧  =

        assert DefCk ⟦ E ⟧;
        havoc I;
        # generate symbol n
        # ↪ ( var n: BPLType int, BPLName Type ⟦ basicInt ⟧; )
        assume ownerObject(I) == null;
        n := Tr ⟦ E ⟧;
        assert 0 <= n;
        assume I != null;
        assume typeOf(I) == arrayType(Type ⟦ T ⟧);
        assume length(I) == n;
        assume NameHeap[I, inv] == Object;
        assume BoolHeap[I, commit] = false;
        # if type T is boolean
           assume (forall i: int :: boolElementSelect(ElemementsHeap[I], i) == false);
        # if type T is int
           assume (forall i: int :: intElementSelect(ElemementsHeap[I], i) == 0);
        # else
           assume (forall i: int :: refElementSelect(ElemementsHeap[I], i) == null);
        # end if
        BoolHeap[I, alloc] := true;
```

Method call with a return value which is assigned to local variable *I* of type *T*. Let *DET* denote the type of *DereferenceExpression:DE*.

This translations is implemented by method `void assignMethodCall(BPLVariable I, BMethodCallExpression Rhs)` in class `org.jtobpl.translation.Translator`.

Assign ⟦ *Ident:I* , *DereferenceExpression:DE* . *Ident:N* (*ExpressionList:EL*) ⟧ =

```
# generate symbol receiver
# ↪ ( var receiver:  BPLType ref, BPLName Type⟦ DET ⟧; )
# if DE != super
    assert DefCk⟦ DE ⟧ ;
    receiver := Tr⟦ DE ⟧ ;
    assert receiver != null;
# else
    receiver := this ;
# end if
# for all e ∈ EL
    assert DefCk⟦ e ⟧ ;
# end for all
# let MS be the method specifications of N
# if ( requires inv == \one; ) ∈ MS
    assert NameHeap[receiver, inv] == typeOf(receiver);
# end if
# if DE ==  super and Super is superclass of ℭ
    call I := Super.N ( receiver
# else - N is defined in Class C
    call I := C.N ( receiver
# end if
# for all e ∈ EL
    , Tr⟦ e ⟧
# end for all
);
AssumeTypes⟦ T I ⟧
```

3.5.4 Control Flow Statements

Translating control flow statements is straightforward. We exploit the fact that *BoogiePL* chooses an arbitrary block out of the list if a `goto` statement lists more than one label.

If-Then-Else statement.

Tr ⟦ `if` (*Expression:C*) *Statement:T* `else` *Statement:E* ⟧ =

```
assert DefCk⟦ C ⟧;
# generate symbol then, else, join
goto then, else, join;
then:
    assume Tr⟦ C ⟧;
    Tr⟦ T ⟧;
    goto join;
else:
    assume !  Tr⟦ C ⟧;
    Tr⟦ E ⟧;
    goto join;
join:
```

If-Then statement.

Tr ⟦ `if` (*Expression:C*) *Statement:T* ⟧ =

```
assert DefCk⟦ C ⟧;
# generate symbol then, join
goto then, join;
then:
    assume Tr⟦ C ⟧;
    Tr⟦ T ⟧;
    goto join;
join:
```

While loop.

Tr ⟦ (*Maintaining*)*:*M* `while` (*Expression:C*) *Statement:S* ⟧ =

```
# generate symbol top, body, after
goto top;
top:
    # for each m ∈ M
        assert Tr⟦ m ⟧;
    # end for each
    assert DefCk⟦ C ⟧;
    goto body, after;
body:
    assume Tr⟦ C ⟧;
    Tr⟦ S ⟧;
    goto top;
after:
assume !(Tr⟦ C ⟧);
```

Returning from method *M* with a return value. First we assign the return expression to the designated return variable *F*`.return` and then we leave the procedure.

```
Tr ⟦ return Expression:R; ⟧  =  Assign ⟦ M.return , R ⟧
                                return;
```

Returning from a method without a return value.

```
Tr ⟦ return; ⟧  =  return;
```

3.5.5 Specification Statements

Assert and assume statements can be translated using the *BoogiePL* built-in statements `assert` and `assume`.

```
Tr ⟦ /*@ assert Expression:E; */ ⟧  =  assert Tr ⟦ E ⟧;

Tr ⟦ /*@ assume Expression:E; */ ⟧  =  assume Tr ⟦ E ⟧;
```

Let *S* denote the immediate superclass of *T* and let *ObjInv* denote the invariant declarations in class *T*. Furthermore let *ET* denote the type of *Expression:E*.

```
Tr ⟦ /*@ pack Expression:E : Ident:T; */ ⟧  =

    assert DefCk ⟦ E ⟧;
    # generate symbol e, heap
    # ↪ ( var e: BPLType ref, BPLName Type ⟦ ET ⟧; )
    # ↪ ( var heap: BPLType bool[ref, name], BPLName ∅; )
    e := Tr ⟦ E ⟧;
    assert e != null;
    assert NameHeap[e, inv] == S;
    assert ( forall r: ref ::
        (r != null && BoolHeap[r, alloc] && ownerObject(r) == e && ownerType(r) == T) ==>
        NameHeap[r, inv] == typeOf(r) );
    # for all ( /* invariant Expression:I; */ ) ∈ ObjInv
        # let I′ = I in which this has been replaced by e in
        assert Tr ⟦ I′ ⟧;
    # end for all
    heap := BoolHeap;
    havoc BoolHeap;
    assume ( forall r: ref :: BoolHeap[r, commit] <==>
        ( heap[r, commit] || (ownerObject(r) == e && ownerType(r) == T) ) );
    NameHeap[e, inv] := T;
```

Let *ET* denote the type of *Expression:E*.

```
Tr ⟦ /*@ unpack Expression:E : Ident:T; */ ⟧  =

        assert DefCk ⟦ E ⟧;
        # generate symbol e, heap
        # ↪ ( var e:  BPLType ref, BPLName Type ⟦ ET ⟧; )
        # ↪ ( var heap:  BPLType bool[ref, name], BPLName ∅; )
        e  := Tr ⟦ E ⟧;
        assert e != null;
        assert NameHeap[e, inv] == T;
        NameHeap[e, inv] := S;
        heap := BoolHeap;
        havoc BoolHeap;
        assume ( forall r:  ref :: BoolHeap[r, commit] <==>
           ( heap[r, commit] && (ownerObject(r) != e || ownerType(r) != T) ) );
```

3.6 Expressions

There is not to much that needs to be written about expressions. Most of the nodes can be translated ont-to-one to *BoogiePL*.

```
Tr ⟦ ImpliesExpression:E <==> ImpliesExpression:F ⟧  =  Tr ⟦ E ⟧ <==> Tr ⟦ F ⟧
Tr ⟦ ImpliesExpression:E <=!=> ImpliesExpression:F ⟧  =  !(Tr ⟦ E ⟧ <==> Tr ⟦ F ⟧)

Tr ⟦ LogicalOrExpression:E ==> LogicalOrExpression:F ⟧  =  Tr ⟦ E ⟧ ==> Tr ⟦ F ⟧
Tr ⟦ LogicalOrExpression:E <== LogicalOrExpression:F ⟧  =  Tr ⟦ E ⟧ <== Tr ⟦ F ⟧

Tr ⟦ LogicalAndExpression:E || LogicalOrExpression:F ⟧  =  Tr ⟦ E ⟧ || Tr ⟦ F ⟧

Tr ⟦ EqualityExpression:E && LogicalAndExpression:F ⟧  =  Tr ⟦ E ⟧ && Tr ⟦ F ⟧

Tr ⟦ RelationExpression:E == RelationExpression:F ⟧  =  Tr ⟦ E ⟧ == Tr ⟦ F ⟧
Tr ⟦ RelationExpression:E != RelationExpression:F ⟧  =  Tr ⟦ E ⟧ != Tr ⟦ F ⟧
Tr ⟦ RelationExpression:E instanceof Type:T ⟧  =  isOfType(Tr ⟦ E ⟧, Type ⟦ T ⟧)

Tr ⟦ AdditiveExpression:E < AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ < Tr ⟦ F ⟧
Tr ⟦ AdditiveExpression:E <= AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ <= Tr ⟦ F ⟧
Tr ⟦ AdditiveExpression:E >= AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ >= Tr ⟦ F ⟧
Tr ⟦ AdditiveExpression:E > AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ > Tr ⟦ F ⟧
Tr ⟦ AdditiveExpression:E <:  AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ <:  Tr ⟦ F ⟧

Tr ⟦ MultExpression:E + AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ + Tr ⟦ F ⟧
Tr ⟦ MultExpression:E - AdditiveExpression:F ⟧  =  Tr ⟦ E ⟧ - Tr ⟦ F ⟧

Tr ⟦ UnaryExpression:E * MultExpression:F ⟧  =  Tr ⟦ E ⟧ * Tr ⟦ F ⟧
Tr ⟦ UnaryExpression:E / MultExpression:F ⟧  =  Tr ⟦ E ⟧ / Tr ⟦ F ⟧
Tr ⟦ UnaryExpression:E % MultExpression:F ⟧  =  Tr ⟦ E ⟧ % Tr ⟦ F ⟧

Tr ⟦ - UnaryExpression:E ⟧  =  - Tr ⟦ E ⟧
Tr ⟦ !  UnaryExpression:E ⟧  =  !  Tr ⟦ E ⟧
```

$Tr[\![$ (*Type*) *UnaryExpression:E* $]\!]$ = $Tr[\![$ *E* $]\!]$

The special fields `inv`, `commit` and `length` are translated using the designated functions or heap fields.

$Tr[\![$ *DereferenceExpression:E* `.inv` $]\!]$	=	*NameHeap*[$Tr[\![$ *E* $]\!]$, inv]	
$Tr[\![$ *DereferenceExpression:E* `.commit` $]\!]$	=	*BoolHeap*[$Tr[\![$ *E* $]\!]$, commit]	
$Tr[\![$ *DereferenceExpression:E* `.length` $]\!]$	=	*length*($Tr[\![$ *E* $]\!]$)	(only for arrays)

Let C denote the type of the *DereferenceExpression:E*. Regular object access can be done by looking up the value of the designated heap at index [$Tr[\![$ *E* $]\!]$, *C.I*].

$Tr[\![$ *DereferenceExpression:E* . *Ident:I* $]\!]$ =
 # if type of I **is boolean**
    BoolHeap[$Tr[\![$ *E* $]\!]$, $C.I$]
 # if type of I **is int**
    IntHeap[$Tr[\![$ *E* $]\!]$, $C.I$]
 # else
    ObjectHeap[$Tr[\![$ *E* $]\!]$, $C.I$]
 # end if

Arrays can be accessed using the appropriate selection function and the *ElemementsHeap*.

$Tr[\![$ *DereferenceExpression:E* [*Expression:F*] $]\!]$ =
 # elements of array E **are of type boolean**
    boolElementSelect(*ElemementsHeap*[$Tr[\![$ *E* $]\!]$], $Tr[\![$ *F* $]\!]$)
 # elements of array E **are of type int**
    intElementSelect(*ElemementsHeap*[$Tr[\![$ *E* $]\!]$], $Tr[\![$ *F* $]\!]$)
 # else
    refElementSelect(*ElemementsHeap*[$Tr[\![$ *E* $]\!]$], $Tr[\![$ *F* $]\!]$)
 # end if

Next, the translation rules for literals, constants and parenthesized expressions.

$Tr[\![$ `inv` $]\!]$	=	*NameHeap*[this, inv]	
$Tr[\![$ `commited` $]\!]$	=	*BoolHeap*[this, commit]	
$Tr[\![$ *Ident:I* $]\!]$	=	I	
$Tr[\![$ *Literal:L* $]\!]$	=	L	(only integers)
$Tr[\![$ `super` $]\!]$	=	∅	(handled in function calls directly)
$Tr[\![$ `true` $]\!]$	=	`true`	
$Tr[\![$ `false` $]\!]$	=	`false`	
$Tr[\![$ `this` $]\!]$	=	`this`	
$Tr[\![$ `null` $]\!]$	=	`null`	
$Tr[\![$ (*Expression:E*) $]\!]$	=	($Tr[\![$ *E* $]\!]$)	

`\old()` can be translated using the built-in function `old()`, `\typeof()` corresponds to function *typeOf* that we have introduced earlier on. The result of a procedure can be referenced using it's unique name N`.return` where N is the name of the method. `\ownerobject()` and `\ownertype()` are translated using the two functions *ownerObject* and *ownerType*.

$$\begin{aligned}
Tr\llbracket\ \texttt{\textbackslash old(}\ Expression{:}E\ \texttt{)}\ \rrbracket &= \mathit{old}(Tr\llbracket\ E\ \rrbracket)\\
Tr\llbracket\ \texttt{\textbackslash typeof(}\ Expression{:}E\ \texttt{)}\ \rrbracket &= \mathit{typeOf}(Tr\llbracket\ E\ \rrbracket)\\
Tr\llbracket\ \texttt{\textbackslash type(}\ Type{:}T\ \texttt{)}\ \rrbracket &= Type\llbracket\ T\ \rrbracket\\
Tr\llbracket\ \texttt{\textbackslash result}\ \rrbracket &= N\texttt{.return}\quad (N \text{ is name of the method})\\
Tr\llbracket\ \texttt{\textbackslash one}\ \rrbracket &= \emptyset\\
Tr\llbracket\ \texttt{\textbackslash ownerobject(}\ Expression{:}E\ \texttt{)}\ \rrbracket &= \mathit{ownerObject}(Tr\llbracket\ E\ \rrbracket)\\
Tr\llbracket\ \texttt{\textbackslash ownertype(}\ Expression{:}E\ \texttt{)}\ \rrbracket &= \mathit{ownerType}(Tr\llbracket\ E\ \rrbracket)
\end{aligned}$$

3.6.1 Checking for Definitions

Sometimes we need to make sure that a given expression is defined, i.e. we need to check that there are no array accesses with a bad index, no divisions by zero, no illegal cast, etc. The following translation rules do this checking. They are implemented by the visitor class `DefinitionChecker` which is part of the package `org.jtobpl.translation`.

$$\begin{aligned}
DefCk\llbracket\ ImpliesExpression{:}E\ \ EquivalenceOp\ \ ImpliesExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\[1ex]
DefCk\llbracket\ LogicalOrExpression{:}E\ \texttt{==>}\ LogicalOrExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ E\ \rrbracket\ \texttt{==>}\ DefCk\llbracket\ F\ \rrbracket)\\
DefCk\llbracket\ LogicalOrExpression{:}E\ \texttt{<==}\ LogicalOrExpression{:}F\ \rrbracket &= DefCk\llbracket\ F\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ F\ \rrbracket\ \texttt{==>}\ DefCk\llbracket\ E\ \rrbracket)\\[1ex]
DefCk\llbracket\ LogicalAndExpression{:}E\ \texttt{||}\ LogicalOrExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ E\ \rrbracket\ \texttt{||}\ DefCk\llbracket\ F\ \rrbracket)\\[1ex]
DefCk\llbracket\ EqualityExpression{:}E\ \texttt{\&\&}\ LogicalAndExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ E\ \rrbracket\ \texttt{==>}\ DefCk\llbracket\ F\ \rrbracket)\\[1ex]
DefCk\llbracket\ RelationExpression{:}E\ \ EqualityOp\ \ RelationExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\
DefCk\llbracket\ RelationExpression{:}E\ \texttt{instanceof}\ Type{:}T\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\\[1ex]
DefCk\llbracket\ AdditiveExpression{:}E\ \ RelationOp\ \ AdditiveExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\[1ex]
DefCk\llbracket\ MultExpression{:}E\ (\ \texttt{+}\mid\texttt{-}\ )\ AdditiveExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\[1ex]
DefCk\llbracket\ UnaryExpression{:}E\ \texttt{*}\ MultExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\
DefCk\llbracket\ UnaryExpression{:}E\ (\ \texttt{/}\mid\texttt{\%}\ )\ MultExpression{:}F\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ F\ \rrbracket\ \texttt{!= 0})\\[1ex]
DefCk\llbracket\ (\ \texttt{-}\mid\texttt{!}\ )\ \ UnaryExpression{:}E\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\\[1ex]
DefCk\llbracket\ (Type{:}T)\ UnaryExpression{:}E\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (\ \mathit{isOfType}(Tr\llbracket\ E\ \rrbracket,\ T)\ )\\[1ex]
DefCk\llbracket\ DereferenceExpression{:}E\ \texttt{.}\ Ident{:}I\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ E\ \rrbracket\ \texttt{!= null}\ )\\
DefCk\llbracket\ DereferenceExpression{:}E\ \texttt{[}\ Expression{:}F\ \texttt{]}\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket\ \texttt{\&\&}\ (Tr\llbracket\ E\ \rrbracket\ \texttt{!= null}\ )\ \texttt{\&\&}\ DefCk\llbracket\ F\ \rrbracket\\
&\quad \texttt{\&\&}\ (\ \texttt{0 <=}\ Tr\llbracket\ F\ \rrbracket\ )\ \texttt{\&\&}\ (\ Tr\llbracket\ F\ \rrbracket\ \texttt{<}\ \mathit{length}(Tr\llbracket\ E\ \rrbracket)\ )\\[1ex]
DefCk\llbracket\ Ident\mid Literal\ \rrbracket &= \texttt{true}\\
DefCk\llbracket\ \texttt{super}\mid\texttt{true}\mid\texttt{false}\mid\texttt{this}\mid\texttt{null}\ \rrbracket &= \texttt{true}\\
DefCk\llbracket\ (\ Expression{:}E\ )\ \rrbracket &= DefCk\llbracket\ E\ \rrbracket
\end{aligned}$$

$$
\begin{aligned}
\mathit{DefCk}\,[\![\ \texttt{\textbackslash old(}\ \mathit{Expression{:}E}\ \texttt{)}\ ]\!] &= \texttt{old}(\mathit{DefCk}\,[\![\ E\ ]\!]) \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash typeof(}\ \mathit{Expression{:}E}\ \texttt{)}\ ]\!] &= \mathit{DefCk}\,[\![\ E\ ]\!] \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash type(}\ \mathit{Type}\ \texttt{)}\ ]\!] &= \texttt{true} \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash result}\ ]\!] &= \texttt{true} \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash one}\ ]\!] &= \texttt{true} \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash ownerobject(}\ \mathit{Expression{:}E}\ \texttt{)}\ ]\!] &= \mathit{DefCk}\,[\![\ E\ ]\!] \\
\mathit{DefCk}\,[\![\ \texttt{\textbackslash ownertype(}\ \mathit{Expression{:}E}\ \texttt{)}\ ]\!] &= \mathit{DefCk}\,[\![\ E\ ]\!]
\end{aligned}
$$

4 Implementation

4.1 Structure of *JtoBPL*

JtoBPL is programmed in *Java* as an extension to the *MultiJava* [5] and *Jml* [4] framework. Both packages can be downloaded from the corresponding websites and are free to use under *GNU General Public License*. First we extended and changed the grammar of the *Jml* parser using the *ANTLR* tool. We formulated our own grammar files as an extension to the existing ones. The `*.g` files can be found in the directory `org/jtobpl/parser`. However, we didn't change the whole parser but rather extended it slightly where it was needed.

An Example: Since we had to extend some classes to store some additional values, such as helper variables needed by the translation (e.g. BVariableDefinition <: JmlVariableDefinition), we let our parser create instances of this subclasses instead of the original *Jml* classes. All newly introduced abstract syntax tree node classes can be found in package `org.jtobpl.ast`.

Another Example: Because we wanted to include the pack and unpack statement we had to change the parser's production for statements and include two new productions for parsing pack and unpack.

Almost the entire *Jml* grammar is still accepted by the parser. Our parser can still parse a new-object expression as an actual method parameter for example. But after parsing we walk through the entire abstract syntax tree that we get from the parser and make sure that such code is not accepted. This is done by the visitor class `Inspector`in package `org.jtobpl.translation`. Besides checking the structure of the syntax tree the `Inspector` also introduces local helper variables where they are needed for the translation, creates dummy classes for classes that are not overridden and much more.

Finally, if the `Inspector` doesn't encounter any errors we start the translation which is done by the visitor class `Translator` which is also part of package `org.jtobpl.translation`.

4.2 How to get the Software

The entire source code of *JtoBPL* can be downloaded from `http://n.ethz.ch/student/burrisa/jtobpl/`.

If one is only interested in using the tool the provided Jar-files might be enough. Otherwise it's possible to download all the source files and also the make files to create the different parsers and token files. Since the sources of *MultiJava* and *Jml*are also evolving over time the latest sources that we have used from this projects is also available.

4.3 Usage

As described above, one can use *JtoBPL* as if it was *Jml* but only sources that are supported according to our language subset may be translated without errors. If the Jar-files are used, the following files have to be provided: `MultiJava.jar`, `Jml.jar` and `JtoBPL.jar`. in a subfolder called `utils` one should place the Jar-files `ant.jar`, `antlr-2.7.5.jar`, `java-getopt-1.0.11.jar`, `junit.jar` and `tools.jar`.

Now *JtoBPL* can be invoked as follows:

```
[AluOSX:BoogieSemesterarbeit/Completion/JtoBPLJars] saem% java -jar JtoBPL.jar Test.java
```

... and the translation can be found in file `BoogiePLTranslation.bpl` in the same directory.

If one wants to use the fields `inv` and `commit` then the *Jml* file `Object.jml` should be used. It can be found in folder `specs/java/lang` and should be compiled together with the other source files. E.g:

```
[AluOSX:BoogieSemesterarbeit/Completion/JtoBPLJars] saem% java -jar JtoBPL.jar -S ./specs/ InvTest.java
```

Option `-S` indicates that alternative API classes can be found in the provided directory.

References

[1] Robert DeLine, K. Rustan M. Leino:
BoogiePL: A typed procedural language for checking object-oriented programs.
Technical Report MSR-TR-2005-70, 2005.

[2] K. Rustan M. Leino:
BoogieOOL: An object-oriented language for program verification.
Manuscript KRML 130, 2003.

[3] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Joseph Kiniry:
JML Reference Manual.
Draft, Revision 1.132, 2005.

[4] *The Java Modeling Language (JML) Website*.
http://www.cs.iastate.edu/ leavens/JML/.

[5] *The MultiJava Project Website*.
http://multijava.sourceforge.net/.

[6] K. Rustan M. Leino, Peter Müller:
Object Invariants in Dynamic Contexts.
ECOOP 2004, LNCS vol. 3086, Springer, 2004.

[7] Mike Barnett, K. Rustan M. Leino, Wolfram Schulte:
The Spec# Programming System: An Overview.
Manuscript KRML 136, 2004.

[8] *Microsoft Spec# Website*.
http://research.microsoft.com/specsharp/.

[9] *ANTLR Website*.
http://www.antlr.org/.