Master Project

Design and Implementation of a JML frontend to Boogie

Erich Laube
laubee@student.ethz.ch

March 2006

Software Component Technology Group
ETH Zurich
Switzerland

Prof. Peter Müller
Supervisor: Ádám Darvas

At Microsoft Research a tool is being developed to do automatic program verification on annotated C# programs. In this project the open interface to that tool is used to do verification on Java programs specified with JML annotations. The result is a compiler supporting ownership based invariants, Universes and modifies clauses to mention the most important achievements. Verification is done in a modular way, which gives the advantage that a verified module can be used in an arbitrary context without losing the verified properties.

Contents

1. Introduction

1.1. Project Goals

In this document I describe how I designed and implemented a selection of *JML* (Java Modeling Language) features [11] for the JML to *BoogiePL* [6] compiler. At ETH there is research being done on verification of software. The general approach for verification is that a program consists of two parts, one is the actual program code, in our case written in Java, which is translated into a binary (or byte code) by a compiler. The other part is a specification of the code, which states properties about what a certain piece of code is supposed to do. A specification describes the behavior of a program, stating what a program does without going into implementation details.

For instance, a specification to a container would include there being an *add* method with which one can put elements into the container and a *retrieve* method to get the elements back. It would say nothing about the actual implementation of the container - it does not matter if the container's store is a linked list or an array - the semantics would be the same.

Having the two parts - program code and specification - leads to the interesting opportunity to being able to check whether they match or not. This is what is meant by verification in our context. To see whether an implementation fulfills its specification. Being able to do this verification should allow one to produce code that can be trusted and reused by others, raising both quality and productiveness of the software industry. The whole approach is known as design by contract and described in a more detail here [13].

In this project, the focus lies on verification of JML [10] programs. JML offers a rich set of constructs to express the behavior of Java modules. For this project a set of JML constructs were selected to be investigated more closely and to be supported by the verification system.

For the actual verification part the Boogie tool was used. Boogie is a powerful automatic program verifier developed by Microsoft Research - more on Boogie in section 1.2.

This project is based on a previous semester project at ETH Zurich by Samuel Burri [4], which in turn was based mostly on [12]. In his work, he built a Java frontend to Boogie, supporting a very limited subset of the Java language. The semantics of his translations was based very much on the Boogie methodology [1]. In this project, JML semantics is

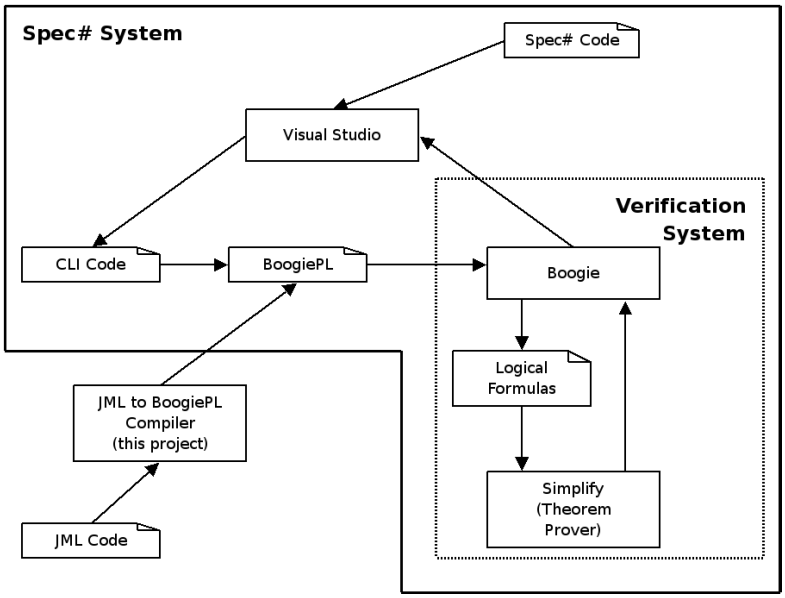

Figure 1.1.: Spec# System

used which leads to many differences, as will be pointed out and explained in detail in the following chapters.

The Java to BoogiePL compiler is built on the official JML compiler, which can be found here [18]. When the semester project started, version *5.2_rc2* was the current one and there was no update to an earlier version since then.

1.2. Spec#, Boogie and Simplify

The Spec# Programming System [3] consists of several components of which the programmer is only supposed to see Visual Studio. That IDE serves as the interface to a programmer, where code is written and feedback is given back by the compiler as well as the verification system. See Figure 1.1 for an illustration on how the components work together.

The source language of the Spec# system is C#, enhanced with specification elements such as pre-/and postconditions (hence the name, Spec#). The current version of Spec# can be found here [19].

In the background, Spec# code is first translated to *CLI* (Common Language Interface) bytecode and then to BoogiePL [6] code. This code is fed to the Boogie tool, which first does a weakest precondition calculus [2] before the code is translated to logical formulas which are the input for the last tool in the chain, *Simplify* [7].

Simplify is an automatic proving system, which in the end does the verification job on the created logical formulas. Its result is then given back to Boogie which interacts directly with Visual Studio and if the verification worked, the programmer will see red curly lines marking the parts of his code which might cause the program to fail.

The whole Spec# system is still a work in progress and many things keep changing. There is also very little documentation available. This means that usually to find out how they did solve an issue at Microsoft, one has to examine the BoogiePL code - which usually looks quite messy and keeps the core ideas well obfuscated. The target of the generated BoogiePL code is a machine after all. Thus, while the core way of how things are handled is similar in this project's compiler to the Spec# compiler, there are variations in detail. For more advanced features, which are described later in this document, semantics (and with that, implementation) are completely different. Also note that in this project, JML programs are directly translated to a file containing BoogiePL code. There is no intermediate code generated. The generated BoogiePL code is compatible with Boogie up to version *0.69*. At the end of the project Boogie version *0.70* came out which introduced some changes that rendered some of the generated axioms illegal. There was not enough time left to investigate this in detail and adapt the compiler.

1.3. BoogiePL introduction

In this section a little summary to BoogiePL is given to make it easier to understand the formulas in the following chapters.

BoogiePL is a procedural language, which is in a way similar to an assembler language. It offers support for most arithmetic operations and a special operator <: to express partial order on the name type. Loops and other conditional statements are covered by *goto* statements, which take an unlimited number of jump labels (i.e. *goto a, b, c;*). Boogie will attempt to verify all possible paths. Methods are encapsulated in procedure blocks, which can (but do not have to) have an implementation block. Procedures can also be given specifications, which is described below.

For the verification part, here is a list of the most important constructs: **function, axiom, assume, assert, requires, ensures and modifies**.

JML methods are translated to Boogie **procedures**. Contrary to procedures, **functions** never have an implementation and serve as logical predicates for verification.

Notation: for the rest of this paper, the term **function** refers to uninterpreted BoogiePL predicates, the term **method** will be used for JML methods and the term **procedure** for their translated version in the BoogiePL code.

Axioms are used outside procedure blocks to specify known facts and rules which hold

inside all procedures. For instance, they are used to specify the semantics of functions, statically known properties about the type system, relations of fields to classes, etc. Writing axioms giving semantics to functions is dangerous because the automatic verification system can and will use them even if they are unsafe. Or in other words, if there is an inconsistency that can be deduced from an axiom (or several axioms working together in unexpected ways), Boogie **will** find this inconsistency and produce a verification which cannot be trusted.

Assume statements are used inside procedure implementations. As the program part of Boogie is not as expressive as the Java language, semantic information can be put into a procedure with assume statements. The additional knowledge is used by the verification system without further tests and enable it to deduce facts that could not be expressed otherwise. For example, type information is assumed at the start of each method body, or the conditions of conditional statements are assumed. Needless to say that it is also important to be very careful with assume statements, as for instance after an *assume false*, Boogie will verify anything.

Assert statements force the verification system to verify a condition. If it does not hold, the verification fails. These are for instance used for checks like non-nullity before a method call.

Requires and **ensures** constructs are built-in features of Boogie. They give a public contract to callers of a procedure, which is verified by Boogie. At each procedure call the expression in the requires clauses have to be fulfilled and in each poststate of a procedure, the ensure clauses have to hold. Thus, on the other hand, at the beginning of each procedure block it is known that the requires clauses hold and a caller can be sure that that the procedure fulfills its postcondition.

In **modifies** clause, a contract for which locations a procedure is allowed to modify is given. In Boogie, contrary to JML, the semantics is such that if the modifies clause is left out, nothing may be modified.

BoogiePL offers primitive types covering boolean and integer values (**bool**, **int**). There is a type **name** which is used to map identifiers like class names and fields. There are only three operators defined on names, equality, disequality and the partial order <:. Complex object structures are modelled using a built-in **ref** type. There is support for arrays, which can be one - or two-dimensional.

A special two-dimensional array is the `$Heap`, which models the object store. It is accessed using a reference and a name. Access to fields of an object are modelled by accessing that heap array via the object's reference and the field's name. For example to access a field *f*, declared in class *A* on a reference *r* would look like this: `$Heap[r, A.f]`.

1.3.1. Mapping of JML Fields in BoogiePL

As in following formulas fields will be used heavily, here a little overview on how fields are implemented in BoogiePL. At the translation of class A, for every field f of type T declared in class A, the **name** $A.f$ is generated and reserved.

- const A.f: name;

On this name two basic axioms are defined:

- *axiom (`fieldHome(A.f)` == A);*
- *axiom (`fieldType(A.f)` == T);*

The `fieldHome` function expresses in which class field f has been declared. The other information that is axiomatized is the type of field f. Note that the axiom is written using the == operator, but in a program the actual object stored in field f might be an instance of a any subtype of T.

1.4. Working with Boogie

An important thing to note is that finding errors in a BoogiePL program is extremely tedious. There is no debugging interface to Boogie that I know of and it is not even possible to do primitive text output debugging. Boogie can be thought of as a black box system. An input program is fed to the black box and the output is a text message. For debugging, the only part that can be modified is the input program. Here are the four basic output options and how I went about debugging them:

Boogie terminates with the message that it found no errors.

This can be good - if the verified program indeed had no errors. It is sometimes the case though, that Boogie manages to verify a program that should not pass. This happens when the verification system manages to deduce *assume false* using a new axiom or just a new setup of axioms that was not considered before. Such errors happen easily and are difficult to track down. Debugging consists of two steps, first the location at which *assume false* was deduced has to be found.

This can be achieved by inserting *assert false;* statements in the code going upwards in a procedure line by line, until Boogie fails to verify the program. Once the line of code which causes the bad assumption is found, the next step is to find out why that expression leads to the unsoundness. For this an insight into Boogie's deduction mechanism would be nice, but is not available or legible. What is left is to take out axioms one by one stripping the program to a minimal size. As it can be that only several axioms playing together cause an unsoundness it can be hard to figure out which these are and which axiom exactly causes problems.

Boogie terminates with the message that an assertion inside a program or a pre-/postcondition was violated and names the corresponding line of code.

Often, this is the expected output of a program verification. But sometimes Boogie fails to verify simple programs that should pass. There are many reasons why this can happen. Sometimes it is a procedure call, where the postcondition and modifies clauses are not strong enough to continue with verification of the current procedure as Boogie lacks knowledge of what stayed the same and what was modified. It can also be that some axioms are not strong enough yet. Finding the error again is not easy, moving around the assertion that fails might give an indication on what statement caused it to fail.

Boogie terminates with the message that it had to skip a procedure due to ,,unexpected prover output".

This error message is one of the worst as there is really nothing to be done about it. It can be assumed that something is substantially wrong about the program and that a problem might need to be tackled using a completely different approach.

Boogie does not terminate after a reasonable time.

There are some statements which are difficult to verify and consume a lot of CPU time - *forall* quantifications to name one of them. To deal with this problem, the only way is to rewrite the JML program such that it becomes easier to verify. Other approaches that sometimes help are to insert assert statements to the BoogiePL code which in some cases help Simplify to find better instantiations, but that is not really a good option, breaking the idea of fully automated verification in a way...

1.5. Document Layout

Chapters are built such that first a problem is described followed by the final solution. For those interested, there follows some words about how that solution was found, where pitfalls lied and how they were found/avoided. This part should be interesting especially for readers who intend to further work on the compiler or with Boogie in general - maybe it can spare them some days of bug searching.

Note that there is no chapter about the translation of Java to BoogiePL. Whenever details of this translation are needed to understand how certain issues were solved, I will briefly introduce and explain them. A brief overview of what is (not) supported by the compiler is given in Appendix A. For a complete overview, take a look at [4] or the language specification in Appendix B.

In chapter 2, an overview is given on what the most important changes to the semester

project were. It covers bugfixes as well as design decisions leading to different translations. In chapter 3, the Universe type system is introduced as a prerequisite to ownership invariants which are covered in chapter 4. A description on how modifies clauses are translated is given in chapter 5 which also includes a little excursion to some JML problems. In chapter 6 history constraints are introduced. In chapter 7 some smaller contributions to the compiler are listed and explained. Lastly, chapter 8 concludes and gives a summary of this project.

2. Differences and Bugfixes to Previous Work

In this chapter I will show the main changes and bugfixes applied to Samuel's compiler. Especially the change regarding the heaps required a major overhaul of the translator.

2.1. Reduction from five Heaps to one Heap

In Samuel's work, there were five heaps to map class fields into. There were heaps for integer values, booleans, references, names and one for array elements. For some new predicates it was needed to have heap parameters - having five of those not only meant an overhead but also made the BoogiePL code much harder to read. While Boogie does not care about readability of the code, a user of the tool needs to look at the BoogiePL code to find out what a Boogie error message meant as there is no backlink to an editor yet.

The reduction was from:

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

to

```
var $Heap :  [ref, name]any;
```

In the new heap elements of all types can be stored - its return type is **any**. As can be seen, the first four heaps can still be accessed using a `[ref, name]` tuple. For the elements array, a new constant was inserted:

```
const $array$:  name;
```

Using the new constant as dummy placeholder for the **name** argument, access to the heap is now uniform.

Updating the heap is just an assignment to the referred location. However, every read

access to the heap now has to be cast to the correct BoogiePL type. Thus a read access to the *$Heap* at location *[r, n]* of type T is now translated to:

```
cast($Heap[r, n], T)
```

where T can be any of the five types mentioned before. These casts of course also clutter the code again and make it harder to read. But in my opinion the casts are to be preferred over an option with five heaps and non-uniform access. Note that for all read accesses the resulting BoogiePL type T of the cast is statically known. The casts only affect the BoogiePL part and not the Java part of the language. For all Java reference types it cannot be known statically if a Java type cast will succeed or fail, but on the BoogiePL level these are all mapped to type **ref** anyway.

2.2. Type System Translation

When I took over the compiler from Samuel, I found the whole mapping of the type system to be very confusing. There were two functions related to types:

```
isOfType
typeOf
```

isOfType is related to *typeOf* as follows:

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

The main difference between the two is that *isOfType* covers the option that a reference might be **null**. In other words, *isOfType*`(r, T)` means r is either **null** or it is a subtype of T. While looking up BoogiePL code after reading error output of Boogie it often occured to me that remembering the coverage of **null** values was not that obvious. For simplification I decided to remove the *isOfType* function.

Modelling everything with one construct bore its own pitfalls though. With *typeOf* it is now obvious in the code to see what is assumed but the guard for null values has to be written explicitely. Assuming types on (potential) **null** references leads to unsoundness and enables Boogie to verify literally anything.

2.3. Inheritance

Inheritance was implemented by copying all fields of a super class and inserting method stubs for method calls. This worked as long as no casts were inserted, but was really problematic as the following example shows:

```
class A{
  int a;
}

class B extends A{
  int b;
}
```

On the BoogiePL level the following names were created:

```
A.a
B.a (inherited)
B.b
```

This was problematic as soon as a client used code of the following form:

```
B b = new B();
b.a = 5;                  // Heap[b, B.a] := 5;
A a = (A) b;
a.a == ?;                 // not defined, as Heap[a, A.a] does not refer
                          // to Heap[b, B.a]
```

This buggy behavior was corrected by removing all the copying of fields and methods - it simply is not needed. As subtyping already was implemented using the partial order operator one could actually pass a `B` reference as the **this** reference to a method declared in `A`. It is also possible to assign directly to the fields declared in a superclass:

```
b.a = 5;                  // Heap[b, A.a] := 5;
```

One important thing to note here is that usually one cannot know whether the **this** object is actually of the type of the class a method is declared in, or an instance of a subclass thereof.

2.4. If-Then-Else translation

Translation of if-then-else statements has earned itself a full section just dedicated to that statement. Reason for this is that I actually had to fix it twice to get it fully functional and writing down the example might help getting some insight into how verification works and how information has to be given to the Boogie system at the right locations.

In an if-then-else statement, a condition c is evaluated and depending on its boolean value the control flow is directed to a different path. In BoogiePL there are no conditional statements, so the if-then-else statement needs to be mapped to BoogiePL constructs. The statement to direct control flow in different directions is the **goto** statement, which takes a number of labels. When mapping an if-then-else statement to a **goto** statement, the condition thus is not evaluated first, but actually assumed depending on which label was taken.

The labels defined for an if-then-else statement are *then*, *else* and *join*. The first version was pretty bug infected, there was no **goto** to the *else* part and it had problems with statements returning on both the *if* and the *else* part, as it left a dangling *join* label, which would not even pass the Boogie parser. These were all fixed rather easily but the short if statement was still causing problems.

The short if statement, without else, in its second version had a *goto then, join;*. The control flow either goes directly to the *join* label if condition *c* is not met or it goes to the *then* label, where *c* is assumed. The complete translation looked as follows:

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:
```

The translation above worked fine for some examples, but at one point I had an example similar to:

```
class C{
  int i=0;
  //@ ensures i == 0;
  void reset_i (){
    if(i != 0){
      i = 0;
    }
  }
}
```

Running the example through Boogie led to the error statement that method `reset_i` did not satisfy its postcondition. Reason for this behavior was that with the translation above, there only is an assumption on condition *c* if the control flow goes through the *then* statement. If the *c* does not hold, the flow goes directly to the *join* statement. Which in the Java world is of course right, but in Boogie, where the *c* is not actually evaluated, but assumed, leads to the problem of there being no assumption on *c* for the *else* path. Fixing this bug was obviously easy - the short if statement had to be expanded to an if statement with an empty else clause, where in the BoogiePL translation, *!c* could be assumed, followed by the *goto* statement to the *join* label:

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

```
assert DefCk [[ C ]];
# generate symbol then, else, join
goto then, else;
then:
   assume Tr [[ C ]];
   Tr [[ T ]];
   goto join;
else:
   assume Tr [[ !C ]];
   goto join;
join:
```

What I intend to show with this example is not how one can put stupid errors into a compiler and how easily they can be fixed. It is rather that a Java program is executed in a different way than a BoogiePL program is verified. Being familiar with Java leads to a way of thinking which is not always applicable to a BoogiePL program - it may even be dangerous to do so. Errors that sneak into a compiler like this are - albeit easy to fix - very hard to find and track down.

2.5. New Axioms and Functions

For the *length* function a new axiom was introduced to denote that the length of an array cannot be negative:

```
axiom (forall r:  ref ::  length(r) >= 0);
```

For all methods, the precondition that the **this** parameter may not be **null** was added.

Functions to distinguish different usages of the type **name** were added. These are *$isTypeName*, *$isFieldName* and *$isArrayFieldName*. Axioms are generated to give meanings to those functions.

Two new functions were added to distinguish subtyping paths. If one had types *A, B, C* and *D*, where both *B* and *C* were subtypes of *A* and *D* a subtype of *B* Boogie was unable to deduce that a subtype of *D* could not be a subtype of *C*. *$asDirectSubClass* is used to state the relations between *B* and *A*, *C* and A as well as *D* and *B*. For the example, the following axioms would be generated:

```
axiom $asDirectSubClass(A, Object) == A
axiom $asDirectSubClass(B, A) == B
axiom $asDirectSubClass(C, A) == C
axiom $asDirectSubClass(D, B) == D
```

The *$oneClassDown* function is defined as follows:

```
axiom (forall X: name, Y: name, Z: name ::
  Z <: $asDirectSubClass(Y, X)
  ==> $oneClassDown(Z, X) == Y;
```

The two functions thus rule out the possibility that between the subclass relation of A and B an additional type T could exist.

3. Universe Type System

A first addition to the compiler was the implementation of a mapping of the Universe type system (UTS) [15] to BoogiePL. The UTS gives static information and guarantees about ownership and aliasing in a JML program. This information can be used by the verification system to prove certain properties and is a prerequisite to ownership invariant encoding.

3.1. Introduction to Universe Type System

In verification, objects are often layered in an ownership hierarchy. This means that every object is owned by another object. Usually these owners are the only ones who may modify the owned objects. The layering of objects also offers a means to control aliasing between objcects. In general, a reference that points to an object owned by the this-object cannot be assigned an object owned by a different object. There are many variations of ownerships, for those interested [5, 17] offer more material on the topic aside from the UTS papers [15, 8]. For this project the UTS was chosen as ownership model.

In the UTS ownership is organized such that there are different contexts. Each context has its owner object. In general, there are only references allowed which point to objects in the same context or to an object in the directly owned context. There is an exception to this, readonly references, which may point anywhere, but on such a reference, only pure methods may be executed and its attributes cannot be written, it is thus not possible to alter the state of an object using a readonly reference. An illustration on how ownership contexts look like in the UTS is given in Figure 3.1.

The UTS is implemented in the JML compiler [8]. References can be annotated using one of the following three keywords:

- **peer**
  A **peer** reference denotes a reference pointing to an object allocated in the same context, thus, an object having the same owner as the this-object.
- **rep**
  Objects owned by the this-object are denoted with a **rep** reference.
- **readonly**
  A **readonly** reference may point to an arbitrary object in an arbitrary context.

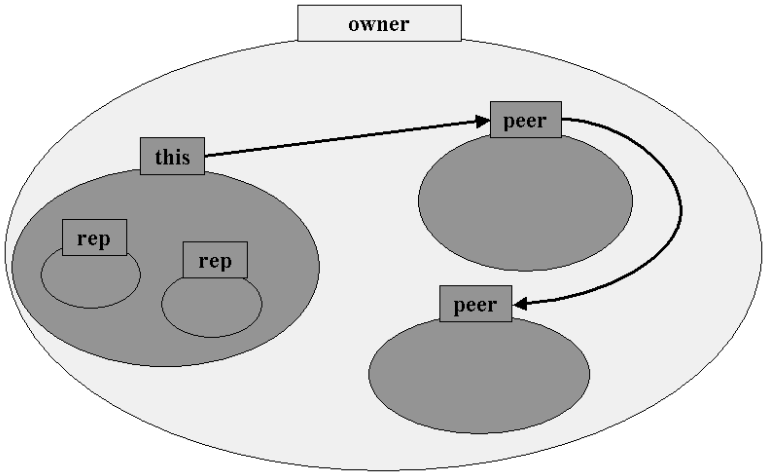

Figure 3.1.: Universe Contexts

The default annotation for a reference is **peer**. This allows that every Java program without UTS annotations can pass the UTS-enabled compiler. Using the UTS enables programmers to encapsulate certain parts of their programs and have guarantees for these parts, which helps reasoning about the correctness of a program.

It is important to note that arrays actually need two UTS modifiers. One is the actual array reference and one for the elements refered to by the array. A **rep peer** array would thus be an array owned by the this-object. As its elements are **peer**, they also refer to objects in the context owned by the this-object.

When a new object is created, its Universe needs to be given. If it is not specified, the Universe of the new object defaults to peer.

Formalized, the UTS gives the following guarantees:

Universe Invariant. In every execution state, the following invariant holds: If an object X holds a direct reference to object Y then either:

1. X and Y belong to the same universe
2. X is the owner of Y or
3. the reference is readonly

3.2. Mapping from JML to BoogiePL

A first step to enable the compiler to use the static guarantees given by the UTS was to map the ownership information into BoogiePL code. The notion of ownership is mapped

by using a predicate called *ownerObject*. It takes a reference parameter and returns a reference to the owner object. Taking the UTS information from the JML compiler, ownership can be deduced and rewritten to BoogiePL syntax. For all references appearing in a program, the ownership information has to be expressed using the *ownerObject* predicate. There are three cases to distinguish:

- **fields**
  For each instance of a class, there will always be the same fields. As fields are mapped using axioms, the respective ownership information is also axiomatized.
- **parameters**
  Ownership information of parameters is part of the contract of a method. They thus are written into the precondition of BoogiePL procedures. They, too are expressed in relation to the this reference of a procedure.
- **local variables**
  For local variables, ownership is assumed at the beginning of a method, with respect to the *this* object. All local variables are only looked at from this perspective. A peer variable will have the same owner as *this* and a rep variable will have *this* as owner.

Note that for **readonly** locations, there is no static information that could be translated, but there is the possibility to give ownership information in the specifications of a method or a class.

A special case are array elements. In the UTS arrays can hold either **readonly** or **peer** references. **Rep** references are not allowed because they would be owned by the array reference and would thus be useless. There is no type information for **readonly** elements, but for **peer** elements the information has to be passed on to Boogie. The resulting formula seems a bit complicated but it is basically just two times a quantification over all references, where the first reference has to be the actual array and the second reference are the elements.

3.2.1. Example

Consider a class C with method m and **peer**, **rep** and **readonly** fields, variables and parameters. The following axioms, assumptions and requires clauses need to be generated:

```
class C{
  peer C peerF;
  rep C repF;
```

```
  readonly C roF;
  peer peer C[] ppF;
  void m(
      peer C peerP,
      rep C repP,
      readonly C roP,
      peer peer C[] ppP
    ){
      peer C peerV;
      rep C repV;
      readonly C roV;
      peer peer C[] ppV;
  }
}
```

```
  axiom (forall r: ref :: (r != null && typeOf(r) <: C)
     ==> ownerObject(cast ($Heap[r, C.peerF], ref)) == ownerObject(r));

  axiom (forall r: ref :: (r != null && typeOf(r) <: C)
     ==> ownerObject(cast ($Heap[r, C.repF], ref)) == r);

  axiom (forall r: ref, e: elements, c: ref, item: ref, i: int ::
     ( c != null && r != null && item != null
     && r == cast ($Heap[c, C.ppF], ref)
     && e == cast ($Heap[r, $array$], elements)
     && item == refElementSelect(e, i)
     ==> ownerObject(item) == ownerObject(c)));


  procedure C.m (this: ref);
  requires peerP == null || ownerObject(peerP) == ownerObject(this);
  requires repP == null || ownerObject(repP) == this;
  requires (forall e: elements, c: ref, item: ref, i: int ::
     ( c != null && r != null && item != null
     && r == cast ($Heap[c, C.ppF], ref)
     && e == cast ($Heap[r, $array$], elements)
     && item == refElementSelect(e, i)
     ==> ownerObject(item) == ownerObject(c)));

  implementation C.m (this: ref) {
  assume peerV == null || ownerObject(peerV) == ownerObject(this);
  assume repV == null || ownerObject(repV) == this;
  }
```

3.3. Formalization of UTS Semantics

After UTS information is formalized using the BoogiePL *ownerObject* function, some properties of the UTS need to be stated in axioms to actually give a meaning to the *ownerObject* predicate. One such property is that an object cannot be the owner of itself:

```
axiom (forall r:  ref ::  (r != null)
    ==> r != ownerObject(r));
```

Another important concept that needs to be formalized is transitive ownership. With the *ownerObject* predicate one can only go one layer deeper in an object structure. But objects of a lower layer again have their own context. If one wants to express all objects inside a context Γ, a new function is needed. The *isTOwnedBy* function is defined such that it takes two ref parameters, iff the first parameter resides inside the context owned by the second parameter the function returns true, else false. Here the axiom defining this behavior:

```
axiom (forall owned:  ref, owner:  ref ::
(owned != null && owner != null) ==>
   (isTOwnedBy(owned, owner)
   ==>
   ownerObject(owned) == owner
   || isTOwnedBy(ownerObject(owned), owner)));
```

Unfortunately, Simplify cannot handle recursive functions properly [9]. While the formula is correct and will not raise any errors, Simplify just cannot use it properly during verification. There currently is no way of expressing transitive ownership for Boogie in a generic way.

A last property that would have been important to express was the acyclicity of ownership in the UTS. However, defining acyclicity depends on having a working definition for transitive ownership, so the designed axiom will not work:

```
axiom (forall r1:  ref, r2:  ref ::
   (r1 != null && r2 != null
   && isTOwnedBy(r1, r2))
      ==>
   !isTOwnedBy(r2, r1));
```

3.4. JML owner Field

In JML, the *Object* class is annotated with a ghost field called *owner*. To make use of that field, it needs to be treated specially during translation and mapped to ownership

assumption using the *ownerObject* predicate. This enables one to specify owners of readonly objects for instance. It is however to be used with caution, because one could specify a rep parameter to have a different owner o than *this*. This would lead to inconsistent assumptions in Boogie, as Boogie would go and deduce $o == this$, which will lead to unsoundness in the verification.

4. Ownership Invariants

4.1. Motivation

In [16] different approaches are shown as on how invariants can be verified in a modular way. The current, classical technique is analyzed and two new approaches are shown. One of those approaches is based on visibility and the other one on ownership. For this project I took a closer look at ownership based invariants and implemented that approach. What follows is a brief overview of classical invariants, followed by a summary of how the ownership based invariants work and what problems/issues they address and solve. Lastly comes the implementation for this project.

4.1.1. On Invariants

Invariants are predicates that specify what states of an object are consistent. This can for example be that certain fields are never null, or certain values will never be negative. Having such specifications for an object helps reasoning about program code, if a field is never null, calls on that field can never raise null pointer exceptions, if a value is never negative one can take the square root of it without worries, etc.

Verification of a generic invariant without restrictions is however very hard and bears the problem that a whole program would need to be looked at for a sound verification. Every new part added to a program would cause the whole verification process to be started anew, which is why it is interesting to look at modular invariants. The idea here is that one wants to verify a module, in our case a Java class. If the code inside that module does not break its invariant, the module is safe to use, meaning even if the module is used in an environment unknown at the verification time, the rules stated and verified about the module will still hold.

To achieve modularity, restrictions have to be imposed on invariants. An invariant technique must address four aspect:

- Encapsulation: What parts of a program can assign to the variables used in an invariant?
- Admissibility: What variables may an invariant depend on?
- Semantics: When do invariants have to hold?

- Modularity: How can one show that objects satisfy their invariants without examining the entire program?

For the definition of semantics, the term *visible state* is very important. An object is in a visible state whenever a method is called or terminated. Every pre- or poststate of a method execution is a visible state.

4.2. Classical Technique

In the classical technique, encapsulation is defined such that assignments to a location $X.f$ are only allowed if they occur in a method with X as receiver object. It is thus only allowed to do direct assignments on fields of the *this* object (including inherited fields), assignments in other objects have to be done via setters.

Classical admissibility only allows an invariant of class C to depend on a field declared in C or on a chain of field accesses, where all but the first field of the chain have to be constant. This means that array elements are not admissible, as the actual array is the field declared in C and array elements are already a step further, where only constants are allowed. Fields declared in a super class are not admissible either, because modification of a super class field in a super class method might destroy an invariant of a subclass. As we strive for modularity, a super class does not know of all the sub classes there might be.

For classical semantics invariants have to hold whenever an object is in a visible state.

Classical invariants are modular, to prove an invariant for class C one must show for each method m in C that:

1. the invariant of C and m's postcondition hold for the poststate of each execution of m
2. for every call of a method $\text{o.p}(\vec{e})$, that appears in the body of m, $\text{pre}(\text{o.p}(\vec{e}))$ holds in the call's prestate and the invariant of C also holds for m's receiver object in the prestate of the call to p.

One may assume that:

1. the invariants of all allocated objects and m's precondition hold in m's prestate.
2. after each call of a method, $\text{o.p}(\vec{e})$, $\text{post}(\text{o.p}(\vec{e}))$ holds and all allocated objects satisfy their invariants.

As pointed out in [16], the classical technique has limits in so far as it cannot describe interesting invariants for layered objects. The restrictions added to achieve modularity, described above are too severe for many interesting cases.

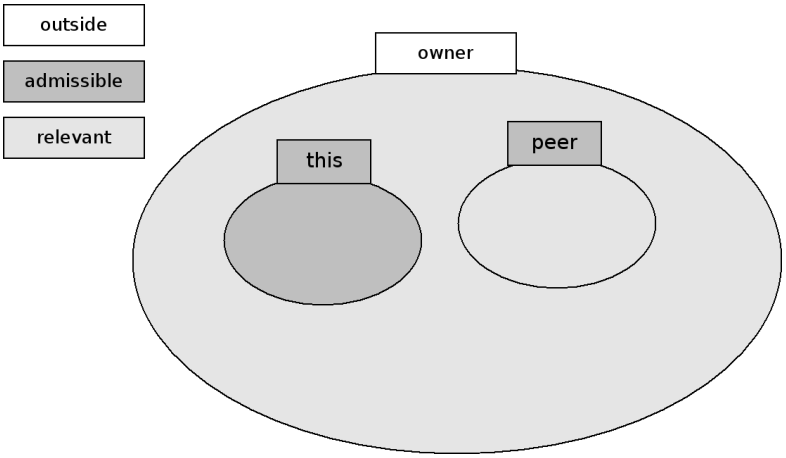

Figure 4.1.: Ownership Admissibility

4.3. Ownership Technique

Applying the guarantees given by an ownership structure enables one to loosen some restrictions of the classical technique. The UTS guarantees that objects from a context Γ cannot modify locations in a different context Γ'. This broadens the definition of encapsulation. A location $Y.f$ is ownership encapsulated in an object X if:

1. $X == Y$ and the only assignments to $X.f$ occur in method executions with X as receiver object or
2. X owns Y

In other words, encapsulated locations now are not only of the form *this.** but also cover *this.rep.**, where *rep* denotes a chain of field accesses going into the context owned by the this-object. Admissibility, too is enhanced by applying the UTS. **All locations over which we have control are admissible.** Or more formally, an access expression occurring in a class C is ownership admissible if it has one of the following forms:

1. A field name g_0, where g_0 is declared in C (e.g. this.f)
2. $g_0...g_N$, $N > 0$, where:
   - for each $0 <= i < N$, $g_0...g_i$ is ownership admissible and g_N is a constant field access
3. $g_0...g_N$, $N > 0$, where
   - the pivot field g_0 is declared with a **rep** annotation and
   - for each $0 <= i < N$, $g_0...g_i$ is ownership admissible and where

- each of the fields and array accesses in $g_1 ... g_{N-1}$ are declared with a **rep** or **peer** annotation

For ownership invariant semantics a new term, **relevant object**, has to be introduced. An object X is relevant to the execution of a method m iff X is inside the context in which m executes. In UTS terms these are all **peer** objects (including **this**) and all their **rep** objects. For the semantics, in a method execution m, all objects X relevant to the execution of m have to satisfy their invariants in the visible states of m's execution. An illustration of admissibility and relevant objects is shown in Figure 4.1 This leads to the following proof technique: for each method m declared in a class C one must show that:

1. The invariant of C and m's postcondition hold for the current receiver object in each poststate of m's execution
2. for every call of a method, o.p($\vec{e}$), that appears in the body of m, pre(o.p($\vec{e}$)) holds in the call's prestate and if o is an element of m's context, then the invariant of C also holds for m's receiver object in the prestate of the call to p
3. For each method m, and for each method call o.p($\vec{e}$), appearing in m, if o has a **readonly** type, then one must show that the object referenced by o is inside the context that contains m's **this** object

For the method execution one may assume:

1. the invariant of all allocated objects that are relevant to the execution of m and m's precondition hold in m's prestate
2. after each call of a method, o.p($\vec{e}$), post(o.p($\vec{e}$)) holds and all allocated objects that are relevant to the execution of o.p($\vec{e}$) satisfy their invariants

There are a few other restrictions for ownership based invariants:

- **rep** fields need to be **private** subsection 4.3.1
- assignments to fields of an object other than **this** has to be done via setters

Using ownership based invariants enables the programmer to write more expressive invariants. Restrictions imposed by the classical approach are lifted, but there are still limits to ownership invariants.

4.3.1. Subclass Separation

A special point to note is the issue of subclass separation. Consider code snippet in Figure 4.3.

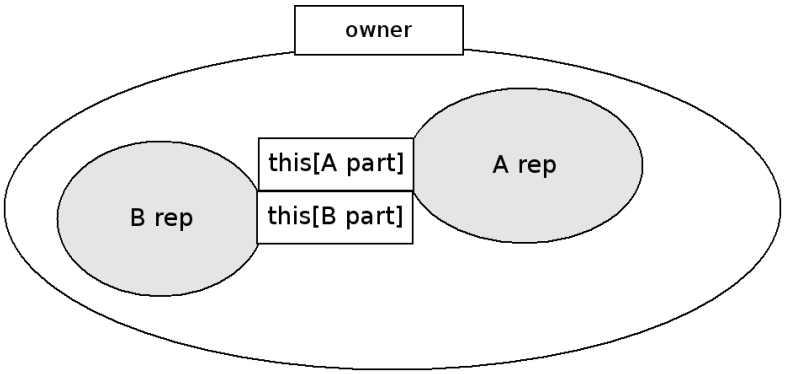

Figure 4.2.: subclass separation

```
class A{
  rep A arep;
  int i;
  //@ invariant arep != null;
  void init(){
        arep.i = 0;
  }

 //@ ensures \result.owner == this;
 readonly A leak(){
   return (readonly A) arep;
 }
}
class B extends A{
  rep A brep;
  //@ invariant brep != null && brep.i > 5;

  void oops(){
    readonly A bro = leak();
    //assert bro.owner == this;
    this.brep = (rep A) bro;
    init (); // invariant broken
  }
}
```

Figure 4.3.: subclass separation example

In this example, both class `A` and `B` have a **rep** reference to an `A` object. If `arep` and `brep` refer to the same object, method `init` in `A` might break the invariant of `B` without knowing so. To prohibit this potential unsoundness, in [16] the requirement that all fields of **rep** type need to be private was added. With that the guarantee that `arep` and `brep` never refer to the same object is given. This actually leads to a different than usual picture for Universe contexts, as shown in Figure 4.2. It can be seen that a subclass extending a superclass now owns more than one context, which are disjoint.

There is however still the potential problem that a **private rep** reference might leak out via a **readonly** reference, later be cast back to a **rep** reference and abused to break invariant of class `B`.

The example in Figure 4.3 shows the potential unsoundness. Method `leak` of class `A` returns a **readonly** reference of its **rep** object. Its specification states that the return value has the **this** object as its owner, thus is a **rep** field. In class `B`, method `oops` abuses the `leak` method by calling it and assigning the return value to the **rep** field of class `B`. After calling method `init`, the invariant of `B` is broken.

With the checks on Universes introduced so far (and shown in comments as assert statement), the broken invariant would not be noticed by the verification, because with the modular verification of class `A` and its methods, subclasses are not taken into consideration. To prevent the unsound reasoning, the *`ownerType`* predicate is used, which ties a reference to a specific type of owner on the BoogiePL level. In a cast from **readonly** to **rep**, not only the owner object, but also the type of the owner object has to be checked:

```
void oops(){
  readonly Object bro = leak();
  //assert bro.owner == this
  //     && ownerType(bro) == B;
  this.brep = (rep Object) bro;
  // assignment fails
}
```

With the checks given above, Boogie will complain about the cast above and tell the user that it is invalid. But what if one actually would want to specify a **readonly** reference to be a **rep** type?

The solution to this issue is given to the programmer by using the JML predicate *`ownerType`* originally introduced to the compiler by Samuel Burri in his semester project. With the predicate a specification like:

```
readonly Object o;
//@ invariant \ownerType(o) == \type(C)
//@             && o.owner == this;
```

can be given, specifying a **readonly** field `o` to be in fact a **rep** Object.

Note that for a **peer** reference, its owner type can be stated with respect to the **this** reference. As can be seen in Figure 4.2, all **peer** fields in a context have the same owner type. Thus, if for one object the owner type is known, it is also known for all its peers.

4.4. Solution

The implementation of ownership invariants is split into two parts. The first part can be statically proven within the compiler and a second part which is done by Boogie in the verification process. In the static part, admissibility is checked, in the verification part, the proof technique is applied.

4.4.1. Admissibility (JML)

At the same time as I was working on this project another student was implementing ownership invariants for the Jive tool and also implemented an admissibility checker for JML. His checker aimed at covering the complete subset of JML constructs and was not ready to use yet when I was working on ownership invariants. So I decided to implement my own admissibility checker, which only had to cover the JML subset supported so far by the compiler. Not having to deal with constructs such as model fields enabled me to implement a relatively simple algorithm.

The rules for admissibility described in the previos section can be characterized by regular expressions. Let G denote a field declared in C, r and p denote non-constant field or array accesses declared with a **rep** or a **peer** annotation, g be a non-constant field or array access and c a constant field access. The following regular expressions cover the corresponding rules from section 4.3:

1. G
2. $Gc*$
3. $r(r|p) * (c)$

Summarizing the three regular expressions one gets:

- $(G|r(r|p) * g?)c*$

The admissibility checker works as follows. When an expression is checked for admissibility, the accessed nodes are traversed and a string is built up, for each visited access expression one of the letters above is added. The resulting string is checked against the regular expression above. If it passes, the expression is admissible.

4.4.2. Invariants in BoogiePL

Invariants are mapped to BoogiePL by introducing two new functions, *inv* and *INV*. Both take as parameters a reference r, for which the invariant has to be shown and a heap parameter h, which referes to the global heap. They return true if the invariant is satisfied and false if it is not. The difference between the two is that *inv* also takes a type parameter, such that with *inv* one can query whether the invariant for a specific type is satisfied. The *INV* predicate conjoins the result of all possible invariants.

Formalization:

```
function inv(T: name, r:  ref, h:  [ref, name]any) returns (bool);
function INV(r:  ref, h:  [ref, name]any) returns (bool);
```

For each class C, the following axiom is generated to define the *inv* of C:

```
axiom (forall r:  ref, h:  [ref, name]any ::
   inv(C, r, h)
   <==>
   ((r != null) ==>
      ((typeOf(r) <:  C) ==>
         Tr [[ Inv_c ]])));
```

Note that there are two guards in the definition for a *inv* axiom. First, everytime a **null** reference is checked for invariants to hold, the function returns true. The second guard is for type. If a reference of type B which is not related to type A is checked for the invariant of A, the expression, too returns true. Inheritance of invariants is also guaranteed, as the *typeOf* predicate's condition is that reference `r` has to be a subtype of `C`. *INV* conjoins all *inv* predicates. Here the idea behind the names of the two functions can be seen, *INV* in capital letters is the sum of all *inv*s with small letters.

Axiom to conjoin all *inv*'s:

```
axiom (forall r:  ref, h:  [ref, name]any ::
   INV(r, h)
      <==>
      (forall t:  name ::  $isTypeName(t)
         ==> (inv(t, r, h))));
```

With these two functions and their definition, the actual invariant expressions are translated to BoogiePL functions. What is left to do is to apply the ownership proof technique. Taken from above, one needs to show for each method m declared in a class C:

1. *inv*(C) and m's postcondition hold for the current receiver object in each poststate of m's execution

2. for every call of a method, o.p($\vec{e}$), that appears in the body of m, pre(o.p($\vec{e}$)) holds in the call's prestate and if o is an element of m's context, then *inv*(C)(C) also holds for m's receiver object in the prestate of the call to p

3. For each method m, and for each method call o.p($\vec{e}$), appearing in m, if o has a **readonly** type, then one must show that the object referenced by o is inside the context that contains m's this object

Translated to BoogiePL, in addition to the pre-/postcondition part, which was already implemented, the following changes had to be implemented:

1. `INV(this, $Heap)` added to the postcondition of every method

2. before a call to a method p executed on a **peer** object

   ```
   assert INV(this, $Heap);
   ```

   is added

3. before a call to a method p executed on a **readonly** object o

   ```
   assert(isTOwnedBy(o, ownerObject(this)));
   ```

   and for the case that o is actually a **peer** object

   ```
   assert( ownerObject(this) == ownerObject(o)
   ==> INV(this, $Heap));
   ```

   has to be shown.

For a method execution one may assume:

1. the invariant of all allocated objects that are relevant to the execution of m and m's precondition hold in m's prestate

2. after each call of a method, o.p($\vec{e}$), post(o.p($\vec{e}$)) holds and all allocated objects that are relevant to the execution of o.p($\vec{e}$) satisfy their invariants

Translated to BoogiePL, the following new assumptions can be generated:

1. At the beginning of every procedure the following assumptions are made:

```
assume (forall r:  ref ::
    ownerObject(this) == ownerObject(r)
    ==> INV(r, $Heap))); # peers
assume (forall r:  ref ::  isTOwnedBy(r, this)
    ==> INV(r, $Heap)); # reps;
```

2. After return from a method call with receiver o, its relevant objects can be assumed to satisfy their invariants:

```
assume (forall r:  ref ::
    ownerObject(o) == ownerObject(r)
    ==> INV(r, $Heap))); # peers
assume (forall r:  ref ::  isTOwnedBy(r, o)
    ==> INV(r, $Heap)); # reps;
```

Note that even though the postcondition of every method will state that the invariant of the receiver object will hold, Boogie will not be able to deduce that invariants of all objects relevant to that execution still hold. This is because there is no formalization that the receiver object can only break its own invariant. The fact that all relevant objects still satisfy their invariants could be added to a method's postcondition, but this would also force Boogie to verify more than it actually has to. The methodology already ensures the property. This is why the postcondition is only assumed.

4.5. Subtleties

While the final formulas seem easy to understand and are practically just a straightforward application of the methodology presented in the paper, there are quite some subtle issues a BoogiePL programmer has to face. Such the really time consuming part of my work was neither implementation nor design, but actually debugging. In this section I am going to present some of the issues I faced while testing my ownership invariants implementation. Looking backwards, everything that was figured out of course seems logical, but figuring out what exactly went wrong, was the really challenging part.

4.5.1. Heap Access

The invariant functions both take a heap parameter. In the Boogie translation there is only one heap though, thus at least to me it was not obvious that such a parameter would be needed. In Jive [14] two heaps are used, one as the current store and one to keep values from before a method's execution. Access to such values is needed for postconditions relating to values in the precondition (JML keyword \old). BoogiePL

however supports an *old* construct that does exactly the same, so there is one heap and it can be globally accessed. Thus the first version of the invariant functions did not have such a parameter. Of course they did not work, which is why now there is a heap parameter. The interesting part is why it did not work though. The example was pretty simple:

```
class A{
  int i;
  //@ invariant i > 0;
  void foo(){
    i = -1;
  }
}
```

The interesting part of the BoogiePL translation looked as follows:

```
assume INV(this);
$Heap[this, A.i] := -1;
assert INV(this);
```

Now, why would this pass the Boogie verification? When an assertion has to be verified, axioms are only used if there is no simpler way to do it. In the example above, the simplest way lies at hand, the property to be shown was just assumed two lines above. The odd thing is that the assignment does not destroy that property. - And, when looking at the axioms, it actually does break the invariant. Yet an invariant is not just a property of the *this* reference, but a property of the tuple *[this, heap]*. What is modified in the example above is the heap - and in the first formalization the invariant property was only designed to be one of the *this* object. Boogie thus cannot notice that an assignment to something completely different, which is not part of the invariant definition can go and break that property. Hence, the heap parameter had to be added, even though there is just one heap and the heap parameter will be the same for all usages of the invariant predicates.

4.5.2. Quantification over Types

After successfully testing some files which were meant to fail the invariant verification (and actually did). I tried an example looking like the following, which should actually pass verification. A first version was this:

```
class A{
  int i = 0;
  //@ invariant this.i >= 0;
  void foo(){}
}
```

and it passed just fine. But after altering method foo to:

```
void foo(){
  this.i = 3;
}
```

the verification failed. Now what had happened? The first example passed because a property was assumed on the *[this, heap]* tuple. In the actual method body neither this nor heap was touched. Thus asserting that the property still held was no problem for Boogie. In the second example, there was an assignment to field i, which meant that the heap was altered. To verify *INV*, its axiom was looked up, at that time it looked this way:

```
axiom (forall r:  ref, h:  [ref, name]any ::
   INV(r, h)
      <==>
      (forall t:  name ::
        (inv(t, r, h))));
```

To find out which of the *inv* predicates caused the problem I split the assertion `INV(this, $Heap)` up into the smaller assertions:

```
assert inv(A, this, $Heap);
assert inv(Object, this, $Heap);
```

and the verification passed... So, what went wrong? The problem lied in the axiom for *INV*. To be more specific, the quantification over types:

```
      (forall t:  name ::
        (inv(t, r, h))));
```

What Boogie does here is a quantification over all names. While classes are modelled using names, Boogie is actually free to instantiate t with other names, such as fields, which are modelled using the same mechanism. Here a list of name constants generated for the program above:

```
const Object:  name;
const A: name;
const A.i:  name;
```

Apparently, Boogie tried to verify `inv(A.i, this, $Heap)`. It might also have tried to verify `inv(N, this, $Heap)`, where N is an arbitrary name Boogie picked as instantiation. The important thing to note is that for all but the class names no axiom is

defined on *inv*. As there was a modification on the heap, Boogie had to verify *inv*(N, this, $Heap) though, because that modification might have potentially broken the invariant on N. What needed to be done was finding a way to limit the instantiations on types for the *INV* axiom.

The solution was to introduce a new function, *$isTypeName*, which returns true if a name parameter fed to it is a type and false if it is something else, for instance a field. The quantification over all names is then limited to only names which fulfill *$isTypeName*. Unfortunately, it is not sufficient to state for each type T introduced the axiom:

```
axiom $isTypeName(T);
```

because *$isTypeName* would then be underspecified and Boogie could not deduce that *$isTypeName*(A.i) needed to return false. What had to be done was to go through the whole compilation unit forest and specify the function once for all types. For the example above, the generated axiom looks as follows:

```
axiom (forall n:  name ::
   (( n== Object || n == A)
      ==> $isTypeName(n))
   &&
   ((n != Object && n != A)
      ==> !$isTypeName(n)));
```

It is not exactly elegant, but works. Note that even though this formula looks like a breach on modularity, it actually is not. It just has to collect all parts of the program because Boogie lacks a mechanism to partially define a method and add default return values.

4.5.3. Null References

An interesting aspect of the whole verification process proved to be **null** references. You might have noticed that in almost all axioms reference parameters are first checked to be **non-null** before anything is stated about a function. This is because **null** values are very dangerous. An easy example to show what can happen is the following:

```
void foo(){
  private rep Object o1:
  peer Object o2;
  // some code
}
```

Assume the UTS ownership information would be assumed at the top of the procedure implementation as follows:

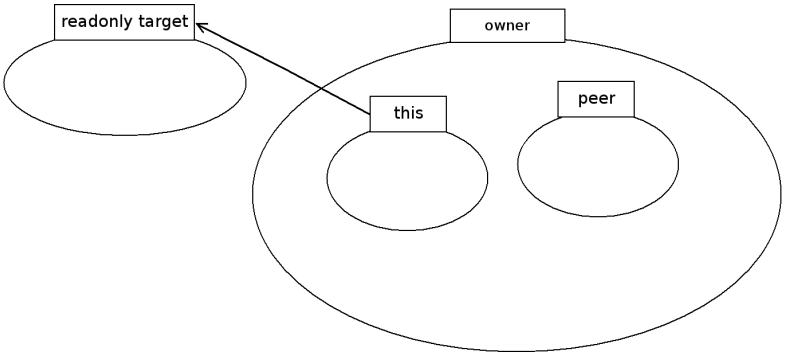

Figure 4.4.: readonly reference soundness issue

```
assume ownerObject(o1) == this;
assume ownerObject(o2) == ownerObject(this);
```

Now if both o1 and o2 are null, Boogie will deduce the following:

```
null == null;
==> o1 == o2;
==> ownerObject(o1) == ownerObject(o2);
==> this == ownerObject(this);
```

At this point Boogie has found out something that equals an *assume false;* because what it found out was the direct opposite of the axiom stating that **this** and the owner of **this** can never be the same. From this point on, anything can be proven.

4.6. Transitive Ownership and Readonly References

At the beginning of each method, invariants are assumed to hold for all relevant objects, respectively objects in the same context as the *this* object. This is done by using the recursively defined transitive ownership predicate which is not handled correctly by Simplify. In the paper, method calls on readonly references are restricted to objects inside the same context though. As this restriction cannot be checked correctly on the Boogie level, there results a soundness issue.

The problem is that a readonly reference might refer to an object outside the current context, upwards the ownership hierarchy - for instance to the owner of the owner of *this*, let's call it *oo*. The invariant of *oo* cannot be assumed to hold, as it is out of the this-object's context reach. We might however execute a pure method mp on *oo*, which

is allowed by the UTS. At the beginning of the execution of mp, invariants of objects relevant to *oo* are assumed, which is unsound. For a problem illustration take a look at Figure 4.4.

There are three options to deal with this problem. One would be to restrict readonly references to only point one layer deeper and actually enforce it by doing checks at verification time. A readonly reference could then only refer to peers or reps, which is quite a severe restriction as it renders readonly references almost useless.

The second option would be to leave things as they are and make readonly references only referring to objects inside the same context of a *this* object a programming guideline. This is unsound as the required property is not verified, but it leaves the opportunity to use the full power of readonly references.

Lastly, one could define the method non-recursively for a given number of layers deep. Defining transitive ownership for say ten layers would cover a good set of possible input programs. While it would be a compromise solution, it is also far from elegant.

For the time being, it was decided to take option two and leave the restriction as a guideline.

5. Modifies Clauses

5.1. Problem Description

In a JML modifies clause a programmer can state which locations of the store a method may modify. Callers of that method then know which locations they can still assume to have the same value as before the method call, which is important for verification. If no modifies clause is given, the verification system must assume that almost all properties known about the store before a method call are destroyed and cannot rely on information deduced before the call.

In the JML compiler (version 5.2_rc2) on which this project was based, the default assumption is that all locations are subject to modification if no specification was given. This is exactly the contrary in BoogiePL, where every location modified has to be stated explicitely. Thus, for all procedures, a

```
modifies $Heap;
```

specification is inserted. To help verification, support for modifies clauses was added with the following semantics and restrictions:

- a modifies clause always covers all fields of an object o (o.*) or all items of an array a (a[*])
- all but peers are assumed to be modified by default

Reason for the second bullet is that first, the represantation of another object is implementation detail and thus not of concern for the verification of a method. The other point is that objects outside our context must be assumed to be modified as there might be model fields which can refer to locations which are modified (for instance representation of other objects).

Thus, the only references which are looked at more closely are **peer** fields. If during a method execution, another method is called it can only modify its own representation or another **peer** field. Such a peer field might actually be the one where the first call emerged from (callback) or another peer, we can have write references to and it is thus interesting to know what peers are modified.

Note that this semantics means that all **readonly** and **rep** locations are assumed to

be modified after any method call. The only locations guarantees can be given about are peers.

5.2. Solution

Modifies clauses are translated into a part of the postcondition. The BoogiePL modifies clause only states that the heap is modified. Note that it is not possible to state at what location, the only thing one can state in a BoogiePL's modifies clause is that the global variable *$Heap* is modified that is all. The ensures clause gives restrictions on what parts of the heap exactly could have been modified. Two parts are added to the ensures clauses, one for peers and one for peer references stored in an array of the current class. The contribution for fields looks as follows:

```
(forall $r: ref, $n: name::
  $r != null
  && $isFieldName($n)
  && typeOf($r) <: fieldHome($n)
    ==>
  ownerObject(this) != ownerObject($r)
  || old($Heap[$r, $n]) == $Heap[$r, $n]
  || old(cast ($Heap[$r, alloc], bool)) == false
  || $r == Tr [[ reference given in modifies clause ]] )
```

On the left hand side of the implication, restrictions on what references need to be looked at are made. The references that are interesting are those pointing to the heap - null references do not satisfy that requirement. On the heap, fields (*$isFieldName*`($n)`) need to be looked at - and actually just those fields a `$r` object actually has (*typeOf*`($r)` <: *fieldHome*`($n)`).

On the right hand side the possible poststates for objects satisfying the LHS are given. Either they are not peer, they were not modified, or they were not allocated in the prestate. Lastly follows the translation of references mentioned in the modifies clause. If the modifies clause was empty, the (|| true) is appended.

The modifies contribution for fields actually covers all fields, but not array elements. That is because arrays are modelled differently using the custom elements type. To translate array accesses, a more complicated formula has to be inserted:

```
(forall $start: ref, $arrayRef: ref,
     $n: name, $i: int, $e: elements ::
  $start != null
  && $arrayRef != null
  && old(isAllocated($start, $Heap))
```

```
&& old(isAllocated($arrayRef, $Heap))
&& $arrayRef == cast ($Heap[$start, $n], ref)
&& $isArrayFieldName($n)
&& typeOf($start) <:  fieldHome($n)
&& $i >= 0
&& $i < length($arrayRef)
&& $e == cast ($Heap[$arrayRef, $array$], elements)
  ==>
ownerObject(this) != ownerObject( refElementSelect($e, $i))
|| old( refElementSelect($e, $i)) == refElementSelect($e, $i)
|| $arrayRef == Tr [[ reference given in modifies clause ]]
```

The access chain to arrays goes over two references, denoted above with $start and $arrayRef. The array access would look something like $start.$arrayRef[i]. The preconditions on the LHS of the formula are similar to fields, only that they are now split on two references instead of one. The reason why $start is needed is that only by starting at that point enables the formula to make sure that $arrayRef actually is an array reference. Together with $n being an array field name this selection is made. Restriction on $i are obvious, array access beyond the boundaries would make no sense.

On the RHS, there are basically the same options as for fields: not peer or not changed. After these two options are covered, the exceptions which are explicitely mentioned in the specification are stated. Again, if there is no modifies specification for a method, || true is added.

5.3. Subtleties

5.3.1. JML

While implementing support for modifies clauses I had to find out that the implementation on the JML compiler is far from bug free. The first problem I encountered was that a specification like:

```
//@ modifies this.*;
void foo(){
  this.x = new X();
}
```

would raise the compile time error that this.x is not in the set of modifiable fields. Apparently, this.* was just skipped when the set of modifiable fields was calculated. After implementing a method that expanded .∗ into all the related fields, the next bug was waiting for me - array accesses. Consider the following specifications:

```
//@ modifies this.arr [2].*;
```

```
//@ modifies this.arr [*];
//@ modifies this.arr [0].*;
```

JML would translate all of these expressions to:

```
this.arr [0]
```

At first I thought the information was not even parsed, but that proved to be wrong. What happened was that the information was actually parsed but not really translated well during typechecking. JML basically builds an array of strings *[this, arr, [], *]* and at typechecking builds an expression out of these. For array accesses, the index would not be fetched but a zero was inserted. After finding this out, it was easy to fix.

Another important thing to note is that JML does very little static checks regarding modifies clauses. So far the only check I have noticed was the one mentioned above, assignment to a field which is not mentioned in the modifies clause. Assignment to array elements not mentioned in modifies clauses pass the compiler without complaint - as do modifying method calls on fields.

Also interesting is the way modifies clauses are translated in Spec#. There, the specification is added in free ensures expressions. These are postconditions which can just be assumed to hold without verification. It must thus be assumed that they statically check whether modifies clauses are respected.

5.3.2. Array Translations to BoogiePL

After testing many different ways of implementing modifies clauses for arrays I finally found out that the current translation of arrays to BoogiePL was buggy. Taking a look at Spec# translations to see how they handled arrays was not helpful, they have the same issue. Consider the following simple Spec# example:

```
class B{}
class C{
  public void foo(){
    B[] barr = new B[10];
    barr[1] = new B();

    C[] carr = new C[5];
    carr[1] = new C();

    assert carr[1] != barr[1];
  }
}
```

In this example, two new arrays are created and new objects are assigned as elements of the array. The arrays are even of a different type. However, Boogie fails to show that the assertion at the end of method `foo` holds. The reason for this is as far as I see the way arrays are modelled. On the BoogiePL level the JML array reference is modeled as a **ref** type. To access array items, a custom type **elements** is used. Here the problem

arises, there seems to be no axiom or statement making sure that **elements** of array barr and carr are not the same. As I found this error towards the very end of my project, there were no further investigation on the bug - and no attempts to fix it.

5.3.3. Boogie slowdown

While the current implementation works and is able to find all errors in my testfiles (with the exception of those with multiple arrays), there is the drawback that verification becomes a lot slower with each added method/constructor call in a method's body. If there are three or more method calls, execution time will quickly go into minutes. This is just something to note, Boogie will terminate, it just might take a while.

6. History Constraints

6.1. Problem Description

A history constraint expresses properties for the object state on the class level. The property states how values change between publicly visible execution states. As usually not all methods of a class change values referred to in a constraint, the constraint expression has to be reflexive and transitive [11, Sec. 8.3].

An example would be a collection class where one cannot remove elements. The number of elements in that class could only increase, which can be expressed as follows:

```
class GrowingCollection{
  List elementList;
  //@ constraint \old(elementList.count) <= elementList.count;

}
```

Here a first difference to invariants is obvious - the use of the **\old** predicate. While an invariant just states a property that can be verified on its own, the property of a history constraint states a relation to the previous visible state. By this, the **\old** predicate actually is given a different semantics than usual. When used in a postcondition of a method m, **\old** refers to the value of a field in the prestate of the m's execution. For history constraints this semantics is too weak. Consider the following method:

```
void addAndRemove(Object o){
  elementList.add(o);     // (1)
  elementList.remove(o); // (2)
}
```

Looking at the example with the normal **\old** semantics, the history constraint is not violated. The number of elements in the collection is the same in the pre- and in the poststate. However, during the execution of the method, the this object went through some more visible states, namely before and after the execution of (1) and (2). After execution of method *add*, the number of elements in the collection was actually increased by one. The call of the *remove* method thus violates the history constraint, as it reduces the number of elements again.

6.2. Solution

To deal with the different old semantics, a new heap variable had to be introduced (*$histHeap*). The *$histHeap* is an exact copy of the *$Heap* in the last visible state.

This means the two heaps hold the same values:

- at the beginning of every method
- after every method call

Before every visible state, it has to be shown that the changes performed on the *$Heap* did not violate any history constraint. These states are:

- at the end of every method
- before every method call

The actual predicates modelling the constraint expressions are very similar to the way invariants were handled. There are two predicates, *HC* and *hc*, whereas *HC* conjoins all *hc*. For each class a *hc* is introduced, which takes two heap parameters and compares the constrained values.

Formalization:

```
function hc(hh: [ref,name]any, heap: [ref, name]any, t: name)
         returns (bool);
function HC(hh: [ref,name]any, heap: [ref, name]any)
         returns (bool);
```

For each class C, the following axiom is generated to define the *hc* of C:

```
axiom (forall hh: [ref,name]any, heap: [ref,name]any ::
  hc(hh, heap, C)
  <==>
  (forall r: ref ::
    ((r != null) ==>
      ((typeOf(r) <: C) ==>
        Tr ⟦ HC_c ⟧))));
```

Axiom to conjoin all *hc*s:

```
axiom (forall hh: [ref,name]any, heap: [ref,name]any ::
  HC(hh, heap)
    <==>
    (forall t: name :: $isTypeName(t)
      ==> (hc(hh, heap, t))));
```

If a method passes the assertions mentioned above, it has been shown that no history constraints are violated by its execution. Having verified all the small steps, it has also been shown that throughout the method's execution the history constraint held. It is thus safe to introduce a free ensures clause (which does not have to be further verified) for the method, stating a translation of the history constraint.

Formalized, after each visible state (begin of a method body, after each method call inside a method body), the assumption that both heaps contain the same values is stated:

```
havoc $histHeap;
assume (forall $r: ref, $n: name ::
  $histHeap[$r, $n] == $Heap[$r, $n]);
```

Note that the `havoc` statement is needed to ensure that actually the values of the *$Heap* are assigned to the *$histHeap* and not vice-versa. Also, to be allowed to execute the `havoc` statement, the variable *$histHeap* needs to be added to each method's *modifies* clause.

Before each visible state (end of method body/return statement, method call), the following assertion has to be shown:

```
assert HC($histHeap, $Heap);
```

6.3. Applying Modular Ownership Technique

The technique described above is very basic. That is to say, a method has to show that for all classes and all references in the program it does not violate any history constraints. It also requires a whole program analysis and is thus not modular. To achieve modularity, the ideas of ownership invariants can be reused.

A basic idea of ownership invariants was to use UTS information to restrict admissibility on invariant expressions. Only locations owned by an object (in its context) or declared in the object's class are admissible. - In other words, only locations over which a class has a certain degree of control are permitted to appear in invariant expressions. This restriction on admissibility can be taken over and applied to history constraints. As a result, proof obligations can be reduced to not show the property for all references, but only for the relevant ones. Formulas would thus need a new parameter r to denote what references need their history constraints checked:

```
function
  hc(hh:  [ref,name]any, heap:  [ref, name]any, t:  name, r:  ref)
         returns (bool);
function
  HC(hh:  [ref,name]any, heap:  [ref, name]any, r:  ref)
         returns (bool);
```

Leading to the following changes in the axioms:

```
axiom (forall hh:  [ref,name]any, heap:  [ref,name]any, r:  ref ::
  hc(hh, heap, C, r)
  <==>
  ((r != null) ==>
    ((typeOf(r) <:  C) ==>
      Tr ⟦ HC_c ⟧))));

axiom (forall hh:  [ref,name]any, heap:  [ref,name]any, r:  ref ::
  HC(hh, heap, r)
    <==>
    (forall t:  name ::  $isTypeName(t)
      ==> (hc(hh, heap, t, r))));
```

In words, the new axioms do not quantify over all references anymore. The functions can only be used on one reference, for which its history constraints are checked. As only the this-object can break its history constraints, it is sufficient to only show the assertion for history constraints on **this** before each visible state:

```
assert HC($histHeap , $Heap , this);
```

On the assumptions side, applying relevant object semantics means that only locations which are relevant to the this-object can be assumed to satisfy their history constraints. This means that it is now not possible anymore to simply create a copy of the *$Heap*. Instead, the actual predicate has to be used. Formula to assume history constraints on objects relevant to the this-object:

```
assume (forall $r:  ref, $n:  name ::
  ownerObject(this) == ownerObject($r) # peers
  || isTOwnedBy($r, this) # reps
    ==>
  HC($histHeap , $Heap , $r));
```

This assumption can be stated at the beginning of a method's body. After a method

execution m on a receiver object o, one can assume that the history constraints of all objects relevant to o hold:

```
assume (forall $r:  ref, $n:  name ::
   ownerObject(o) == ownerObject($r) # peers
   || isTOwnedBy($r, o) # reps
      ==>
   HC($histHeap , $Heap , $r));
```

Unfortunately, at the end of the project, there was no time left to implement history constraint support to the compiler. The formulas presented above were only tried out in some simple examples by putting them in by hand to at least get a proof of concept.

7. Miscellaneous Others

In this chapter I will describe the translation of some other JML constructs to BoogiePL. The distinction to the constructs described in previous chapters is that there is not enough to tell about the following constructs to earn them a full, own chapter. Some are just syntactical conversions, some a bit more complicated.

7.1. Casts

Especially when dealing with container classes, there are often methods that either return an *Object* reference or take an *Object* parameter. But what is actually passed to those stores are not objects of type *Object* but all kind of types. This would not be possible without some notion of cast and as I ran into this restriction several times I decided to implement casts which have to be verified by the Boogie system.

In order to keep things simple, casts may only appear in assignments. It is forbidden to have a cast in the middle of an expression (enforced by compiler). This not only helps the compiler to have a simple translation, but also the programmer who has it easier to find a mapping from BoogiePL code back to the JML code if Boogie fails to verify a cast.

For every cast of a reference r to target type T:

```
T target = (T) r;
```

An assertion is inserted to verify the legality of the cast:

```
assert r == null || typeOf(r) <:  T;
```

Note that the **null** case has to be treated specially again. Cast of null to any type will always be successful. Now, how can the assertion be fulfilled by a JML program? There are two cases to distinguish:

- Downcasts
- Upcasts

An upcast is a cast from a subtype to a supertype. This is in fact trivial. As the type system is mapped to Boogie, the tool can deduce for any type Sub which is a subtype

of type Super, that the assertion Sub <: Super will hold.

To enable a downcast to pass is not that trivial. There are several possibilities though. The first is to add type information in specification. One could state in the precondition of a method m that a parameter p of type *Super* is in fact of a *Sub* type. In the method body, a cast of p to *Sub* could then successfully pass the verification. Another setup that would pass verification is the creation of a new *Sub* object which is first assigned to a *Super* variable and later cast back to *Sub*. This would pass because after the creation of the *Sub* object, its exact type is known to Boogie and this information is not lost after assignment to a different variable.

As Universe types can also be cast to different Universes, these have to be checked at run-time as well. In this case there are three cases:

- casts to readonly
- casts to peer
- casts to rep

There are no requirements to cast any object to its readonly type, thus there is nothing to be verified. For a cast of a reference r to **peer**, it has to be shown that the r's owner is the same as the owner of the this reference:

```
assert ownerObject(r) == ownerObject(this);
```

If r is cast to a **rep** type, one has to show that the owner of r is the this-object and because of subclass separation (subsection 4.3.1), it also needs to be shown that the owner type of r is the type in which the current method was declared, let us call it C:

```
assert ownerObject(r) == this
     && ownerType(r) == C
```

7.2. Quantifications

JML quantifications [11, section 11.4.24] are a powerful specification construct, especially when used for arrays. For instance, one could state that all array elements are never null or always positive. In the compiler, so far only forall quantifications are supported. Translation to BoogiePL is mainly of a syntactical nature.

```
JML:
\forall (varDecls:  V) [(; [Exp:  RE); ] (Expression:  E);

BoogiePL:
forall Tr [[ V ]] ::  Tr [[ RE ]] ==> Tr [[ E ]];
```

Unfortunately, such forall quantifications seem very difficult to handle for Boogie. In one example I had an array of size ten where elements were not allowed to be negative. After Boogie was trying to verify it for over ten minutes and Simplify had eaten a few hundred megabytes of memory, I aborted the verification and step by step reduced the array size. When the array was shrinked to four elements, Boogie finally was able to verify my code. Thus, this feature is implemented and it works, but only for small quantifications as Simplify seems to run into problems with bigger ones.

7.3. Native Methods

When working on a bigger example where several methods interacted together I soon found myself in trouble writing specifications such that:

1. all the methods interact correctly
2. all methods satisfy their postconditions
3. Boogie is able to verify forall quantifications

I was then given the hint that in the Jive tool, native methods are used to state methods which have just a specification but no actual implementation. Those native methods were just assumed to satisfy their specification. I liked the idea and implemented a translation of native methods to Boogie which would just do that. Native method translation now works like normal method translation with two differences:

1. instead of an ensures clause, a free ensures clause is written
2. there is just a dummy procedure body

The rest is handled/translated just like normal methods. The free ensures clauses are part of BoogiePL and can be used to state postconditions which do not have to be verified by Simplify. Using this mechanism works just fine and offers a workaround for methods which Boogie can not verify due to too difficult quantifications or currently not supported features used. For a program that has parts which are supported/handled by Boogie and parts that are not handled by Boogie, the latter part could be made native for verification purpose and the tool could still be used to verify the first part.

8. Conclusion

8.1. Achievements

In this paper I have shown how a set of JML constructs were translated to BoogiePL. All the translations were actually implemented in a compiler and tested thoroughly. Automatic verification of JML programs is possible with the resulting tool as long as a programmer limits himself to the supported constructs. One drawback currently is the lack of a backward link, if Boogie reports errors, the BoogiePL code has to be looked at and the link back to the JML program has to be found manually. This is in most cases not so hard to do, as in the BoogiePL code comments are inserted which indicate what statements the code relates to. There are some other restrictions, pointed out in this paper, like Boogie not terminating for difficult forall quantifications, but these can be worked around by modifying the code a little in most cases.

Note that I put the focus in this project on implementing features and get them to work properly. I think it would have easily been possible to implement two or three other JML constructs as well, but at the cost of having done fewer tests and getting a product that is a lot less trustworthy. While I do not claim that the compiler is now error free, there are many bugs which were found and fixed during the course of the project. It was a decision of quality over quantity, which I deemed very important, especially as the topic is verification of software.

I have also pointed out some of the pitfalls in working with the Boogie system. It being an automatic verification system holds the benefits of potentially high verification speed and is thus fitted with the requirements for working together with an IDE. However, during the verification process, there not only is no user interaction needed, it is impossible. This makes debugging a challenge as there is no way of finding out exactly what is going on, the only option left is to play around with the input program and hoping for relevation. This is very time consuming and at times frustrating.

8.2. Future Work

There are still many JML constructs left that could be supported in future version of the compiler. Interesting next steps would be to apply visibility based invariants semantics to JML, give support for model fields or work on static fields and methods. Another not so trivial task would be to make the upgrade to the newest JML compiler. I was told that there were quite some changes to the JML compiler since the last version and that

some things work differently now, partly having different semantics. Lastly, a link back to an IDE would also be a very interesting enhancement.

8.3. Boogie Feature Wishlist

Lastly, a little wishlist for the Boogie system. These are the three things I missed mostly during my project:

- some kind of help to find errors in BoogiePL programs
- underspecified definitions for functions with default return values
- support for recursive definitions

I think the first bullet speaks for itself. By the second one I am thinking of some ugly solutions that had to be found in my project. One being the `$isTypeName`, which has to be defined in one go after collecting information from the whole compilation unit forest. The idea is to give the BoogiePL programmer the option to give a general specification for a function, stating a default return value. For exceptional cases there would be the possibility to declare axioms where a different value is returned. This would allow for simpler and more elegant solutions in some cases, but also enrich the input language in general.

The last bullet is not even to blame on Boogie, but on Simplify, as I was told it is a known problem that Simplify cannot handle recursion. Still, especially for the UTS support, a way of making recursive definitions would be most wanted, as currently the given expressiveness only allows one to look one layer up or down in a layered ownership structure, which is insufficient to describe properties where a complete context has to be formalized (for instance readonly method calls in ownership invariants and acyclicity of ownership).

Another nice thing would be to have Boogie kill Simplify when it is forced to quit by ctrl-c. It is just annoying to always go to the task manager and kill the Simplify process separately - but that has little to do with verification.

8.4. Personal Opinion

This master project gave me the chance to get an insight into current research on verification. Looking back, it seems naive to me that I expected to be able to design and implement support for a far bigger number of constructs than what I ended up with. While in the beginning I planned four days of implementing and one day of testing my code peer week for the implementation phase, I often found myself four days testing and bug searching and only one day actually getting on with the code. This has a lot to do with the nature of the project. There are a number of different tools involved, all of

which turned out to be non-trustworthy in the end. Sometimes it was the basic JML compiler that failed me, sometimes the code of the previous semester project, sometimes Boogie - and of course my additions. Finding out which out of these tools caused the unexpected behaviour was not trivial to start with and fixing one error often just led to the next one popping up. As a result, my frustration level had one peak followed by the other.

Nevertheless, it was an interesting experience to deal with automatic verification and the resulting compiler sure does help implementing programs. Often, new test files did not pass verification the first time because I actually did make errors in their implementations. A good example to that was adding invariants to an example without actually writing a constructor that established it. Such experiences of Boogie showing me that my code was a bit too sloppy to pass verification were also an encouragement, because I could actually see that such a tool really can provide to be useful.

So in the end there are mixed feelings about automatic verification. On one hand side, as long as things work fine, a tool like Boogie can add much to the safety and correctness of code. On the other hand side, limits of Boogie can be reached fairly quick and then, very little can be done to help the verification system deduce a correct result. If the latter case happens often, trust to a system cannot be built up and no one will use it.

Bibliography

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6), 2004. `www.jot.fm`.

[2] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *PASTE*, 2005.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, March 2004.

[4] Samuel Burri. Translation of object-oriented programs into guarded commands. *"http://sct.ethz.ch/projects/student_docs/Samuel_Burri_SA_paper.pdf"*, 2005.

[5] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[6] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (MSR), March 2005.

[7] Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.

[8] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[9] Rajeev Joshi. Extended static checking of programs with cyclic dependencies, 1997. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-1997-028-html/joshi.html`.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, January 2006. To appear in *ACM SIGSOFT Software Engineering Notes*.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*, November 2004. Draft revision 1.98.

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

[13] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[14] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from `sct.inf.ethz.ch/publications`, 2000.

[15] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[16] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.

[17] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*. Springer, 1998.

[18] The JML project homepage: `http://www.cs.iastate.edu/~leavens/JML/`.

[19] The Spec# project homepage: `http://research.microsoft.com/specsharp/`.

A. Supported and Unsupported Constructs

A.1. Java part

In the current version of the compiler, the following limitations apply on the Java level:

Outside of Type Declarations:

- Import statements.

In Type Declarations:

- Static and nonstatic initializers.
- Inner classes.

Statements:

- For-loop, do-loop, switch-block.
- Exceptions

Expressions:

- Casts may not appear within expressions. There is a cast statement that allows to apply a cast to an expression. The result is assigned to a local variable.

Expression Operators:

- Compound assignment operators (†= where † represents any binary op)
- Assignment within expressions (`x = y = z`).
- Pre- and postfix operators ++ and −−.

General Language Constructs:

- Primitive types char, long, float, double.
- Multi-dimensional arrays. Arrays may only have one dimension.
- Array initializers.
- Labeled statements, labeled break.

- Continue, break.
- It is not allowed to declare several local variables or fields at once (e.g. `int a, b;`); they have to be declared one by one.
- String literals, character literals and floating point literals.
- Reflection.
- 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.
- Overloading is not supported. Methods could be renamed to support the same functionality
- Shadowing. All fields of a class (including inherited fields from superclasses) should have different names.
- Access Modifiers. Visibility and access control of classes, methods and fields are not covered by the BoogiePL code. However, the static provided by JML do work properly.

A.2. JML part

In JML specifications, the following constructs are supported:

- Pre-and postconditions.
- Modifies clauses.
- Assume and assert statements.
- Loop invariants.
- Universe Types.
- Invariants (ownership based).
- Forall quantifications.
- \ownertype predicate.

B. Language Description

B.1. Subset of supported JML Grammar

The following Subset of the *JML* Grammar is supported by the compiler.

B.1.1. Notations

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	*Java* Class that represents one or more productions *Prod*

B.1.2. Types, Programs and Classes

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

Type ::= `void`
| `boolean`
| `int`
| *ReferenceType*

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

Universe ::= *peer*
| *rep*
| *readonly*

CompilationUnit ::= (*ClassDefinition*)*

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

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

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

B.1.3. Method Declarations and Specifications

MethodDeclaration ::= *MethodSpecification Type Ident Formals CompoundStatement*
| *MethodSpecification Ident Formals ConstructorBlock*

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

ParameterDeclaration ::= *Type Ident*

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

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

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

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

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

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

StoreRefExpression ::= `this` (`.` *Ident*)* *StoreRefNameSuffix*

StoreRefNameSuffix ::= `.` *Ident*
| `.*`
| `[` *Expression* `]`
| `[*]`

B.1.4. Statements

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

Statement ::= *CompoundStatement*
| *VariableDeclarations* `;`
| *MethodCall* `;`
| *Expression* `=` *RightHandSide* `;`
| `if (` *Expression* `)` *Statement* [`else` *Statement*]
| (*Maintaining*)* `while (` *Expression* `)` *Statement*
| `return` [*Expression*] `;`
| `;`
| `/*@ assert` *Expression* `; */`
| `/*@ assume` *Expression* `; */`

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

VariableDeclarator ::= *Ident* [`=` *Expression*]

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

MethodCall ::= *DereferenceExpression* `(` [*ExpressionList*] `)`

NewExpression ::= `new` [*Universe*] *Type* `(` [*ExpressionList*] `)`
| `new` [*Universe*] *Type* `[` *Expression* `]` (1 dim.)

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

B.1.5. Expressions

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

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

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

ImpliesExpression ::= *LogicalOrExpression*
| *LogicalOrExpression* (`==>` *LogicalOrExpression*)+
| *LogicalOrExpression* (`<==` *LogicalOrExpression*)+

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

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

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

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

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

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

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

MultExpression ::= *UnaryExpression*
| *UnaryExpression* `*` *MultExpression*
| *UnaryExpression* `/` *MultExpression*
| *UnaryExpression* `%` *MultExpression*

UnaryExpression ::= `-` *UnaryExpression*
| `!` *UnaryExpression*

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

DereferenceExpression ::= *PrimaryExpression*
| *DereferenceExpression* `.` *Ident*
| *DereferenceExpression* `[` *Expression* `]`

PrimaryExpression ::= *Ident*
| *Literal*
| `super` only allowed for method calls - no field access
| `true` | `false`
| `this`
| `null`
| `(` *Expression* `)`
| *JmlPrimary*

Literal ::= `0` | `1` ...

JmlPrimary ::= `\old(` *Expression* `)`
| `\typeof(` *Expression* `)`
| `\type(` *Type* `)`
| `\result`
| `\ownerobject(`*Expression* `)`
| `\ownertype(`*Expression* `)`

B.2. Translation to BoogiePL

B.2.1. Setup

Constant literals denoting the types in *BoogiePL*.

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

Type `elements` for arrays and its dummy constant for heap access.

```
type elements;
const $array$: name;
```

The heap is modeled using an array.

```
var $Heap: [ref, name]any;
```

Using the following functions read-only fields are accessed.

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

Arrays are always of positive length:

```
axiom (forall r: ref :: length(r) >= 0);
```

Functions to find out whether given names denote fields/arrays.

```
function $isFieldName(name) returns (bool);
function $isArrayFieldName(name) returns (bool);
```

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

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

Some axioms for array types:

```
axiom( forall n: name ::
        arrayType(n) <: Object && 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)));
```

The following functions are used to model array access and modifications.

```
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);
```

And they satisfy these 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)) ) );
```

Predicates dealing with subtyping:

```
function $oneClassDown(name, name) returns (name);
function $asDirectSubClass(name, name) returns (name);
```

Axiom defining *$oneClassDown*:

```
axiom (forall X: name, Y: name, Z: name ::
   Z <: $asDirectSubClass(Y, X)
   ==> $oneClassDown(Z, X) == Y;
```

Axioms regarding Universe type system:

Object cannot own itself:

```
axiom (forall r:  ref ::  (r != null)
    ==> r != ownerObject(r));
```

Transitive ownership:

```
function isTOwnedBy(ref, ref) returns (bool);

axiom (forall owned:  ref, owner:  ref ::
(owned != null && owner != null) ==>
   (isTOwnedBy(owned, owner)
   ==>
   ownerObject(owned) == owner
   || isTOwnedBy(ownerObject(owned), owner)));
```

Acyclicity:

```
axiom (forall r1:  ref, r2:  ref ::
   (r1 != null && r2 != null
   && isTOwnedBy(r1, r2))
      ==>
   !isTOwnedBy(r2, r1));
```

Functions and axioms for invariants:

```
function inv(T: name, r:  ref, h:  [ref, name]any) returns (bool);
function INV(r:  ref, h:  [ref, name]any) returns (bool);
```

Axiom to conjoin all *inv*'s:

```
axiom (forall r:  ref, h:  [ref, name]any ::
   INV(r, h)
      <==>
      (forall t:  name ::  $isTypeName(t)
         ==> (inv(t, r, h))));
```

B.2.2. Types

Only the following *BoogiePL* types will be used throughout the translation:

`bool` Corresponds to the *Java/Jml* type `boolean` and represents the values `true` and `false`.

`int` Corresponds to the *Java/Jml* type `int`, modelling integer numbers.

`ref` Comparable to basic pointers and used to model *Java/Jml* object and array references. 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` 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.

First, a mapping of *Java/Jml* types to *BoogiePL* types is needed:

$$\begin{array}{lcl} \mathit{CoarseType}\ [\![\ \texttt{boolean}\ ]\!] & = & \texttt{bool} \\ \mathit{CoarseType}\ [\![\ \texttt{int}\ ]\!] & = & \texttt{int} \\ \mathit{CoarseType}\ [\![\ \mathit{ReferenceType}\ ]\!] & = & \texttt{ref} \end{array}$$

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

As *BoogiePL* does not offer a rich type system *Java/Jml* types are modelled as *BoogiePL* names and therefore the following mapping of *Java/Jml* types to *BoogiePL* names is needed.

$$\begin{array}{lcll} \mathit{Type}\ [\![\ \texttt{boolean}\ ]\!] & = & \mathfrak{basicBool} & \\ \mathit{Type}\ [\![\ \texttt{int}\ ]\!] & = & \mathfrak{basicInt} & \\ \mathit{Type}\ [\![\ \texttt{Object}\ ]\!] & = & \mathfrak{Object} & \\ \mathit{Type}\ [\![\ \mathit{Ident{:}I}\ ]\!] & = & I & \text{for all user defined classes} \\ \mathit{Type}\ [\![\ \mathit{Type{:}T}\texttt{[]}\ ]\!] & = & \mathit{arrayType}\texttt{(}\mathit{Type}\ [\![\ T\ ]\!]\texttt{)} & \end{array}$$

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

Throughout the translation the following notation is used: $\hookrightarrow$ `(` `var` *varName*`:` `BPLType` *type*`,` `BPLName` *name*`;` `)` to indicate that a new variable with name *varName* and *BoogiePL* type *type* is introduced. If type == `ref` a *BoogiePL* name - *name* is needed, too. Since local variables can only be introduced in the beginning of a *BoogiePL* method all temporary variables introduced during translation have to be collected and placed in front of all statements.

Function *typeOf* .

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

The following two functions assign class members to their 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, h:[ref, name]bool ::
   isAllocated(r, h) <==> (r == null || cast ($Heap[r, alloc], bool)) );
```

B.2.3. Programs and Classes

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

Assumption: there are no fields and methods defined in Object except *inv* and *commited* which will be handled specially.
Assumption: there is no overloading - neither for methods nor for constructors. That means there is maximally one constructor per class.
Throughout the following translations $\mathfrak{C}$ will be used to denote the name of the class being translated.
Initialization expressions are handled within the constructor.

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

```
const C: name;
axiom( C <:  Object && asDirectSubClass(C, Object) == C);
axiom( forall K: name ::  C <:  K <==> C == K || Object <:  K);
# for all ( Type:T  Ident:I  [  =  Expression ] ; ∈ Ms
   const C.I: name;
   # if T  is ReferenceType then
      TrRefField ⟦ T ,  C.I  ⟧
   # end if
# end for all
# for all MethodDeclaration:MD  ∈ Ms
   Tr ⟦ MD ⟧
# end for all
axiom (forall r:  ref, h:  [ref, name]any ::
   inv(C, r, h)
   <==>
   ((r != null) ==>
```

```
        ((typeOf(r) <: C) ==>
    # for all InvariantExpressions:IE ∈ Ms
      Tr ⟦ IE ⟧
    # end for all
      )));
    # if there is no constructor defined within class C
      Tr ⟦ C () {;} ⟧
    # end if
```

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

```
  const C: name;
  axiom( C <: E && asDirectSubClass(C, E) == C);
  axiom( forall K: name :: C <: K <==> C == K || E <: K);
  # for all ( Type:T Ident:I [ = Expression ] ; ∈ Ms
    const C.I: name;
    # if T is ReferenceType then
      TrRefField ⟦ T , C.I ⟧
    # end if
  # end for all
  # for all MethodDeclaration:MD ∈ Ms
    Tr ⟦ MD ⟧
  # end for all
  axiom (forall r: ref, h: [ref, name]any ::
    inv(C, r, h)
    <==>
    ((r != null) ==>
      ((typeOf(r) <: C) ==>
  # for all InvariantExpressions:IE ∈ Ms
    Tr ⟦ IE ⟧
  # end for all
    )));
  # if there is no constructor defined within class C
    Tr ⟦ C () {super();} ⟧
  # end if
```

TrRefField ⟦ *Type:* *T* , *Ident:* *I* ⟧ =

```
axiom( fieldHome(C.I) == Type[[ C ]] );
axiom( fieldType(C.I) == Type[[ T ]] );
# if T is peer
   axiom (forall r:  ref ::
      (r != null && typeOf(r) <:  C
      && cast ($Heap[r, C.I], ref) != null)
         ==>
      (ownerObject(cast ($Heap[r, C.I], ref)) == ownerObject(r)
      ownerType(cast ($Heap[r, C.I], ref)) == ownerType(r)));
# else if T is rep
   axiom (forall r:  ref ::
      (r != null && typeOf(r) <:  C
      && cast ($Heap[r, C.I], ref) != null)
         ==>
      (ownerObject(cast ($Heap[r, C.I], ref)) == r
      ownerType(cast ($Heap[r, C.I], ref)) == C));
# end if
# if T is array holding peer references
   axiom (forall r:  ref ::
      (r != null && typeOf(r) <:  C
      && cast ($Heap[r, C.I], ref) != null)
         ==>
      (forall item:  ref, i:  int ::
         item != null
         && item =
            refElementSelect(cast ($Heap[r, $array$], elements), i)
            ==>
         (ownerObject(item) == ownerObject(r)
         ownerType(item) == ownerType(r))));
# end if
```

B.2.4. Method and Constructor Declarations and Specifications

Assumptions: all methods have different names. This could be achieved by inserting parameter types into method names, but is currently not implemented.

Translation of a normal method.

$$
\begin{aligned}
&\mathit{Tr}\ [\![\ \mathit{MethodSpecification{:}MS}\ \ \mathit{Type{:}RT}\ \ \mathit{Ident{:}N}\ \ \mathit{Formals{:}F}\ \ \mathit{CompoundStatement{:}Body}\ ]\!] \ = \\
&\qquad \texttt{procedure}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ RT\ ]\!]\ ;\ \mathit{TrSpec}\ [\![\ N\ ,\ \ MS\ ]\!] \\
&\qquad \texttt{implementation}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ RT\ ]\!] \\
&\qquad\qquad \mathit{TrBody}\ [\![\ N\ ,\ \ F\ ,\ \ RT\ ,\ \ Body\ ]\!]
\end{aligned}
$$

Translation of a native method. (Needs no implementation.)

$$
\begin{aligned}
&\mathit{Tr}\ [\![\ \mathit{MethodSpecification{:}MS}\ \ \mathit{Type{:}RT}\ \ \mathit{Ident{:}N}\ \ \mathit{Formals{:}F}\ ]\!] \ = \\
&\qquad \texttt{procedure}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ RT\ ]\!]\ ;\ \mathit{TrFreeSpec}\ [\![\ N\ ,\ \ MS\ ]\!] \\
&\qquad \texttt{implementation}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ RT\ ]\!]
\end{aligned}
$$

Translation of a constructor. ($N = \mathfrak{C}$)

$$
\begin{aligned}
&\mathit{Tr}\ [\![\ \mathit{MethodSpecification{:}MS}\ \ \mathit{Ident{:}N}\ \ \mathit{Formals{:}F}\ \ \mathit{CompoundStatement{:}Body}\ ]\!] \ = \\
&\qquad \texttt{procedure}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ \texttt{void}\ ]\!]\ ;\ \mathit{TrSpec}\ [\![\ N\ ,\ \ MS\ ]\!] \\
&\qquad \#\ \textbf{var}\ \mathit{FieldDecList} = \emptyset\ ; \\
&\qquad \#\ \textbf{for all}\ \mathit{FieldDeclaration{:}FD}\ \textbf{in class}\ \mathfrak{C} \\
&\qquad\quad \#\ \ \mathit{FieldDecList} = \mathit{FieldDecList} \cup \mathit{FD}\ ; \\
&\qquad \#\ \textbf{end for all} \\
&\qquad \texttt{implementation}\ \mathfrak{C}.N\ \ \mathit{TrSig}\ [\![\ N\ ,\ \ F\ ,\ \ \texttt{void}\ ]\!] \\
&\qquad\qquad \mathit{TrConstructorBody}\ [\![\ N\ ,\ \ F\ ,\ \ \mathit{FieldDecList}\ ,\ \ \mathit{Body}\ ]\!]
\end{aligned}
$$

$$
\mathit{Tr}\ [\![\ \mathit{MethodSpecification{:}MS}\ \ \mathit{Ident{:}N}\ \ \mathit{Formals{:}F}\ \ \mathit{CompoundStatement{:}Body}\ ]\!] \ =
$$

```
procedure 𝔈.N TrSig [[ N , F , void ]] ; TrSpec [[ N , MS ]]
# var FieldDecList = ∅ ;
# for all FieldDeclaration:FD in class 𝔈
   #  FieldDecList = FieldDecList ∪ FD ;
# end for all
implementation 𝔈.N TrSig [[ N , F , void ]]
      TrConstructorBody [[ N , F , FieldDecList , Body ]]
```

Signature translation:

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
```

TrSpec [[*Ident:M* , *MethodSpecification:Specs*]] =

```
requires TrRequires [[ 𝔈 , M ]] ;
ensures TrEnsures [[ 𝔈 , M ]] ;
ModifiesContribution [[ 𝔈 , M ]]
```

TrFreeSpec [[*Ident:M* , *MethodSpecification:Specs*]] =

```
requires TrRequires [[ 𝔈 , M ]] ;
free ensures TrEnsures [[ 𝔈 , M ]] ;
ModifiesContribution [[ 𝔈 , M ]]
```

Pre- and Postconditions

TrRequires ⟦ *Ident:Class* , *Ident:Method* ⟧ = q

```
# MS is method specification of Method in Class
# if method Method is a constructor
   # var req = true
# else
   # var req = this != null
# end if
# for all ( /*@ requires Expression:R; */ ) ∈ MS
      # req && Tr ⟦ R ⟧
# end for all
( req )
```

TrEnsures ⟦ *Ident:Class* , *Ident:Method* ⟧ =

```
# MS is method specification of Method in Class
# if method Method is a constructor
   # var req = true
# else
   # var req = this != null
# end if
# for all ( /*@ requires Expression:R; */ ) ∈ MS
      # req && Tr ⟦ R ⟧
# end for all
# var ens = true;
# for all ( /*@ ensures Expression:E; */ ) ∈ MS
   # ens && Tr ⟦ E ⟧
# end for all
( old(req) ==> (ens
&& ModifiesPart ⟦ 𝔈 , M ⟧))
```

Modifies Clauses

ModifiesContribution ⟦ *Ident:Class* , *Ident:Method* ⟧ =

```
modifies $Heap ;
```

ModifiesPart ⟦ *Ident:Class* , *Ident:Method* ⟧ =

TrModifiesFields ⟦ *Ident:Class* , *Ident:Method* ⟧
&& *TrModifiesArrays* ⟦ *Ident:Class* , *Ident:Method* ⟧

TrModifiesFields ⟦ *Ident:Class* , *Ident:Method* ⟧ =

```
# MS is method specification of Method in Class
(forall $r: ref, $n: name::
   $r != null
   && $isFieldName($n)
   && typeOf($r) <: fieldHome($n)
      ==>
   ownerObject(this) != ownerObject($r)
   || old($Heap[$r, $n]) == $Heap[$r, $n]
   || old(cast ($Heap[$r, alloc], bool)) == false
# var m = ∅ ;
# for all ( /*@ modifies ( StoreRefExpression
         ( , StoreRefExpression )* ) :Mlist ; */ ) ∈ Specs
   # m = m ∪ Mlist;
# end for all
# for all (StoreRefExpression:E ) ∈ m , E denoting a field
   || $r == Tr ⟦ E ⟧
# end for all
# if modifies clause empty or containing \everything
   || true
# end if
   )
```

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

```
# MS is method specification of Method in Class
(forall $start: ref, $arrayRef: ref,
     $n: name, $i: int, $e: elements ::
  $start != null
  && $arrayRef != null
  && old(isAllocated($start, $Heap))
  && old(isAllocated($arrayRef, $Heap))
  && $arrayRef == cast ($Heap[$start, $n], ref)
```

```
   && $isArrayFieldName($n)
   && typeOf($start) <:  fieldHome($n)
   && $i >= 0
   && $i < length($arrayRef)
   && $e == cast ($Heap[$arrayRef, $array$], elements)
      ==>
   ownerObject(this) != ownerObject( refElementSelect($e, $i))
   || old( refElementSelect($e, $i)) == refElementSelect($e, $i)
# var m = ∅ ;
# for all ( /*@ modifies ( StoreRefExpression
         ( , StoreRefExpression )* ) :Mlist ; */ ) ∈ Specs
   # m = m ∪ Mlist;
# end for all
# for all (StoreRefExpression:E ) ∈ m , E denoting an array
   || $arrayRef == Tr [[ E ]]
# end for all
# if modifies clause empty or containing \everything
   || true
# end if
   )
```

Method Body Translation

Assumption: All variables in a method have unique names.

TrBody [[*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:
            AssumeTypes [[ VDs ∪ PMs ]]
            AssumeInvariants [[ this ]];
            Tr [[ Body ]]
            assert INV(this, $Heap);
            # 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 ⟧
      assert INV(this, $Heap);
      return;
  }
```

Let C denote the type in which the method being translated is declared.
AssumeTypes ⟦ *variableBindings* ⟧ =

```
# for all ( var x: BPLType T, BPLName N; ) ∈ variableBindings
  # if T is a ref type then
    assume x != null
      ==> typeOf(x) <: N && isAllocated(x, $Heap);
  # end if
  # if T is peer
    assume x != null
    ==> (ownerObject(x) == ownerObject(this)
      && ownerType(x) == ownerType(this));
  # else if T is rep
    assume x != null
```

```
      ==> (ownerObject(x) == this
        && ownerType(x) == C);
  # end if
# end for all
```
GlobalTypes ⟦ ⟧

GlobalTypes ⟦ ⟧ =

```
assume (forall $x: ref, $n: name::
  ($x != null
  && cast ($Heap[$x, $n], ref) != null
  && $isFieldName($n)
  && typeOf($x) <: fieldHome($n)
    ==>
  typeOf(cast ($Heap[$x, $n], ref)) <: fieldType($n)
  || typeOf(cast ($Heap[$x, $n], ref)) == fieldType($n)));

assume (forall $x: ref, $n: name ::
  cast ($Heap[$x, alloc], bool)
    ==>
  isAllocated(x, $Heap));

assume (forall $a: ref, $n: name, $i: int, $e: elements::
  ($a != null
  && typeOf($a) == arrayType($n)
  && $e == cast ($Heap[$a, $array$], elements)
    ==>
    (refElementSelect($e, $i) != null
      ==>
      typeOf(refElementSelect($e, $i)) == $n))
```

AssumeInvariants ⟦ Ident: I ⟧ =

```
assume (forall $r: ref ::
  ownerObject(I) == ownerObject($r)
    ==>
  INV(I, $Heap));
assume(forall $r : ref ::
  isTOwnedBy($r, I)
```

```
    ==>
INV($r, $Heap));
```

B.2.5. Statements

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

Tr ⟦ ; ⟧ = ∅

Variable Declaration

The types of all variables are specified at the beginning of each method.

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

Method Call

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

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
assert INV(this, $Heap);
# 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
   );
AssumeInvariants [[ receiver ]]
```

Assignments

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;
Assign [[ i , Rhs ]]
$Heap[e, C.I] := i;
```

For a 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
   $Heap[e, $array$] := boolElementStore(ElementsHeap[e], n, f);
# if type T is int
   $Heap[e, $array$] := intElementStore(ElementsHeap[e], n, f);
# else
   $Heap[e, $array$] := refElementStore(ElementsHeap[e], n, f);
# end if
```

Assignment to a local variable:

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

```
Assign[[ Ident:I , Expression:E ]] = assert DefCk[[ E ]];
                                     I := Tr[[ E ]];
```

New object creation: if the expression list EL is empty, then the default constructor is called without any additional expressions. Assigning a new class instance with an owner declaration to a local identifier I which is of type *ReferenceType*.
Let C denote the class in which the current method was declared.

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

```
havoc I;
assume I  != null;
assume typeOf(I) == Type ⟦ T ⟧;
# if T is peer
   assume ownerObject(I) == ownerObject(this)
      && ownerType(I) == ownerType(this);
# else if T is rep
   assume ownerObject(I) == this
      && ownerType(I) == C;
# end if
assume (forall n:  name ::  cast ($Heap[I, n], ref) == null);
assume (forall n:  name ::  cast ($Heap[I, n], bool) == false);
assume (forall n:  name ::  cast ($Heap[I, n], int) == 0);
$Heap[I, alloc] := true;
# for all e ∈ EL
   assert DefCk ⟦ e ⟧;
# end for all
assert INV(this, $Heap);
call T.T  ( I
# for all e ∈ EL
   , Tr ⟦ e ⟧
# end for all
);
AssumeInvariants ⟦  this  ⟧
```

Assigning a new array instance with an owner declaration to a local identifier I. Note that for the items of the array, no Universe type assumptions are made because they are all set null. Null has no owner.

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

```
assert DefCk ⟦ E ⟧;
havoc I;
# generate symbol n
# ↪ ( var n:  BPLType int, BPLName basicInt; )
# if T is peer
   assume ownerObject(I) == ownerObject(this)
      && ownerType(I) == ownerType(this);
# else if T is rep
   assume ownerObject(I) == this
```

```
      && ownerType(I) == C;
# end if
n := Tr [[ E ]];
assert 0 <= n;
assume I != null;
assume typeOf(I) == arrayType(Type [[ T ]]);
assume length(I) == n;
# if type T is boolean
   assume (forall i: int ::
      boolElementSelect(cast ($Heap[I, $array$], elements), i)
      == false);
# if type T is int
   assume (forall i: int ::
      intElementSelect(cast ($Heap[I, $array$], elements), i)
      == 0);
# else
   assume (forall i: int ::
      refElementSelect(cast ($Heap[I, $array$], elements), i)
      == null);
# end if
$Heap[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*.

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
assert INV(this, $Heap);
# N is defined in Class C
call I  := C.N ( receiver
# for all e ∈ EL
   , Tr ⟦ e ⟧
# end for all
);
AssumeInvariants ⟦ receiver ⟧
AssumeTypes ⟦ T I ⟧
```

Control Flow Statements

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

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

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

```
assert DefCk [[ C ]];
# generate symbol then, else, join
goto then, else;
then:
   assume Tr [[ C ]];
   Tr [[ T ]];
   goto join;
else:
   assume !  Tr [[ C ]];
   goto join;
join:
```

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 function F with a return value.

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

Returning from a function without a return value.

Tr ⟦ `return;` ⟧ = `return;`

Specification Statements

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

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

B.2.6. Expressions

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* ⟧ =
E `!= null ==>` *typeOf*`(`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 ⟧

$Tr\ [\![\ \texttt{(}\ \mathit{Type}\ \texttt{)}\ \mathit{UnaryExpression{:}E}\ ]\!] = Tr\ [\![\ E\ ]\!]$

$Tr\ [\![\ \mathit{DereferenceExpression{:}E}\ \texttt{.length}\ ]\!] = \mathit{length}\texttt{(}Tr\ [\![\ E\ ]\!]\texttt{)}$

$Tr\ [\![\ \mathit{DereferenceExpression{:}E}\ \texttt{.}\ \mathit{Ident{:}I}\ ]\!] =$

```
# if type of I is boolean
   cast ($Heap[Tr [| E |], C.I], bool)
# if type of I is int
   cast ($Heap[Tr [| E |], C.I], int)
# else
   cast ($Heap[Tr [| E |], C.I], ref)
# end if
```

$Tr\ [\![\ \mathit{DereferenceExpression{:}E}\ \texttt{[}\ \mathit{Expression{:}F}\ \texttt{]}\ ]\!] =$

```
# elements of array E are of type boolean
   boolElementSelect(ElementsHeap[Tr [| E |]], Tr [| F |])
# elements of array E are of type int
   intElementSelect(ElementsHeap[Tr [| E |]], Tr [| F |])
# else
   refElementSelect(ElementsHeap[Tr [| E |]], Tr [| F |])
# end if
```

Tr ⟦ *Ident:I* ⟧	=	*I*	
Tr ⟦ *Literal:L* ⟧	=	*L*	only integers
Tr ⟦ `super` ⟧	=	$\emptyset$	

handled in function calls directly

Tr ⟦ `true` ⟧	=	`true`	
Tr ⟦ `false` ⟧	=	`false`	
Tr ⟦ `this` ⟧	=	`this`	
Tr ⟦ `null` ⟧	=	`null`	
Tr ⟦ `(` *Expression:E* `)` ⟧	=	`(`*Tr* ⟦ *E* ⟧`)`	

Tr ⟦ `\old(` *Expression:E* `)` ⟧	=	*`old`*`(`*Tr* ⟦ *E* ⟧`)`	
Tr ⟦ `\typeof(` *Expression:E* `)` ⟧	=	*`typeOf`*`(`*Tr* ⟦ *E* ⟧`)`	
Tr ⟦ `\type(` *Type:T* `)` ⟧	=	*Type* ⟦ *T* ⟧	
Tr ⟦ `\result` ⟧	=	*N*`.return`	

N is name of function

Tr ⟦ `\ownerobject(` *Expression:E* `)` ⟧	=	*`ownerObject`*`(`*Tr* ⟦ *E* ⟧`)`	
Tr ⟦ `\ownertype(` *Expression:E* `)` ⟧	=	*`ownerType`*`(`*Tr* ⟦ *E* ⟧`)`	

Checking for Definitions

The following translation rules to check whether a given *JML* expression is valid or not are implemented by the class
DefinitionChecker which is part of the package org.jtobpl.translation.

DefCk ⟦ *ImpliesExpression:E EquivalenceOp ImpliesExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧

DefCk ⟦ *LogicalOrExpression:E* ==> *LogicalOrExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && (*Tr* ⟦ *E* ⟧ ==> *DefCk* ⟦ *F* ⟧)

DefCk ⟦ *LogicalOrExpression:E* <== *LogicalOrExpression:F* ⟧ = *DefCk* ⟦ *F* ⟧ && (*Tr* ⟦ *F* ⟧ ==> *DefCk* ⟦ *E* ⟧)

DefCk ⟦ *LogicalAndExpression:E* || *LogicalOrExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && (*Tr* ⟦ *E* ⟧ || *DefCk* ⟦ *F* ⟧)

DefCk ⟦ *EqualityExpression:E* && *LogicalAndExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && (*Tr* ⟦ *E* ⟧ ==> *DefCk* ⟦ *F* ⟧)

DefCk ⟦ *RelationExpression:E EqualityOp RelationExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧

DefCk ⟦ *RelationExpression:E* instanceof *Type:T* ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ *AdditiveExpression:E RelationOp AdditiveExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧

DefCk ⟦ *MultExpression:E* (+ | -) *AdditiveExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧

DefCk ⟦ *UnaryExpression:E* * *MultExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧

DefCk ⟦ *UnaryExpression:E* (/ | %) *MultExpression:F* ⟧ = *DefCk* ⟦ *E* ⟧ && *DefCk* ⟦ *F* ⟧ && (*Tr* ⟦ *F* ⟧ != 0)

DefCk ⟦ (- | !) *UnaryExpression:E* ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ (*Type:T*) *UnaryExpression:E* ⟧ = *DefCk* ⟦ *E* ⟧ &&
(*typeOf*(*Tr* ⟦ *E* ⟧) <: *T*) &&
#if T is peer: *ownerObject*(*Tr* ⟦ *E* ⟧) == *ownerObject*(this) && *ownerType*(*Tr* ⟦ *E* ⟧) == C
#if T is rep: *ownerObject*(*Tr* ⟦ *E* ⟧) == this && *ownerType*(*Tr* ⟦ *E* ⟧) ==

ownerType(this)

DefCk ⟦ *DereferenceExpression:E* . *Ident:I* ⟧ = *DefCk* ⟦ *E* ⟧ && (*Tr* ⟦ *E* ⟧ != null)

DefCk ⟦ *DereferenceExpression:E* [*Expression:F*] ⟧ = *DefCk* ⟦ *E* ⟧ && (*Tr* ⟦ *E* ⟧ != null) && *DefCk* ⟦ *F* ⟧ && (0 <= *Tr* ⟦ *F* ⟧) && (*Tr* ⟦ *F* ⟧ < *length*(*Tr* ⟦ *E* ⟧))

DefCk ⟦ *Ident* | *Literal* ⟧ = true

DefCk ⟦ super | true | false | this | null ⟧ = true

DefCk ⟦ (*Expression:E*) ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ \old(*Expression:E*) ⟧ = *old*(*DefCk* ⟦ *E* ⟧)

DefCk ⟦ \typeof(*Expression:E*) ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ \type(*Type*) ⟧ = true

DefCk ⟦ \result ⟧ = true

DefCk ⟦ \ownerobject(*Expression:E*) ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ \ownertype(*Expression:E*) ⟧ = *DefCk* ⟦ *E* ⟧

DefCk ⟦ \everything ⟧ = true

DefCk ⟦ \nothing ⟧ = true