Integration of a new VCGen in ESC/Java2

Claudia Brauchli

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

September 2007

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

"*The earlier errors are found, the less costly they are to fix!*"[1] Code consumers make high demands on software applications. By providing a certificate associated with a program, the code producers can guarantee the properties the consumer demands on.

The project team of MOBIUS is developing techniques to verify security and functional properties of Java programs in order to generate certificates. Based on ESC/-Java2, an already existing static checker, the new environment of MOBIUS produce proof obligations of difficult security properties that have to be discharged manually, using Coq.

In order to achieve this, we integrate a new direct Verification Condition generator (VCGen) into ESC/Java2. This master thesis covers the work needed to integrate this VCGen. We developed the translations of code specifications into first order logic terms. Code specifications, to specify the functional behavior of a particular program, are annotations written in Java Modeling Language (JML). We have defined and implemented the translation of most of the JML level 0 annotations into first order logic terms.

[1] Preamble of article [1].

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 Aims of this Thesis . . . 7
- 1.3 Acronyms . . . 7
- 1.4 Tools and Languages . . . 8

2 Preliminaries **9**
- 2.1 ESC/Java2 . . . 9
  - 2.1.1 JML . . . 11
  - 2.1.2 AST . . . 15
  - 2.1.3 Using ESC/Java2 . . . 16
  - 2.1.4 JavaFE . . . 19
- 2.2 Sorted Logic . . . 20
- 2.3 Pre- and Poststate . . . 21
- 2.4 Heap . . . 21
- 2.5 Coq . . . 22

3 Integration of a new VCGen **23**
- 3.1 General Concept . . . 24
  - 3.1.1 Example of a Translation . . . 24
- 3.2 Translation Features . . . 25
  - 3.2.1 Property Object . . . 25
  - 3.2.2 Lookup Table . . . 27
  - 3.2.3 Annotation Table . . . 28
  - 3.2.4 Visible States . . . 29
  - 3.2.5 Dependences of Fields in Type Specification . . . 29
- 3.3 Definition of FOL Terms . . . 30
  - 3.3.1 Numerical Operations . . . 30
  - 3.3.2 Boolean Expressions . . . 31
  - 3.3.3 Predicates . . . 32
  - 3.3.4 Special Notations . . . 33
- 3.4 JML Statements . . . 35
  - 3.4.1 Assert, Assume, Set . . . 36
  - 3.4.2 Maintaining . . . 36
  - 3.4.3 Example of JML Statements . . . 37
- 3.5 Method Specifications . . . 39
  - 3.5.1 Requires . . . 39

3.5.2 Ensures . . . 41
3.5.3 Signals_only . . . 43
3.5.4 Signals . . . 45
3.5.5 Assignable . . . 46
3.6 JML Modifiers . . . 49
3.6.1 Ghost . . . 49
3.6.2 Helper . . . 50
3.7 Quantifiers . . . 50
3.7.1 Forall . . . 51
3.7.2 Exists . . . 52
3.8 JML Expressions . . . 53
3.8.1 Result . . . 53
3.8.2 Fresh . . . 54
3.8.3 Old . . . 55
3.8.4 Typeof . . . 56
3.8.5 Type . . . 57
3.9 Type Specifications . . . 58
3.9.1 Initially . . . 58
3.9.2 History Constraint . . . 59
3.9.3 Example of History Constraint . . . 60
3.10 Invariant . . . 61
3.10.1 Object Invariant . . . 61
3.10.2 Invariant Term for Precondition Φ . . . 62
3.10.3 Invariant Term for Postcondition Ψ . . . 64
3.10.4 Type Collector . . . 65
3.11 JML Subset Checker . . . 70
3.11.1 Example of Subset Checker . . . 70
3.12 Summary of Translations . . . 71

4 Conclusion and Future Work 72
4.1 Conclusion . . . 72
4.2 Future Work . . . 72
4.2.1 Implementation of missing JML level 0 Translations . . . 72
4.2.2 Assignable Semantics . . . 72
4.2.3 Different Frontend/Backend . . . 73
4.2.4 Acknowledgements . . . 73

A Translation Example of most JML Level 0 Features 75

B JML Predicate Syntax 82

C Expression Syntax 84

1 Introduction

Code consumers expect some properties, such as type and memory safety, which has to be satisfied during code execution. For that reason, code producers provide a certificate that indicates these properties of their code. A certificate of simple program properties can be done automatically from source code. In the case of advanced security properties, there are additional techniques necessary in order to produce certificates. Based on these facts, the project MOBIUS (**Mobi**lity, **U**biquity, and **S**ecurity) [2] was founded by a consortium of sixteen partners and is coordinated by INRIA [3]. The MOBIUS project has started in September 2005 and lasts for 48 months. One of the partners is the Software Component Technology Group [4] at the Swiss Federal Institute of Technology Zurich [5] where this master thesis is written.

The project's goal is to develop modular logical techniques for the verification of security and functional properties of Java programs. A new environment will be developed that supports these techniques for program verification on source and byte code level. This environment will be based on ESC/Java2 [6], an already existing static checker. Specifications are added to the code as annotations to express pre- and postconditions, invariants, and other properties. These annotations will be written in a standard annotation language, JML [7].

The aims of the group at the Swiss Federal Institute of Technology Zurich are the creation of techniques and tools for the development of provably correct object-oriented software components.
This master thesis covers the integration of a new direct Verification Condition generator (VCGen) into ESC/Java2 on source code level. The new VCGen is related to the already existing VCGen on byte level [8] and generates Verification Conditions (VCs) for a manual theorem prover (e.g. Coq [9]). The old VCGen requires Guarded Commands (GCs) as input data in order to generate VCs for an automatic theorem prover (e.g. Simplify [10]). We disclaim the intermediate language of GCs and reuse the data structure given by the Java frontend (JavaFE) of ESC/Java2. This keeps the proving mechanism as simple as possible regarding the proof will be done by hand.

1.1 Motivation

The project goal is to use the manual theorem prover Coq. As it is a manual tool, the generated Verification Conditions (VCs) have to be human readable as much as possible. We disclaim any intermediate language as Guarded Commands (GCs) to keep the VCs more related to the verifying Java source code. Thus, the new VCGen is also called "direct VCGen", since we avoid any intermediate language. Using a sorted logic, we keep the Java types of fields and variables until the generation of VCs.

It already exists a VCGen on byte code level. The project goal is to integrate a similar VCGen on source code level.

1.2 Aims of this Thesis

1. Define the transformation of JML level 0 to first-order logic.
2. Implement four lookup functions to get information about the pre-, post-, and exceptional postconditions of a method, and about the local assertions inside a method.
3. Design and implement a Java and JML subset checker, which can be turned on and off.
4. Design and implement static analysis that checks which object can be changed within a method (for invariant checking).
5. Supporting history constraints.

Possible extensions are:

- Verifying some simple example codes in Coq
- Support ownership structure to check invariants

1.3 Acronyms

Table 1.1 contains a list of acronyms that are frequently used in this thesis.

AST	Abstract Syntax Tree
ESC	Extended Static Checking
FE	Front End
FOL	First-Order Logic
GCs	Guarded Commands
JavaFE	Java Front End
JML	Java Modeling Language
PVE	Program Verification Tool
VCGen	Verification Condition Generator
VCs	Verification Conditions

Table 1.1: Table with acronyms

1.4 Tools and Languages

The following tools and languages are used:

Java: Object-oriented high-level programming language developed by Sun Microsystems [11]
JRE: Java Runtime Environment [12] 1.42 (for ESC/Java2)
JRE: Java Runtime Environment [12] 1.50 (for MOBIUS)
ESC/Java2: Tool to prove correctness of specifications at compile time [6]
Simplify: Automatically theorem prover [10]
Coq: Manual theorem prover [9]

2 Preliminaries

This chapter gives an overview of the verification environment ESC/Java2 and its usage. The first section 2.1 covers the operation steps in ESC/Java2. A Java source code under verification gets parsed by the JavaFE and gets transmitted to an Abstract Syntax Tree (AST), which is explained in section 2.1.2. In section 2.2, we give a brief introduction to sorted logic that we use for the translation of JML to FOL terms. Pre- and poststates in section 2.3 are relevant for the heap handling, described in section 2.4. The last section 2.5 describes the manual prover Coq, which is used in this project.

2.1 ESC/Java2

ESC/Java2 is a programming tool to check program specifications. It detects, at compile time, common programming errors that are not ordinarily detected until run time, and sometimes not even then. Common run time errors are division by zero, null dereference errors, array bound errors and type cast errors.
ESC/Java2 is an extension of ESC/Java[1] and supports more of JML features. ESC is the acronym for Extended Static Checking. The tool extends conventional static checkers by catching more errors than they do and it is static because the program does not have to run in order to get checked. ESC/Java2 computes VCs from a JML annotated Java program. VCs are made up of a set of logical formulas. Later on, these generated VCs are proven by a theorem prover, currently Coq.

Figure 2.1 illustrates the operation of ESC/Java2 step by step. During these operation steps, the Java code under verification gets converted into three other data structures as listed below:

JML annotated Java code: This is the main Java program under verification using ESC/Java2. It is a plain Java code including JML features for specifying the code.

AST: An AST represents a logical tree structure. It expresses the content of a given document in form of nodes and leaves. An AST is the result of parsing Java codes by JavaFE. Using the JavaFE of ESC/Java2, JML annotated Java codes can be parsed as well.

GCs: Guarded Commands are imperative and intermediate representations of the original annotated Java code. GCs are tailored towards verification. They consist of some simple instructions. We mainly have assertions, assumptions and program flow instructions.

[1] Developed by the Compaq Systems Research Center, today HP.

VCs: VCs are conditions that have to be met in order to verify a program. Each method holds one weakest precondition, which is calculated from the given GCs. A VC describes the weakest precondition under the assumption that the class predicate is satisfied. A theorem prover (e.g. Simplify) tries to proof these VCs.

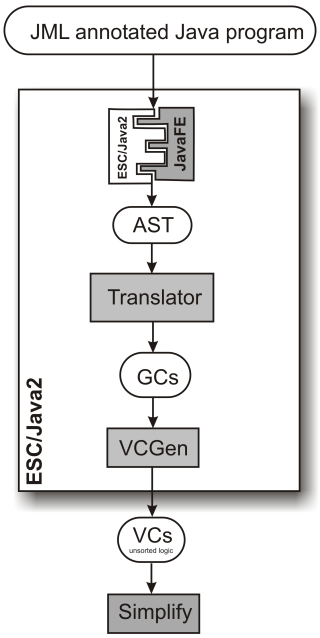

Figure 2.1: Operation flow of ESC/Java2

There are three operations within ESC/Java2 to generate the individual data structure. A brief introduction to each operation is given below:

JavaFE: The input of JavaFE is a JML annotated Java code, written by the user. JavaFE parses the code and transforms it into an AST. The output of JavaFE and ESC/Java2 is a full structured AST containing JML annotations.

Translator: The translator takes the AST and generates GCs.

VCGen: VCGen uses a weakest precondition calculus. VCGen takes GCs as input and generates VCs. The VCGen used in ESC/Java2 generates VCs in unsorted logic.

Simplify: Simplify is an automatically theorem prover. It tries to proof the generated VCs in unsorted logic. If it finds a counter example, which violates these assumptions, an error message is returned.

2.1.1 JML

JML is a language to specify the functional behavior of Java classes and methods. Thus, the quality of the Java source code will be increased. The Java programmer can express the design decision of the program by adding JML features to the Java code. Run time exceptions can be caught by using JML specifications as for example pre-, postconditions, and invariants. These JML features appear as comments and are ignored by Java compilers. Only the ESC/Java2 tool makes use of them. A one line JML feature starts with the prefix "//@", which is appended with a JML annotation:

```
//@ <JML-annotation>;
```

For several JML annotation lines, one can use "/*@" to begin and "@*/" to end the annotation lines. The following example shows two methods. The first method could throw a **DivideByZeroException**, whereas the second method prevents this exception by using a JML feature to avoid that exception case:

```
public void doSomethingBad(int val) {
  int test = 100 / val;
}

//@ requires val != 0;
public void doSomethingGood(int val) {
  int test = 100 / val;
}
```

To avoid a **DivideByZeroException**, the second method uses the `requires` JML feature. It limits the range of the input values by allowing values not equal to zero. Thus, the second method never will throw a **DivideByZeroException**.

JML annotations are subdivided into several levels. The next section provides the reader with a tutorial to JML level 0 and to the upper level. More information about JML is described in [13].

JML Level 0

JML level 0 constitutes the heart of JML. JML features in this level are fundamental to all usages of JML and should be covered in all tools. This thesis covers the translation of the most JML level 0 annotations. These annotations are categorized into same specification types. We give a short explanation to each category and each level 0 feature and refer to [14] for full explanation. The syntax of all JML features is given in appendix B. This master thesis does not cover the design and implementation of all JML level 0 features. See comments in brackets at the end of each explanation whether or not it is covered.

JML modifiers:
A class, interface, method, or a class field may be modified using one of the JML modifiers.

- `Model` fields are abstractions of concrete fields. They are only present for specification purposes and are not available for use in Java code outside of annotations. `Model` fields hold the value of the concrete fields they abstract of, using the JML feature `represent`. *(Not covered)*
- `Ghost` fields and variables are similar to `model` fields. They are also present for purposes of specification only, but they are not abstractions of concrete fields. `Ghost` variables are statement annotations and interspersed to Java statements. The value of `ghost` fields and variables are directly determined by their initializations or by using the JML feature `set`. *(`Ghost` variables are covered)*
- `Instance` modifiers express that the corresponding fields are not static. They are mostly used for `ghost` fields in interfaces. *(Not covered)*
- `Spec_public` is a way to change the visibility of a feature for Java and JML purposes (e.g. public, protected and private). Spec_public declares any field as public for JML purposes without any regards to the Java visibility. *(Not covered)*
- `Spec_protected` is similar to `spec_public`, but it declares any class field as protected for JML purposes. *(Not covered)*
- `Nullable` is a way to allow null references. By default, declarations, whose types are reference types, are implicitly declared as non_null references. Using the JML feature `nullable`, one can explicitly allow declaration without yet being instantiated. *(Not covered)*
- `Non_null` declares a value or reference not to be null. *(Not covered)*
- `Helper` modifiers state that the `invariant`, `initially`, and `history constraint` are not relevant to the private method or constructor, which is modified with this JML feature. The next example demonstrates the usage of the `helper` JML feature on a private method: *(Covered)*

```
private /*@ helper */ void doSomething() {..}
```

Type specifications: The following type specification features must hold within the whole class or interface definition. They are relevant to an object type.

- `Invariants` have to hold in every visible state of the type. They are implicitly included in all pre-, post-, and exceptional postconditions. *(Covered)*
- `Represents` clauses can be used to assign the value of a concrete field to a `model` field. *(Not covered)*

- `Initially` features can be used to state the behavior of each concrete subtype after its instantiating. `Initially` predicates are applied to post- and exceptional postconditions of all non-helper constructors. *(Covered)*
- `Type` feature denotes the type of the holding reference or primitive type. *(Covered)*

Method specifications: The behavior of any method can be described by using method specifications.

- `Requires` annotations represent the precondition of a method or constructor. Any precondition must hold at the beginning of the routine execution. *(Covered)*
- `Ensures` annotations represent the postcondition of a method or constructor. Any postcondition must hold when the routine terminates without throwing any exception. *(Covered)*
- `Signals` annotations specify the exceptional postconditions. The property of the `signals` clause has to hold when the method or constructor execution terminates abruptly by throwing a given exception. The predicate P of the `signals` clause in the next example has to hold if an exception of type E is thrown: *(Covered)*

```
//@ signals (E e) P;
```

- `Signals_only` annotations declare all exceptions that may be thrown by a routine. *(Covered)*
- `Assignable` JML features contain a list of object references that are assignable during the execution of the method's or constructor's body. All other objects have to remain unchanged.*(Covered)*

Statements: JML statements are interspersed with Java statements in the body of any method or constructor.

- `Assert` statements require the given predicate to hold at the given point in the program execution. JML has to check that the specified predicate is *true*. *(Covered)*
- `Assume` statements specify that the given predicate holds. Static analysis tools do not need to check if the predicate holds. The `assume` statement expresses that the given predicate is assumed to hold. *(Covered)*
- `Set` statements are used to assign values to `ghost` variables or to `ghost` fields. *(Covered)*
- `Maintaining` is a `loop invariant`. It has to hold before and after each loop iteration. *(Covered)*

Predicates: Some JML annotations can be extended by using JML predicates.

- `Result` predicate refers to return value of non-void methods. It can only be used in `ensure` clause. *(Covered)*
- `Old` predicate refers to values of the given expression in the prestate of a method or constructor. *(Covered)*
- `Fresh` predicate assert that the objects in the holding list will be freshly allocated in the body of that routine declaring this JML feature. They are declared but not allocated in the prestate. *(Covered)*
- `Typeof` features return the most-specific dynamic type of the given expressions. *(Covered)*

Quantors: JML offers the usage of the known logical quantors in any JML feature.

- `Forall` is the universal quantifier. It ranges over all potential values of the declared variable. The range predicate is given between the semicolons ";". The next quantified expression demonstrates the usage of the universal quantifier. This predicate requires a given integer array "a" to hold values different to the integer value of two. We range the scope to the first ten entries of the array. *(Covered)*

  (\forall int i; i <= 10; a[i] != 2)

- `Exists` is the existential quantifier. It has the same syntax as the universal quantifier. *(Covered)*

Operands:

- < , > , <=, >= ; ==, != *(Covered)*
- <==> , <=!=> *(Not covered)*
- <==, ==> *(==>: Covered)*
- \max, \min, \product, \sum, \num_of *(Not covered)*

Upper levels

To complete the JML overview, we give a brief introduction to the upper JML levels. From the bottom-up there exist level 1, level 2, level 3, level C, and level X features. They are more exotic features and not implemented by many tools.

`Pure:` This JML modifier can be applied to methods. Such a method does not modify any field location and thus, it is without any side-effects. A pure method has the same meaning as the JML annotation "**//@ assignable \nothing**". Pure denoted methods can be used themselves in JML `invariant` features since they result in the same in every execution. *(Not covered)*

`History constraints`: `History constraints` are related to `invariants` and constrain the way in which values may change in the program executions. `History constraints`, simply named as `constraints`, typically use the JML feature `old`. The example below shows a `constraint` JML feature of an integer "c" that can only be increased in the whole lifetime of the object declaring it. *(Covered)*

```
//@ constraint c >= \old(c);
```

2.1.2 AST

An AST is a logical tree structure consisting of nodes and leaves. The JavaFE generates an AST by parsing an entire Java code. Figure 2.2 shows the meta model of the AST syntax used in ESC/Java2. Due to lack of space, we only display the most important nodes.

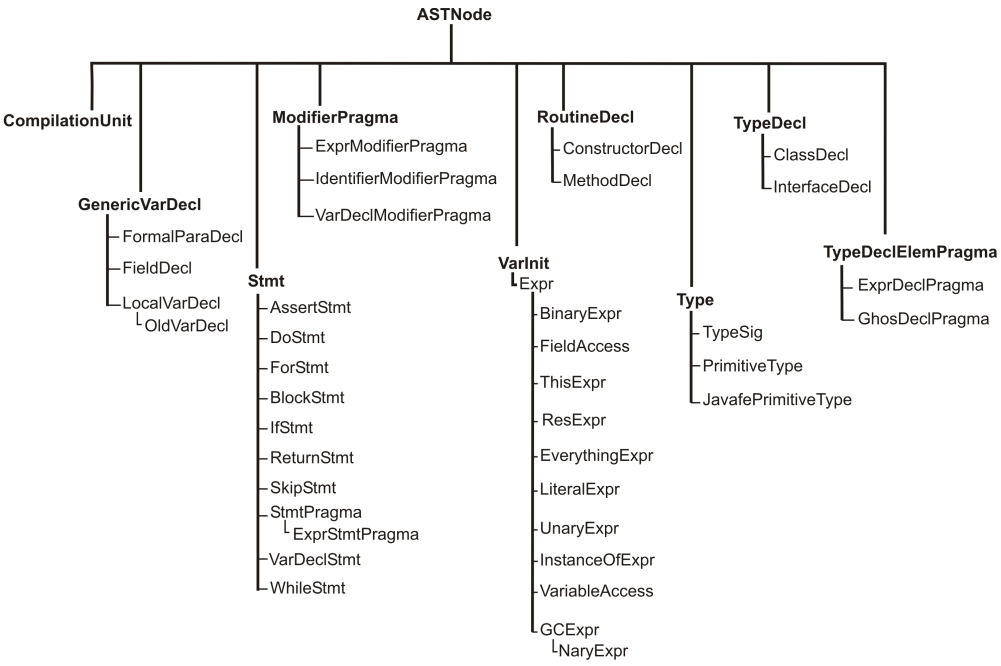

Figure 2.2: Meta model of AST syntax in ESC/Java2

The following listing gives a brief explanation on the most relevant nodes for our usage:

CompilationUnit: There exists exactly one compilation unit for each verifying program. It acts as the root of the generated AST tree and contains a list of type specification as well as other elements as for example class declarations.

GenericValDecl: Any variable declaration (either local or field variables) is represented as a generic variable declaration node.

ModifierPragma: Every modification of a routine, field or variable, is represented as a modifier annotation.

VarInit: Initializing a variable is done in a *VarInit* node.

Stmt: The body of a routine consists of statements, represented as *Stmt* nodes. A local variable declaration *LocalVarDecl* is represented as a statement node *VarDeclStmt*.

RoutineDecl: A routine declaration is either a method or a constructor declaration. It contains method specifications as **ModifierPragma** nodes and the routine's body, containing Java and JML statements.

TypeDecl: *TypeDecl* node represents a new declared type or interface.

To scan such an AST, we use the visitor design patterns. ESC/Java2 provides interfaces to implement visitors.

Example of an AST

We demonstrate the generation of an AST in the following example. JavaFE generates the AST of the Java code as illustrated in figure 2.3:

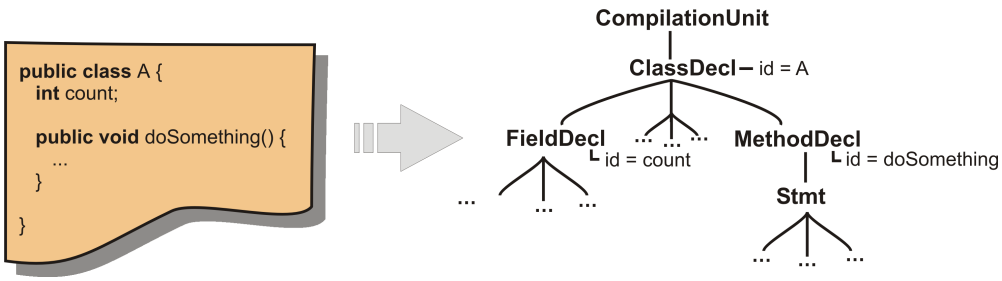

Figure 2.3: Java code and generated AST

2.1.3 Using ESC/Java2

ESC/Java2 tries to find common runtime exceptions at compile time by doing a static analysis of the Java source code. This section demonstrates the usage of JML annotations by running ESC/Java2.
One common runtime exception is the division by zero. ESC/Java2 detects these kinds of runtime errors and warns the programmer. The following example shows a class with one constructor and one method.

```
1  public class A {
2    int count;
3    boolean[ ] boolArray;
4
5    public A(int num) {
6      this.count = num;
7      boolArray = new boolean[num];
8      for (int i = 0; i < boolArray.length; i++){boolArray[i]=true;}
9    }
10
11   public void foo(int x) {
12     this.count = this.count / x;
13   }
14 }
```

The constructor initializes the array **boolArray** by setting the entire array to *true*. Calling the method "foo()" with an input integer value **x**, the method divides the count value by the value of **x**. ESC/Java2 is a modular checker. It checks each routine separately.
A routine either passes the proof checker, or the occurred warning will be printed out with the specific line number. Running ESC/Java2 on the Java class "A", the following warnings are printed out:

```
A.java 7: Warning: Possible attempt to allocate array of negative length (NegSize)
      boolArray = new boolean[num];
                                  ↑

A.java 12: Warning: Possible division by zero (ZeroDiv)
      this.count = this.count / x;
                               ↑
```

The first warning occurs because the constructor could be called with a negative input value. To prevent this possible runtime exception, we have to provide input values greater than zero:

```
//@ requires num > 0;
```

The second possible exception can be caught by restrict the possible input value. We allow only input values not equal to 0. We state this behavior by using the following JML annotation:

```
//@ requires x != 0;
```

Both JML annotations represent a precondition for each routine. ESC/Java2 assumes that preconditions holds on entry.

The next example shows the same Java class but extended with the JML annotations on line 5 and 12 to avoid the given warnings:

```
1  public class A {
2    int count;
3    boolean[ ] boolArray;
4
5    //@ requires num > 0;
6    public A(int num) {
7      this.count = num;
8      boolArray = new boolean[num];
9      for (int i = 0; i < boolArray.length; i++){boolArray[i]=true;}
10   }
11
12   //@ requires x != 0;
13   public void foo(int x) {
14     this.count = this.count / x;
15   }
16 }
```

The following output will be printed out after running the new JML annotated Java code:

```
A: A(int) ... passed

A: foo(int) ... passed
```

The A.java example passes without any warnings after adding two JML requirement annotations.

Behind the scene:
To accomplish this modular checking, an AST of the Java code will be generated. This AST includes nodes for classes, routines, and nodes for JML annotations. Figure 2.4 shows the AST of the Java class "A" with JML annotations. The two **ModifierPragma** nodes represent the preconditions of the constructor and of the "foo()" method.

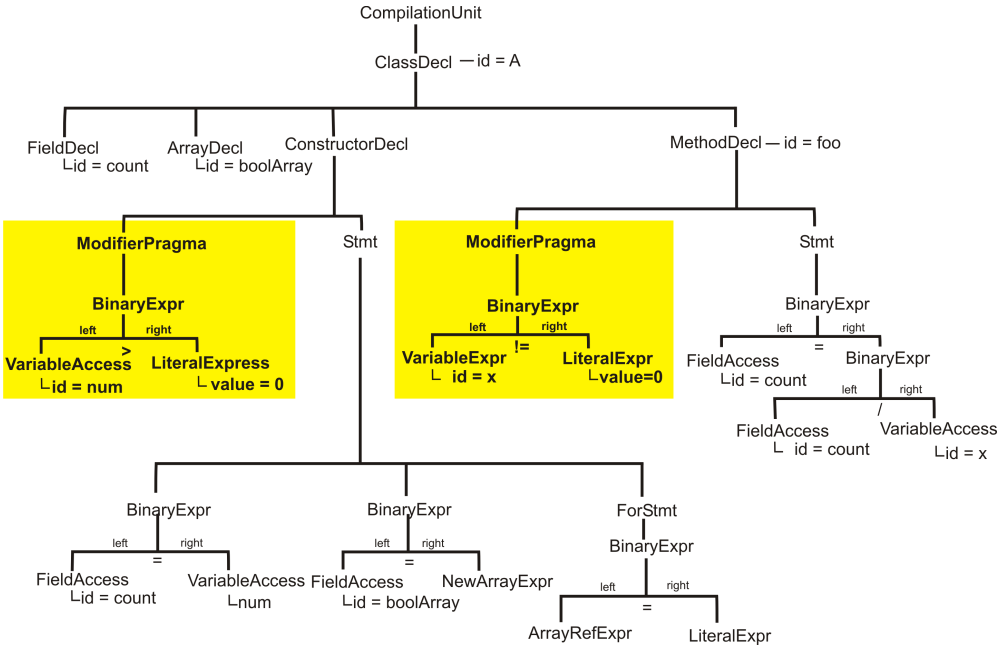

Figure 2.4: Fragment of the AST that JavaFE generates of the Java code of class “A”.

2.1.4 JavaFE

The frontend of ESC/Java2 consists of two parts. The first one is the Java Frontend (JavaFE), which parses a Java code and checks the contained types. The second part is ESC/Java, which extends the JavaFE in order to use JML annotations and to do more complex static checks. The frontend of ESC/Java2 operates in three phases as illustrated in figure 2.5:

- **Lexer:** Translating symbols to known tokens
- **Parser:** Creating an AST
- **Typechecking:** Syntactically and semantically type checking

JML annotations are treated as proper nodes, called **Pragmas**. JavaFE combined with ESC/Java2 generates an AST containing these pragmas.

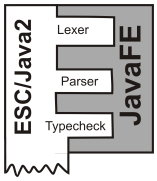

Figure 2.5: ESC/Java2 and JavaFE as Java frontend.

2.2 Sorted Logic

Sorts are abbreviations for a collection of universe types. One fundamental objective of this project is to obtain Java types of a Java code until the proving mechanism, independent of any translation. Hence, we must do the translation for every JML annotation dependent on its type. FOL terms are based on sorts instead of Java types. ESC/Java2 offers a method to convert Java types to sorts. Table 2.1 shows a small insight of the mapping defined in ESC/Java2 between the most useful Java types and sorts.

Java Type	**Sort**	**As String**
BOOLEANTYPE	`sortBool`	bool
DOUBLETYPE	`sortReal`	real
FLOATTYPE		
BYTETYPE	`sortInt`	int
SHORTTYPE		
INTTYPE		
CHAR		
LONG		
BIGINT		
TYPESIG	`sortRef`	ref
TYPENAME		
TYPECODE		
NULLTYPE		
ANY	`sortAny`	any
Value	`sortValue`	value

Table 2.1: Mapping from Java types to sorts

For more flexibility of generating FOL terms, we use some more sorts as represented in table 2.2.

Sort	As String	Represents
sortPred	PRED	A predicate
sortAny	any	Any possible sort
sortValue	value	A value
sortMap	map	The type of heap

Table 2.2: Additionally sorts

What is the difference between sortBool and sortPred?
Both sortBool and sortPred represent a boolean type. Every predicate[2] is defined as sort sortPred whereas every boolean literal (actually *true* and *false*) is defined as sort sortBool. There are operations that either results in sortBool or sortPred, depending on the operands sort.

2.3 Pre- and Poststate

The state before a constructor's or method's execution is called prestate. After executing a routine, the state is called poststate. Thus, every constructor and method has one pre- and one poststate. The most JML features either have to hold in a routine's prestate, poststate, or even both. We use a precondition Φ term to state what must hold in the prestate of a routine. The behavior of the poststate is represented by the normal postcondition Ψ and the exceptional postcondition Ψe.

2.4 Heap

The heap is an area of the main memory. It is used as free storage for dynamic memory allocation in a computer program during runtime. It is always accessible and thus, it represents all allocated objects at the point of access. There are three main functions applying on a heap:

select: selecting a variable on the heap

store: storing a variable on the heap

loc: representing the location of a variable within the heap

As explained in section 2.3, there are two different states in one routine. Because we need access to the prestate and the current state during the execution of a routine, there are given two different heaps:

pre_heap: A copy of the current heap representing the heap in the routine's prestate. This heap is only valid for one routine's execution.

[2] See section 3.3.3 for more details about predicates.

heap: The current heap, valid for the entire execution of the program, contains all values for each program point.

Figure 2.6 shows a method and the two heaps associated with it. The pre_heap holds values of fields in the prestate whereas the common heap holds values of fields of the entire execution of the routine's body.

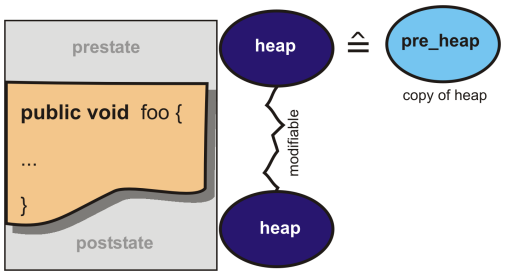

Figure 2.6: Pre_heap and heap during a method's execution

We have always access to both heaps during the execution of this method. After and before the execution of any method the pre_heap is not available anymore.

2.5 Coq

Coq is a manual theorem proofing system of INRIA. One can define functions, predicates and theorems as VCs to check them by Coq. But Coq does not check all VCs at one time, but step by step. Even Coq is a manual theorem prover, it evaluates specifications efficiently. Already proofed theorems can be reused. The MOBIUS project is using Coq as the theorem prover in order to generate certificates of advanced security properties.

3 Integration of a new VCGen

This chapter describes the integration of a new direct VCGen in order to generate VCs in sorted logic.

Figure 3.1 illustrates the operation flow with the new VCGen on the right hand side. We avoid GCs to keep the generated VCs more related to the origin Java source code. We first give an overview of the new VCGen and the general concept in the first section in 3.1. Section 3.2 describes the required translation features in order to define the syntax of FOL terms in section 3.3. Sections from 3.4 to 3.9.2 describe the translation of the most JML level 0 annotations[1] into FOL terms in detail. We use the Extended Backus-Naur form (EBNF) to describe the syntax of a JML annotation. The bold JML annotations within the EBNF syntax will be translated into FOL terms. All other annotations belong to future work, as described in section 4.2.

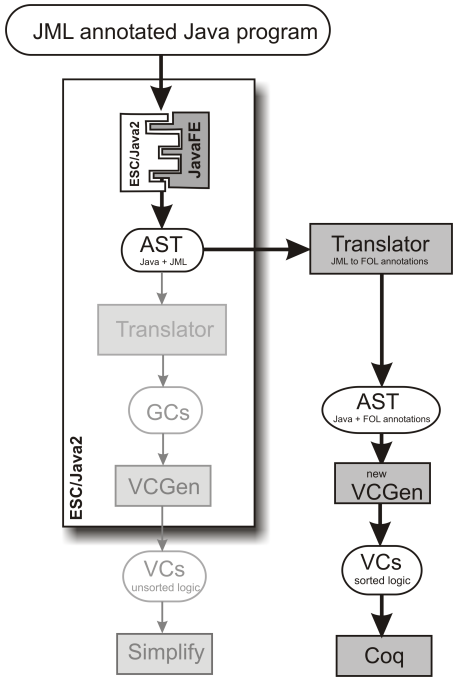

Figure 3.1: ESC/Java2 and the new integrated VCGen

[1] See section 2.1.1 that describes which JML annotations are covered in this thesis.

3.1 General Concept

For the integration of our new VCGen, we use a new translation mechanism to avoid the intermediate language represented as GCs. The new VCGen is developed by the project team at INRIA. The project team at ETH Zurich is responsible for the translation of JML annotations to FOL terms in sorted logic. Our new translation operation takes place after generating the AST as illustrated on the right hand side of figure 3.1. Afterwards, there is a new independent operation flow to generate VCs in sorted logic. The main function of this translation is to scan the AST for JML annotations which than get translated into FOL terms in sorted logic, depending on the kind of JML annotation. For routine's pre- post-, and exceptional postconditions as well as object's `invariants`, `initially`, and `constraint`, we use a lookup table to store their FOL terms. In the case of JML statements within the body of a routine, ESC/Java2 has the ability to decorate the next Java statement with the translated FOL term. Every term that decorates a Java statement, has to hold in order to execute the Java statement decorated with. After handling all JML nodes, we delete these nodes of the AST to gain a JML free AST.

In this master thesis, we use ξ(e) to denote the translated FOL term of the corresponding JML annotation e.

3.1.1 Example of a Translation

We demonstrate the handling and the translation of a JML statement on a simple example. The next Java code shows the method "foo()" that contains three statements. JavaFE and ESC/Java2 generates the AST as partly illustrated in figure 3.2 on the left hand side. The body of the "foo()" method consists of two Java and one JML statement, an `assume` clause.

```
public int foo(){
  x = y + z;
  ...
  //@ assume x = y + z;
  return x;
}
```

We decorate the next Java statement (the return statement) with the local annotation, currently `assume` ξ(e). This local annotation has to hold before executing the Java statement decorated with it. Every JML clause gets deleted from the AST after its handling. The modified AST of the method "foo()" is partly illustrated on the right hand side in figure 3.2.

During the complete translation, we have to consider different situations. For example, what happens if there is no last AST node. The handling of such statement annotations is explained in detail in 3.4.

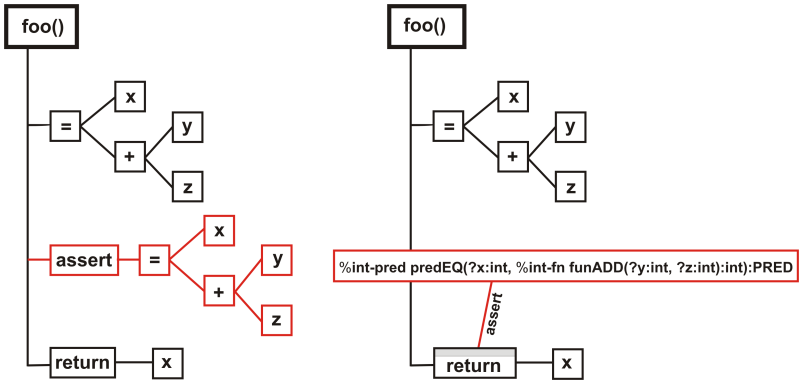

Figure 3.2: Illustration of the AST before and after translating the local annotation

3.2 Translation Features

This section explains the features that are used for the translation operation. The list below gives a brief introduction to each feature. The next sections describe these features in more details. In addition, sections 3.2.4 defines the visible states of a routine and section 3.2.5 defines the dependences of fields in type specifications.

- **Property object:** Contains flags, different data structures, and other values to aid the handling of JML annotations.
- **Lookup table:** Contains all specification information of a routine (method and constructor), namely the precondition Φ, postcondition Ψ, and exceptional postcondition Ψe. Class specification as object `invariants` and **history constraints** are also available for each class declaration.
- **Annotation table:** Contains local annotations of a particular Java statement. These local annotations, represented as individual objects, characterize the state before the execution of that particular Java statement.

3.2.1 Property Object

The property object contains any values to enable modular translations of JML annotations. That property object is always passed as an argument to the next method call in the visitor class. Therefore, we can state any behavior that is relevant for the underlying visitor's methods. The property object is of type **Java.util.properties** and stores pairs of values. There are no duplicates of keys, whereas the value of same keys gets overridden. The most important ones, including their type and default values in brackets, are introduced in the next list.

- **interesting:bool** (*false*): If the node is a JML feature and thus, of our interest
- **assignableSet:** (empty): The set containing all modifiable fields of a method
- **nothing:bool** (*false*): If the `assignable` feature holds the "nothing" keyword
- **quantifier:bool** (*false*): If the local variable belongs to a quantified expression
- **quantVars:** (empty): The set of all variables belonging to a quantified expression
- **isHelper:bool** (*false*): If the routine is a `helper` routine
- **fresh:bool** (*false*): To collect all field variables of the `fresh` feature
- **freshSet:** (empty): The set containing all fields that are allocated but not yet initialized
- **pred:bool** (*false*): Whether the FOL term has to be of sort `sortPred`
- **old:bool** (*false*): If the value of the field is the same as in the prestate
- **dsc:bool** (*false*): If the subset checking tag was set by the user in a command line
- **doSubsetChecking:bool** (*false*): Same as *dsc*, but in full written words
- **subsetCheckingSet:** (empty): To check if a set only accesses the own fields
- **visibleTypeSet:** (empty): To collect all modifiable types in the body of a method

We demonstrate the usage of one common property. During the translation, we often meet the act of collecting object fields into a set. To state this intention, we set a certain boolean property before we visit all object fields. Afterwards, we set the boolean property back to its default value. The next example shows the collecting part of object fields a, b, and c of the `fresh` feature in the translation part of `fresh` annotation object "x":

```
//@ fresh a, b, c;
```

```
1  prop.put("fresh", Boolean.TRUE);
2    visitASTNode(x, prop);
3  prop.put("fresh", Boolean.FALSE);
```

This code snippet is situated in the method of visiting a `fresh` feature. Before we visit the stored object list (second line), we set the "fresh" value to *true*. On the third line, we set the value of "fresh" again to *false*.

```
1 if prop.get("fresh")) {
2   final QuantVariableRef qref = Expression.rvar(fieldAccess.decl);
3   HashSet<Term> freshSet = (HashSet) prop.get("freshSet");
4   freshSet.add(qref);
5   prop.put("freshSet", freshSet);
6 }
```

During the visit of an object field, we check if it is an object field of the `fresh` set. In that case, we add the variable as a quantified variable reference into the existing set "freshSet". We have access to this set during the whole program execution. It will be reused in the method of handling `fresh` annotations. Than, we generate a FOL term to express that all objects in the "freshSet" are newly allocated in the routine's body.

3.2.2 Lookup Table

The lookup table for routine annotations contains all specification information that is available for that routine. It contains the precondition Φ, the normal postcondition Ψ, and several exceptional postconditions $\Psi e(t)$, whereas all exceptional postconditions have same exception object t. The VCGen only has to deal with the FOL terms in the lookup table and not with the large number of JML annotations. On the implementation level, we use hash maps for each condition. Hash maps have the advantage to have fast access to any entry and do not store duplicates. Any entry consists of a key (the routines declaration), and a value (the FOL term). The entries can be read, written and extended at any time. These hash maps are accessable through a static **Lookup** class:

- **Preconditions** Φ: <RoutineDecl, Term>
- **Postconditions** Ψ: <RoutineDecl, Post>
- **Exceptional Postconditions** $\Psi e(t)$: <RoutineDecl, Post>

Ψ and $\Psi e(t)$ contain values of a new data type **Post**. This data type additionally contains a variable that is associated with the FOL term. For Ψ, we store the return type of the routine, and for $\Psi e(t)$, we use a new generated variable to represent the thrown exception.

Routine annotations are influenced by various JML annotations such as `requires`, `ensures`, and `signals` clauses as well as `invariant`, `assignable`, and `initially` clauses. Table 3.1 summarizes the mentioned influences for Φ, Ψ, and Ψe.

Precondition Φ	Postcondition Ψ	Exceptional Postcondition Ψe
`requires`	`ensures`	`signals`
`invariant`	`initially`	`signals_only`
	`assignable`	`initially`
	`constraint`	`assignable`
	`invariant`	`constraint`
		`invariant`

Table 3.1: Routine conditions and their influencing JML features

The lookup table also contains information about the object `invariant` and `constraint` predicates. Even they have influences on Φ, Ψ, and Ψe, as seen in table 3.1, there exist a separate lookup table for the mentioned type specifications. On implementation level there exist two hash maps with the following syntax:

- **Invariants:** <ClassDecl, Term>
- **Constraints:** <ClassDecl, Term>

Instead of conjoining the translated predicates to each routine annotation, we only conjoin a term that refers to the **invariant** and **constraint** term in the lookup table. See section 3.10 for more details about the handling of `invariant` annotations.

In the following, we use the notation **Lookup.<condition>.<declName>** to represent the FOL term of a specific condition of a routine or a class, represented by its name. To represent the FOL term of the `invariant` of a class named "TestClass", for instance, we use:

Lookup.invariant.TestClass

3.2.3 Annotation Table

Similar to the lookup table, we use an annotation table to decorate any Java statement with zero or more local annotations. These local annotations characterize the state before the execution of that Java statement. A local annotation is either a JML's `assume`, `assert`, `set`, `ghost` variable, or a `loop invariant` statement. On the implementation level, we have one hash map containing Java statements as a key with a set of local annotations represented as annotation objects. We use annotation objects to denote the type of a JML statement. There are four different object types to represent each JML statement. Every instantiated object contains a field that yields the translated FOL term. The next listing explains each object type and its represented JML statement:

- **Assert:** Represents `assert` ξ(e). It has to be proven to hold.
- **Assume:** Represents `assume` ξ(e). It is assumed to hold.

- **Set:** Represents `set` ξ(e) and `ghost` ξ(e)[2].
- **Maintaining:** Represents `maintaining` ξ(e). It has to hold before and after each loop iteration. The Java statement decorated with must be a loop statement.

3.2.4 Visible States

Visible states are the pre- and poststates of routines unless they are marked as `helper` routines. Any private routine augmented by the `helper` modifier is independet of type specifications and can violate them. Figure 3.3 shows all visible states located in a non-helper method. A visible state exists if the control is:

- at the beginning and end of each non-helper method
- at the end of each non-helper constructor
- at method or constructor invocations in the body of any non-helper routine

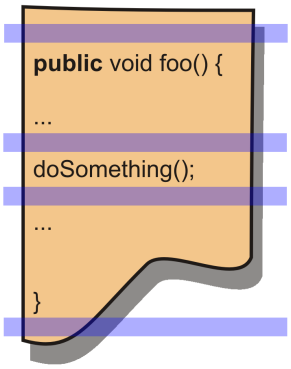

Figure 3.3: Visible states in a method

3.2.5 Dependences of Fields in Type Specification

We restrict the expressiveness of `invariant`, `initially` and `constraint` specifications, since their predicates are only allowed to depend on predestinated fields. See [15] for more details about `invariant` field dependences on superclass fields. A field is predestinated if it is defined in the class that declares these type specification mentioned above. We check the admissibility by a subset check[3] during the translation process. This subset check can be turned on and off by a command line entry.

[2]See chapter 3.6.1 for more details about `ghost` variables.
[3]See section 3.11 for more details about the subset check.

3.3 Definition of FOL Terms

First **O**rder **L**ogic (FOL) is a system to express a relation between arguments like numbers and variables. The validity of a FOL term is provable.

A FOL term consists of:

- Predicates
- Functions
- Constants
- Variables
- Logical operators: not, and, or, conditional, biconditional
- Quantifiers: $\forall, \exists$
- Equality symbol: $=$

Predicates express any relation between one and several variables. In our project, predicates are simply specification expressions. See appendix C for the complete syntax of specification expressions. Expressions can be linked with particular operators. We distinguish between numerical and boolean operators. Two numerical operands and a numerical operator result again in a numerical expression. Using a boolean operator instead of the numerical one, the result expression will be a boolean expression.

3.3.1 Numerical Operations

```
bool−expr ::= num−expr relation num−expr | ...
relation  ::= == | != | < | <= | > | >=
num−expr ::= num−expr [num−op num−expr] | number
num−op ::= + | − | ∗ | / | % | ++ | −− | ...
number ::= N | R | Q |  ...
```

This syntax of numerical operations is an insight only and thus, it is not completet. See appendix C for the complete syntax of numerical expressions. The operator between to numerical operands is either a relation or a numerical operator. Numerical operators are translated as numerical functions, whereas relation operators act as numerical predicates. The next two subsections describe the translations of numerical operators and numerical predicates into FOL terms:

Numerical Functions

The prefix of every numerical function term is either **“%int-fn”** or **“%real-fn”**, depending on the sort of the operands. Each numerical operator is represented as an own string. The string of a multiplication function, for example, would be **“funMul”**.

The syntax of every numerical function is:

```
prefix op−string (operand1, operand2):sort
```

The translation of “2 * (1+3)” would result in the following term:

```
%int-fn funMUL(2, %int-fn funADD(1,3)):int
```

The sort of such a term is either `sortInt` or `sortReal`, depending on the sorts of the operands. If one of the operands is defined as sort `sortReal`, the other operand get converted into `sortReal` as well, unless it is already defined as sort `sortReal`. Additionaly, each translation does a type check and throws an exception if the operands have wrong sorts.

Numerical Predicates

A relation operator between two numerical operands is called numerical predicate. The sort of such a predicate is either `sortBool` or `sortPred`. The syntax of the translated term is similar to the syntax of numerical function terms. The prefix is either **“%int-pred”** or **“%real-pred”** depending on the sorts of the operands. The numerical predicates are represented as a string expressing the numerical predicate operation. Any term of a numerical predicate operation has the following syntax:

```
prefix op−string (operand1, operand2):sort
```

The translation of “2 < 4” would result in the following term:

```
%int-pred predLT(2, 4):PRED
```

We discuss the difference between `sortBool` and `sortPred` after section 3.3.2.

3.3.2 Boolean Expressions

```
bool−expr ::= bool−expr bool−op bool−expr | ...
bool−op ::= '|' | & | '||' | && | == | !=
```

Any boolean expression[4] can either be of `sortBool` or `sortPred`. We normally deal with `sortPred` except for one special case that is described after this section. The operation is directly written as a prefix followed by its operands. For example the expression “true && true” will be translated into the following term, if we wish a result term of `sortPred`.

```
%and(%isTrue(true):PRED, %isTrue(true):PRED):PRED
```

[4] See appendix C for complete syntax of boolean expressions.

The same example, but translated into a sortBool term, is:

%bool-fn boolAnd(true, false):bool

When do we select sortBool and when sortPred?
There is one operation that only exists for operands both of sort sortBool. We are talking about the equality operation. There is no equalitiy operation between two operands, if at least one is defined as sortPred. In that case, we use the imply operation to keep the same meaning. Both operands have to be of sortPred. There is a method available to convert from sortBool into sortPred. In fact, if we have an equality operation between two operands, we try to keep both operands of sortBool to use the "real" equality operator. We state this behavior by setting a flag within the property object to gain two operands of sortBool. Thus, before each logical operation, we check that flag to result the operation in the preferred sort. It is just a preferred sort, not a directive, because not every operation can result in a expression of sortBool. Independently of that, we always check the sorts and throw an exception if two terms do not fit for a particular operation.

3.3.3 Predicates

predicate ::= spec−expression
spec−expression ::= expression

Predicates are simple specification expressions. A predicate in ESC/Java2 always yields a term of sort sortPred, thus, a predicate can be verified at any time. The complete syntax of an expression is given in appendix C. ESC/Java2 already contains a large amount of defined predicates. We extend the list with new predicates:

- **isAssignCompat**

isAssignCompat(heap:sortMap, object:sortValue, type:sortType):sortPred

This predicate denotes whether or not the type of a given object is of same type or a subtype of a given type on the same heap. This subtype relation is also known as "\type(object) <: Type". By calling "isAssignCompat(heap, o, t)", we get the following term:

%**assignCompat**(?**heap**:**map**, ?#o:**ref**, ?#t:**type**):**PRED**

- **isAlive**

isAlive(heap:sortRef, target:sortRef):sortPred

This predicate yields a term expressing if a given object is at least allocated on a given heap. The object does not yet have been instantiated. By calling "isAlive(heap, t)", we get the following term:

%**isAlive**(?**heap**:**map**, ?#t:**ref**):**PRED**

- **inv**

inv(heap:sortMap, object:sortRef, type:sortType):sortPred

This predicate expresses that the `invariant` of the given object of a given type should hold in a given heap. An object has to preserve the own object `invariants` and also the `invariants` of all instantiated objects in all visible states. The "inv" predicate will be used to refer to other object `invariants`. See chapter 3.10.2 and 3.10.3 for more details. By calling "inv(heap, o, t)", we get:

```
%inv(?heap:map, ?#o:ref, ?#t:type):PRED
```

- **isFieldOf**

isFieldOf(heap:sortMap ,target:sortRef, field:sortAny):sortPred

This predicate is used to state that a field is of a particular object in given heap. By calling "isFieldOf(heap, t, f)", we get:

```
%isFieldOf(?heap:map, ?#t:ref, ?#f:any):PRED
```

3.3.4 Special Notations

We use special notations to simplify the semantics of the translations. Special notations have their own meanings, which are described in this chapter.

- **isAssignable**

isAssignable(field:sortAny, store-ref-list):sortPred

This notation is used to state whether or not a given field is assignable. It is assignable if it is equal to a reference within the **store-ref-list**. The semantics of this notation for a field:TestClass and a store-ref-list {x1:X, x2:X} is:

(field == x1) ∨ (field == x2)

And as a FOL term:

```
%or(
  %anyEQ(
    %valueToAny(
      %dynLoc(?heap:map, ?#r1:ref, ?#field:any):ref
    ):any,
    %valueToRef(
      %dynLoc(?heap:map, ?this:ref, ?TestClass?x1FieldSignature:ref):ref
    ):ref
```

```
  ):PRED,
  %anyEQ(
    %valueToAny(
      %dynLoc(?heap:map, ?#r1:ref, ?#field:any):ref
    ):any,
    %valueToRef(
      %dynLoc(?heap:map, ?this:ref, ?TestClass?x2FieldSignature:ref):ref
    ):ref
  ):PRED
):PRED
```

- **isVisibleIn**

isVisibleIn(t:sortType, type-list:sortType):sortPred

This notation is used to find out if a given type is within a **type-list**. This list contains all modifiable types of sort `sortType` of a routine which are relevant for the `invariant` in the poststate. The semantics of this notation for a type t and a store-ref-list {T1, T2} is:

(t == T1) ∨ (t == T2)

And as a FOL term:

```
\%or(
  \%anyEQ(
    ?\#t:type,
    ?(ReferenceType (ClassType T1.className)):type
  ):PRED,
  \%anyEQ(?\#t:type,
    ?(ReferenceType (ClassType T2.className)):type
  ):PRED)
:PRED
```

3.4 JML Statements

jml−annotation−statement ::= **assert**−statement
| **assume**−statement
| hence−by−statement
| **set**−statement
| refining −statement
| unreachable−statement
| debug−statement

JML statement clauses characterize any behavior before the execution of the next Java statement in the body of a routine. Zero or more JML statements (also known as local annotations) can be used to decorate a Java statement using the local annotation table[5]. This table contains any Java statement associated with a set of local annotations.

The following sequence shows the translation of all JML statements within the body of a routine. The translation of the `assume`, `assert` and `set` annotations is given in section 3.4.1, and the translation of the `maintaining` annotation in section 3.4.2.

1. **Search** for next JML statement.
2. **Translate** JML statement to a FOL term, ξ(e).
3. **Generate** a new annotation object (Assert, Assume, or Set, depending on the JML statement) containing ξ(e).
4. **Decorate** next Java statement with this annotation. If there is no more Java statement, insert a dummy 'Java statement "skipStmt" as the very last Java statement and decorate it.
5. **Delete** the JML statement node from the AST.

There are Java statements that may contain again JML statements. If we reach one of these Java statements, actually loop statements, (**WhileStmt**, **ForStmt, DoStmt, BlockStmt, TryCatchStmt,** and **IfStmt**), we execute step 1 to 5 again to handle all JML statements.

The left hand side of figure 3.4 illustrates the usage of JML and Java statements in a body of a method. After adding a local annotation to the table, we delete it from the AST. The result of the Java code of figure 3.4 is shown on the right hand side.

[5] See section 3.2.2 for more details about the local annotation table.

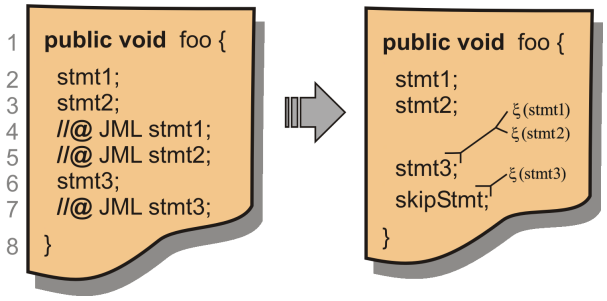

Figure 3.4: Same code, before and after translation

3.4.1 Assert, Assume, Set

assert−statement ::= **assert** predicate [: expression] ;

assume−statement ::= **assume** predicate [: expression] ;

set−statement ::= **set** assignment−expr ;

An `assert` annotation requires the given predicate to hold. It must be proven to hold by the theorem prover. An `assume` annotation specifies that the given predicate is assumed to hold. It acts as additional information and does not have to be proven. A `set` annotation is used to assign any value to a `ghost` variable[6]. The `assert` and `assume` annotations both have an optional expression that must be of type string. That string expression is printed if the assertion fails. The current implementation does not provide this kind of error message yet and is a topic for future work.

3.4.2 Maintaining

loop−**invariant** ::= **maintaining**−keyword predicate;
maintaining−keyword ::= **maintaining**
| maintaining_reduntandly
| **loop_invariant**
| **loop_invariant_redundantly**

`Loop invariant` annotations can only occur prior to a loop statement (**WhileStmt, ForStmt** or **DoStmt**). They have to hold at the beginning and at the end of each loop iteration. The next statement after a `loop invariant` annotation is either another `loop invariant` or a loop statement. We conjoin all `loop invariant` translations and conjoin them to the invariant of next loop statement. The annotation table contains an association between loop statements and a FOL term, the `loop invariant`.

[6]For more details about `ghost` variables, see section 3.6.1

Note, that only loop statements can be decorated with `loop invariants`. The next example shows two nested `loop invariants`. The translation of these `loop statements` are conjoined to the invariant of the next loop statement.

```
int count, x;

//@ maintaining count > 0;
while (

  //@ maintaining x != 0;
  for  (...){...}
  ...
}
```

Loop invariant of WhileStmt

```
%int−pred predGT(?count:int, 0):PRED
```

Loop invariant of ForStmt

```
%int−pred predNE(?x:int, 0):PRED
```

3.4.3 Example of JML Statements

We demonstrate the handling of JML statements on the following Java code. There are three JML annotations that decorate the following Java statement.

```
1  public void foo() {
2     int count = 0;
3
4     //@ assume count == 0;
5     count++;
6     //@ assert count != 0;
7
8     //@ maintaining count > 0;
9     while (...) {
10       ...
11    }
12
13 }
```

Java statement on line 5 is decorated with one `assume` object that contains ξ(e):

%**int**−**pred** predEQ(?count:**int**, 0):**PRED**

Java statement on line 9 is decorated with one `assert` object that contains ξ(e):

%**not**(%**int**−**pred** predEQ(?count:**int**, 0):**PRED**):**PRED**

Java statement on line 9 is decorated with one `loop invariant` object that contains ξ(e):

%**int**−**pred predGT**(?count:**int**, 0):**PRED**

Remarkable: The Java statement on line 9 is decorated with one `assert` object and its invariant is extended by the `loop invariant`.

3.5 Method Specifications

method−spec ::= **requires**−clause
| diverges−clause
| **assignable**−clause
| captures−clause
| when−clause
| working−space−clause
| duration−clause
| **ensures**−clause
| **signals**−only−clause
| **signals**−clause

Method specifications allow to specify the behavior of a routine. In addition to pre- and postcondition specifications, JML offers other method specifications, for example the `assignable` clause. They must hold either in the pre- or poststate of a routine's execution, depending on the kind of method specification. The lookup table (see chapter 3.2.2) contains data structures for precondition Φ, postcondition Ψ, exceptional postconditions Ψe and for class `invariants` as well as for class `history constraints`. Each routine corresponds to one entry in the lookup table for each method specification. Our goal is to translate each method specification into a term and conjoin all of them to the associated constraint.

3.5.1 Requires

requires−clause ::= **requires**−keyword pred−or−not ;
requires−keyword ::= **requires**
| **pre**
| **requires_redundantly**
| **pre_redundantly**
pred−or−not ::= predicate
| \not_specified
| \same

Predicates of `requires` clauses, also known as preconditions, state the behavior before the routine's invocation. Any `requires` predicate has to hold before the body of the declaring routine can be executed. If the `requires` clause is omitted, ESC/Java2 generates a dummy `requires` clause with the boolean value *true*. Calling a method, we have to be sure that the callee object type is a subtype of the class declaring that method. For the translation of `requires` clauses, we distinguish between constructors and methods, since there is no callee object instantiated yet during invoking a constructor.

Constructors

We do not check the type of the callee object, but we check if the `requires` predicate "e" holds. To gain the translated term, we conjunction each `requires` translation. This is represented by using the formula "$\bigwedge$(k) A_k". It yields a term of all occurrences of "A" connected by the operand "$\wedge$" as for example "$A_1 \wedge A_2 \wedge A_3 \wedge \ldots$. To the constructor's precondition Φ, we conjoin a term with the following sematics:

$\bigwedge$(i) requires$_\xi$(e_i)

In the case of an omitted clause, we conjoin a term with following semantics to the constructor's precondition Φ:

%isTrue(true)

Methods

The dynamic type of the callee object must be a subtype of the type declaring that method. This assertion is expressed by using the predicate **isAssignCompat(heap, o, t)**[7] whereas the declaring class type is represented as "ClassType". We conjoin this assertion combined with the translated predicate "e" of all `requires` clauses to the method's precondition Φ. The semantics of that term is:

isAssignCompat(heap, this, ClassType) $\wedge$ $\bigwedge$(i) requires$_\xi$(e_i)

In the case of an omitted clause, we only conjoin the "isAssignCompat" term to the precondition Φ:

isAssignCompat(heap, this, ClassType)

The next example shows one method with two `requires` clauses. The translated FOL term for the method's precondition is shown below the Java code:

```
1  public class Example {
2    boolean test;
3
4    //@ requires num > 0;
5    //@ requires test == true;
6    public void foo(int num) {...}
7
8  }
```

[7] See section 3.3.3 for more details about the predicate "isAssignCompat(heap, o, t)".

Lookup.precondition.foo

```
%and(
  %and(
    %assignCompat(
      ?heap:map,
      ?this:ref,
      ?(ReferenceType (ClassType BlackType.className)):type):PRED,
    %int−pred predGT(
      ?num:int,
      0
    ):PRED
  ):PRED,
  %anyEQ(
    %valueToBool(
      %dynSelect(?heap:map, ?this:ref, ?Example?testFieldSignature:bool):value
    ):bool,
    true
  ):PRED
):PRED
```

3.5.2 Ensures

```
ensures−clause   ::= ensures−keyword pred−or−not ;
ensures−keyword ::= ensures
    | post
    | ensures_redundantly
    | post_redundantly
```

`Ensures` clauses are used to specify what must hold after terminating a routine's execution without throwing any exception. The usage of the JML keyword `old` is common for postconditions to refer to values in pre_heap. An `ensure` clause requires all `requires` clauses to hold. Thus, we include this requisition for each `ensures` clause to the translated FOL term. We distinguish the generation of FOL terms between constructors and methods, because we have to check the dynamic type of the called objects in the case of method calls. We conjoin the translated FOL term to the method's postcondition Ψ and exceptional postcondition Ψ(e).

Constructors

For every `ensures` predicate we check if all `requires` predicates hold. Ensuresξ(e) denotes the translation of the `ensures` predicate. The semantics of the translated `ensures` clauses is:

$$\bigwedge(\mathbf{i})\ (\bigwedge(\mathbf{k})\ \mathbf{requires}\xi(e_k) \rightarrow \mathbf{ensures}\xi(e_i))$$

If the `ensures` clause is omitted, we generate the following term:

%isTrue(true):PRED

Methods

The generated FOL term of method's `ensures` clauses is similar to that of constructor's. In addition, we check the dynamic type of the called object. The semantics of that generated FOL term is:

$$\bigwedge(\mathbf{i})((\mathbf{isAssignCompat(heap, this,}\ \mathit{ClassType}) \wedge \bigwedge(\mathbf{k})\ \mathbf{requires}\xi(e_k)) \rightarrow \mathbf{ensures}\xi(e_i))$$

In the case of an omitted `ensures` clause, we generate the following term:

%isTrue(true):PRED

The next example shows the generation of a method's postcondition term. Any method argument used in an `ensures` predicate holds the value of its prestate. We denote this by adding the prefix **pre_**[8] to the name of that argument.

```
1 public class Example {
2   boolean test;
3
4   //@ requires num > 0;
5   //@ ensures test == true;
6   public void foo(int num) {...}
7
8 }
```

```
%implies(
  %and(
    %assignCompat(?heap:map, ?this:ref, ?ClassType.className):type):PRED,
    %int−pred predGT(?\pre_num:int, 0):PRED
  ):PRED,
  %anyEQ(
    %valueToBool(
      %dynSelect(?heap:map, ?this:ref, ?Example?testFieldSignature:bool):value
    ):bool,
    true
  ):PRED
):PRED
```

[8] See section 3.8.3 for more details about the `old` clause and argument values in `ensures` clauses.

On Implementation Level:
ESC/Java2 does not guarantee the existence of an `ensures` clause, since it does for `requires` clause. If the `ensures` clause is omitted, we have to add manually an `ensures` clause to the AST, that contains a predicate with boolean value *true*.

3.5.3 Signals_only

signals−only−clause ::= **signals**−only−keyword [, reference−**type**] ;
| **signals**−only−keyword \nothing ;
signals−only−keyword ::= **signals_only**
| signals_only_redundantly

`Signals_only` clauses, also known as exceptional postcondition, are used to specify the behavior in an exceptional termination of the routine. The normal postconditions may not hold at the exceptional termination of a routine, but the exceptional postconditions have to. There are two exceptional clauses: (1) `signals_only` clauses are used to specify what type of exception might be thrown by the routine while (2) `signals` clause must hold when a certain type of exception has been thrown. The `signals` clause will be described in the next subsection.

The `signals_only` clause specifies what exceptions can be thrown by a method or constructor. All other exceptions may not be thrown. The method's throw list achieves the same task as the `signals_only` clause. They will be translated identically. One either uses the `signals_only` clause or the methods throw list, since only one can be evaluated. If both are used, the `signals_only` clause has higher priority. There can at most be one `signals_only` clause, which may hold several exception reference types. The translated `signals_only` clause gets conjoined to the method's exceptional postconditions Ψe. We demonstrate the translation of the `signal_only` clause by using the following example:

```
//@ signals_only E1, E2, E3;
```

The corresponding method may only throw exceptions of type E1, E2, or E3. To translate this behavior, we generate a term with the following meaning: *"If the thrown exception is a subtype of java.lang.Exception and the object that threw the exception is subtype of type to whom the method belongs to, then the thrown exception must be a subtype of the listed exception of the* `signal_only` *clause"*.

The thrown exception is represented as the variable associated with the term of the instantiated **Post** object[9]. We substitute all occurrences of that thrown exception variable by the overall exception variable within the **Post** object. If the `signals_only` clause is omitted, ESC/Java2 generates a new `signals_only` annotation with the predicate keyword "nothing". This denotes that no other exception may be thrown except those in the routine's thrown list.

[9]See section 3.2.2 for more details about **Post** objects.

The syntax of the generated FOL term is:

isAssignCompat(heap, ex, *java.lang.Exception*)
→ **(isAssignCompat(heap, this,** *ClassType*) ∧ ⋀(k) **requires**ξ(eₖ)
→ (⋁(i) **isAssignCompat(heap, ex, E**ᵢ)))

We demonstrate the translation of the `signals_only` clause on the following example:

```
1 public class A {
2
3   //@ signals_only E1, E2;
4   public void foo() { ... }
5
6 }
```

Using "r" as the overall exception variable "ex" and the `signals_only` clause from above, we gain the following FOL term:

```
%implies(
  %assignCompat(
    ?heap:map,
    ?#r:ref,
    ?(ReferenceType (ClassType java.lang.ExceptionType.className)):type
  ):PRED,
  %implies(
    %assignCompat(
      ?heap:map,
      ?this:ref,
      ?(ReferenceType (ClassType A.className)):type
    ):PRED,
    %or(
      %or(
        %isTrue(false):PRED,
        %assignCompat(
          ?heap:map,
          ?#r:ref,
          ?(ReferenceType (ClassType E1.className)):type
        ):PRED
      ):PRED,
      %assignCompat(
        ?heap:map,
        ?#r:ref,
        ?(ReferenceType (ClassType E2.className)):type
      ):PRED
    ):PRED
  ):PRED
):PRED
```

Remark: Each `signals_only` clause contains a hidden "nothing" exception. This means, that even we have some reference types, the generated FOL term also contains the disjunction of "%isTrue(false):PRED", that represents the "nothing" exception.

3.5.4 Signals

signals−clause ::= **signals**−keyword (reference−**type** [ident]) [pred−or−not] ;
signals−keyword ::= **signals** | **signals_redundantly**
| **exsures** | **exsures_redundantly**

A routine may augment its method specification by using one ore more `signals` clauses. The `signals` exception type must be listed in either a corresponding `signals_only` clauses or the routine's throw list. The semantic of the translated FOL term of all `signals` clauses is:

$$\bigwedge(\mathbf{i})(\mathbf{isAssignCompat(heap, ex, ExceptionType)} \rightarrow \mathbf{(isAssignCompat(heap, this, ClassType)} \wedge \bigwedge(\mathbf{k})\ \mathbf{requires}_{\xi}(e_k)) \rightarrow \mathbf{signals}_{\xi}(ex_i))$$

We demonstrate the translation of a `signals` clause with the next example:

```
1  public class Example{
2    int count;
3
4    //@ signals_only E1;
5    //@ signals (E ex) count == 3;
6    public void foo() { ... }
7
8  }
```

```
%implies(
  %assignCompat(
    ?heap:map,
    ?#r:ref,
    (ReferenceType (ClassType E1.className)):type
  ):PRED,
  %implies(
    %assignCompat(
      ?heap:map,
      ?this:ref,
      ?(ReferenceType (ClassType Example.className)):type
    ):PRED,
    %int−pred predEQ(
      %valueToInt(
        %dynSelect(
          ?heap:map,
          ?this:ref,
```

```
            ?Example?countFieldSignature:int
          ):value
        ):int,
        3
      ):PRED
    ):PRED
):PRED
```

3.5.5 Assignable

assignable−clause ::= **assignable**−keyword store−ref−list ;
assignable−keyword ::= **assignable** | assignable_redundantly
| modifiable | modifiable_redundantly
| modifies | modifies_redundantly

By using an `assignable` clause, one can limit the heap locations of objects, which are assignable during the method's or constructor's execution. The assignable objects, represented as heap locations, may have different values in the pre- and poststate. All heap locations, that are not mentioned, have to keep the value of their prestate. The generated FOL terms are conjoined to the method's or constructor's postcondition Ψ and exceptional postcondition Ψe in order to verify this behavior after terminating the execution. The `assignable` and the `modifiable` clause different in the fact that assigned and then re-established heap locations have to be mentioned in the `assignable` clause but not in the `modifiable` clause. Thus, we check in every assign expression whether or not that heap location is mentioned in the `assignable` clause. We simplify this semantics by treating the `assignable` clause similar to the one for `modifiable` clauses, which only checks the modified heap locations at the end of the routine's execution. The handling of the complete `assignable` syntax will be mentioned as a future work in chapter 4.2. There are three different kinds for the **store-ref-list**:

everything: All heap locations of the entire program are assignable.

nothing: No heap location can be assignable. It is equal to a pure method.

store-ref-list: Denotes a set of assignable heap locations.

The translated FOL term expresses whether or not a location may be modified, using an "assign" expression. Method or constructor parameters can never be listed in the `assignable` clause. ESC/Java2 generates a dummy node "assignable \everything", if the `assignable` clause is omitted. There is a different translation for each case.

everything: An `assignable` clause with the "everything" keyword has not to be translated into a FOL term. There is no need to handle any heap location with respect to their assignability.

\nothing: No heap location may be modified. Every heap location imperatively has to keep the value of the prestate. We generate a term with the meaning of "*Every heap location of a particular object is either not allocated, or it has the same value in the pre- as in the poststate*".

$$\forall\ \textbf{target:ref, field:any (isFieldOf(heap, target, field)} \\ \rightarrow\ \textbf{(!isAlive(pre\_heap, target)} \vee\ \textbf{isAssignable(\old(field), field)))}$$

store-ref-list: The **store-ref-lists** contains one or more heap locations that can be modified. If a heap location is allocated and not listed in the **store-ref-list**, its value of the prestate is still the same as in the poststate. We generate a term with the meaning of "*Every field of a particular object, is either not allocated, or if allocated, it is allowed to be assignable (in* `assignable` *list), or it has to keep its value of the pre- in the poststate*". The syntax of the generated FOL term is:

$$\forall\ \textbf{target:sortRef, field:sortAny (isFieldOf(heap, target, field)} \\ \rightarrow\ \textbf{(!isAlive(pre\_heap, target)} \vee\ \textbf{isAssignable(field, ref-sort-list)} \\ \vee\ \textbf{isSame(\old(field), field)))}$$

The next example contains each mentioned `assignable` annotation on line 8, 20 and 22. The translated FOL terms are shown below the example, except the translation of the `assignable` clause using the "everything" keyword. In that case, no term will be generated.

```
1  public class ATest {
2
3    int age;
4    BTest b;
5
6    //@ assignable age, b;
7    public ATest() { ...   }
8
9    //@ assignable \nothing;
10   public void doSomething(){ ... }
11
12   //@ assignable \everything;
13   public void foo() {...}
14
15 }
```

Lookup.postcondition.ATest

```
%forAll [#r6:ref, #x4:any]
%implies(
  %isFieldOf(?heap:map, ?#r6:ref, ?#x4:any):PRED,
  %or(
    %or(
      %not(%isAlive(?\pre_heap:map, ?#r6:ref):PRED):PRED,
      %or(
        %anyEQ(
          %valueToAny(%dynLoc(?heap:map, ?#r6:ref, ?#x4:any):ref):any,
          %valueToRef(
            %dynLoc(?heap:map, ?this:ref, ?ATest?bFieldSignature:ref):ref
          ):ref
        ):PRED,
        %anyEQ(
          %valueToAny(%dynLoc(?heap:map, ?#r6:ref, ?#x4:any):ref):any,
          %valueToInt(
            %dynLoc(?heap:map, ?this:ref, ?ATest?ageFieldSignature:int):ref
          ):int
        ):PRED
      ):PRED
    ):PRED,
    %anyEQ(
      %valueToAny(%dynSelect(?\pre_heap:map, ?#r6:ref, ?#x4:any):value):any,
      %valueToAny(%dynSelect(?heap:map, ?#r6:ref, ?#x4:any):value):any
    ):PRED
  ):PRED
):PRED
```

Lookup.postcondition.doSomething

```
%%forAll [#r1:ref, #x0:any]
%implies(
  %isFieldOf(?heap:map, ?#r1:ref, ?#x0:any):PRED,
  %or(
    %not(%isAlive(?\pre_heap:map, ?#r1:ref):PRED):PRED,
    %anyEQ(
      %valueToAny(
        %dynSelect(?\pre_heap:map, ?#r1:ref, ?#x0:any):value
      ):any,
      %valueToAny(
        %dynSelect(?heap:map, ?#r1:ref, ?#x0:any):value
      ):any
    ):PRED
  ):PRED
):PRED
```

3.6 JML Modifiers

JML modifiers are used to modify any class, interface, method, and constructor declaration, as well as formal parameters and local variables.

jml−modifier ::= spec_public | spec_protected
| model
| **ghost**
| pure
| instance
| **helper**
| uninitialized
| spec_java_math | spec_safe_math | spec_bigint_math
| code_java_math | code_safe_math | code_bigint_math
| non_null | nullable | nullable_by_default
| extract

3.6.1 Ghost

ghost ::= decl | decl var−**set**
set ::= var−**set**

The `ghost` JML modifier introduces a specification-only field or variable. A `ghost` field or variable can only be used for specification purposes and is not visible outside any JML feature. This thesis covers the translation of `ghost` variables, not of `ghost` fields. The scope of a `ghost` variable is the body of the routine, where the variable is declared. Such a `ghost` variable can be introduced by using either a `ghost` declaration **decl** and a variable setting **var-set**, or just a `ghost` declaration. The `set` statement itself can only be used if the `ghost` variable is already declared, ESC/Java2 returns an error message otherwise. These two kinds of introducing a `ghost` variable are represented in the next example.

```
1 //@ ghost int count1;
2 //@ set count1 = 8;
3
4 //@ ghost int count2 = 4;
```

Line 1 declares a new `ghost` variable called **count1**. This `ghost` variable is initialized to an unspecific value. We have to assign a value to that `ghost` variable by using the `set` JML feature. This is done on line 2. The 4th line declares and assigns the integer value 4 to a newly introduced `ghost` variable **count2**. We can set new values for both `ghost` by using another `set` statement. In section 3.4.1, we have introduced a new data type **Set** for `ghost` annotations. Any object of type **Set** contains a field "assignment" and a field "declaration". We decorate the next Java statement with a `set` object, holding both declaration and assignment expression.

3.6.2 Helper

```
//*@helper*//
```

The JML feature `helper` is a modifier for private methods or constructors. Routines declared as `helpers` do not depend on the type specification as `invariant`, `initially` and `constraint` annotations. The next example shows the usage of the `helper` modifier. The constructor and the first method are declared as `helper` routines. Their pre-, post-, and exceptional postconditions are not augmented with the object `invariant`, the `initially`, and the `constraint` predicates. The second constructor (line 9) and the second method (line 11), on the other hand take care about the class specification.

```
1  public class Foo() {
2    int count;
3
4    //@ invariant count > 0;
5    //@ constraint \old(count) > count;
6    //@ initially  count != 0;
7
8    private /*@helper*/ Foo() {...}
9    public Foo(int x) {}
10   private /*@helper*/ void doSomething() { ... }
11   public void doNewStuff() { ... }
12 }
```

3.7 Quantifiers

```
spec−quantified−expr ::= ( quantifier quantified−var−decls ; [ [ predicate ] ; ]
                           spec−expression )
quantifier ::= \forall | \exists | \max | \min | \num_of | \product | \sum
```

JML quantifiers can be used in every JML expression. A quantified expression is defined by its unique name, a list of variable declarations, and a list of expression. Every quantified expression is of type **QuantifiedExpr**. The scope of the declared variables is the expression itself. To restrict the values of the declared variables, one can use the optional predicate between the two semicolons. The following steps are required before the translation of JML quantified expressions can be done:

1. Get the node tag of the **QuantifiedExpr** node (distinguish between `forall` and `exists`).
2. Collect all variable declarations by storing them into the "quantVars" within the object property.
3. Translate the contained expression.

The translation of the `forall` quantifier is given in section 3.7.1, and the one of the `exists` quantifier in 3.7.2.

3.7.1 Forall

The `forall` quantifier is the universal quantifier and restricts the values that satisfy the specification. We first give an example of a JML precondition using a `forall` quantified expression:

```
public class TestA{
  int[ ] testArray = {1, 2, 3};

  //@ requires (\ forall  int  i;  i < 3; testArray[i] < 4);
  public  TestA( ) {...}

}
```

We allocate an array called "testArray" in a class "TestA". The array contains three integer elements. For the method "foo()", we require, that the array only holds values less than 4. To express this behavior in the prestate of the method "foo()", we use the universal quantified expression in a `requires` clause. The translation of this quantified expression is given below:

Lookup.precondition.TestA

```
%forAll [i:int]
%implies(
  %int−pred predLT(?i:int, 3):PRED,
  %int−pred predLT(
    %valueToInt(
      %arrSelect(
        ?heap:map,
        %valueToRef(
          %dynSelect(?heap:map, ?this:ref,  ?TestA?testArrayFieldSignature:ref
              ):value
        ):ref,
        ?i:int):value
    ):int,
    4
  ):PRED
):PRED
```

3.7.2 Exists

The existential quantifier can be used to state the existence of at least one element. This element is defined by the expression and the optional predicates, which has to be fulfilled.

```
public class TestB{
  int[ ] testArray = {1, 2, 3};

  //@ requires (\exists int i; i < 3; arri[i] == 3);
  public TestB( ) {...}
}
```

The existential quantified expression of the `requires` clause declares that there exists at least one element equal to 3 in the array. The translated FOL term is given below:

Lookup.precondition.TestB

```
%exists [i:int]
%and(
  %int-pred predLT(?i:int, 3):PRED,
  %int-pred predEQ(
    %valueToInt(
      %arrSelect(
        ?heap:map,
        %valueToRef(
          %dynSelect(
            ?heap:map,
            ?this:ref,
            ?TestB?testArrayFieldSignature:ref
          ):value
        ):ref,
        ?i:int
      ):value
    ):int,
    3
  ):PRED
):PRED
```

3.8 JML Expressions

jml−primary ::= **result**−expression
| **old**−expression
| not−assigned−expression
| not−modified−expression
| only−accessed−expression
| only−assigned−expression
| only−called−expression
| only−captured−expression
| **fresh**−expression
| reach−expression
| duration−expression
| space−expression
| working−space−expression
| nonnullelements−expression
| informal−description
| **typeof**−expression
| elemtype−expression
| **type**−expression
| lockset−expression
| max−expression
| is−initialized−expression
| **invariant**−for−expression
| lblneg−expression
| lblpos−expression
| spec−quantified−expr

In JML annotations, it is not allowed to use one of the common Java operators such as ++, - - and the assignment operators, because they would cause side effects. There is a set of JML primary expressions that keeps Java side-effect free. In this section, we define the translation of the `result`, `old`, `fresh`, `typeof`, and `type` expressions.

3.8.1 Result

The `result` expression refers to the value returned by a non-void method. It can only be used in `ensures`, `duration`, and `workingspace` clauses. Therefore, we create a new variable that refers to the evaluated result value at run time. The name of the new variable consists of the unique string "\result:" connected with the sort of the return value type, for example "\result:int". The next example demonstrates the translation of the `result` annotation:

```
//@ ensures \result < 8;
public int foo( ) {
  return 3;
}
```

Lookup.postcondition.foo

%**int**−**pred** predLT(?\result:**int**, 8):**PRED**

3.8.2 Fresh

fresh−expression ::= **fresh** (spec−expression−list)

The `fresh` annotation states whether or not any object is allocated between the pre- and poststate of any routine. Since this feature puts a condition on the poststate, it can only be used in postconditions Ψ. The objects in the **spec-expression-lists** are declared but not yet allocated before the method's execution. The `fresh` feature asserts that these objects were freshly allocated in the body of a routine. We generate a term for every single object within the **spec-expression-lists**. The meaning of the term is: *"At the time of the method call, each listed object is already declared but not yet allocated, and will be allocated in the poststate"*. The term for a `fresh` annotation is:

$\bigwedge(i)$ ((x_i != null) ∧ !isAlive(pre_heap, x_i) ∧ isAlive(heap, x_i))

(x != null) expresses that the object was declared before the method's execution.

!isAlive(pre_heap, x) denotes that the object was not allocated before the method's execution.

isAlive(heap, x) expresses that the object is allocated after the method's execution.

The following example shows two declared objects in a `fresh` annotation:

```
public class Example{

    A a;
    B b;

    //@ ensures \fresh(a, b);
    public Example(){
      a = new A();
      b = new B();
    }
}
```

The FOL term of the entire `fresh` clause is:

Lookup.postcondition.foo

```
%and(
  %and(
    %and(
      %not(
        refEQ(?Example?aFieldSignature:ref, null):PRED
      ):PRED,
      %not(
        isAlive(?\pre_heap:map, ?Example?aFieldSignature:ref):PRED
      ):PRED
    ):PRED,
    isAlive(?heap:map, ?Example?aFieldSignature:ref):PRED
  ):PRED,
  %and(
    %and(
      %not(refEQ(?Example?bFieldSignature:ref, null):PRED):PRED,
      %not(
        isAlive(?\pre_heap:map, ?Example?bFieldSignature:ref):PRED
      ):PRED
    ):PRED,
    isAlive(?heap:map, ?Example?bFieldSignature:ref):PRED
  ):PRED
):PRED
```

3.8.3 Old

old−expression ::= **old** (spec−expression [, ident])
| **pre** (spec−expression)

The JML clause `old` refers to values of object fields before the routine's execution. In other words, the **spec-expression** list is evaluated in the method's or constructor's prestate. To access any field location in the prestate, we use the **pre_heap**, defined in section 2.4, within the heap selection method:

Heap.select(pre_heap, object, field);

This method is provided by the heap class. It returns a FOL term that yields the dynamic heap selection. For instance, Heap.select(pre_heap, this, count) generates the following term, whereas this:A and count:int:

```
%dynSelect(?\pre_heap:map, ?this:ref, ?A?countFieldSignature:int):value
```

To state the usage of the `old` clause, we set a boolean flag within the property object to *true*. The handling of an `old` clause requires four steps:

1. Upon reaching any `old` feature, set `old`-flag to *true*.
2. When visiting any variable, add prefix **pre_** to the variable name.
3. When visiting any field, use **pre_heap** instead of heap for dynamic heap access, if `old` flag is *true*.
4. When done visiting, reset `old` flag to *false*.

The `old` feature is not applicable to method's and constructor's parameters. These parameters always refer to their values in prestate. We decorate the first Java statement with `set` objects, each one representing a routine argument with its value of the prestate and the name with prefix "pre_"[10].

3.8.4 Typeof

typeof–expression ::= **typeof** (spec–expression)

JML offers a feature to get the most-specific dynamic type of an expression's value. The `typeof` clause represents the type of the containing expression. ESC/Java2 distinguish between reference types and non reference types:

Reference Types: We use the function symbol **typeof** to refer to an object's type in a given heap. The translated FOL term has the following syntax:

typeof(heap:sortMap**, var:**sortRef**):**sortType

We demonstrate the usage of the `typeof` feature on two examples:

\typeof(this)

typeof(?**heap**:**map**, ?**this**:**ref**):**type**

\typeof(testClass)

typeof(
  ?**heap**:**map**,
  %**valueToRef**(
    %**dynSelect**(?**heap**:**map**, ?**this**:**ref**, ?TestClass?testClassFieldSignature):**ref**
    ):**value**
  ):**ref**
):**type**

[10] See section 3.6.1 for the handling of `ghost` variables.

Non-Reference Types: All non-reference types are represented as a **TypeExpr** node that contains a field holding the type of the **spec-expression**. To represent a FOL term of a non reference type (e.g. integer, boolean), we generate a variable holding the **spec-expression** type in the name. Any translated FOL term of a non-reference `typeof` clause has the following syntax:

?<type-to-string>:type

The next examples show the usage of this annotation on a non-reference type, namely on a boolean type:

\typeof(true)

?boolean:**type**

3.8.5 Type

type−expression ::= **type** (**type**)

The `type` clause offers the ability to denote the type of a reference or primitive type. We handle this translation in similar fashion as with the `typeof` clause applied to non-reference types. We generate a new variable with the name of the given type. The sort of that variable is `sortType`. We get the following syntax for `type` clause translations:

?<type-to-string>:type

Using the `type` clause on a type called "MyNewTypeClass", we get the following translated FOL term:

?MyNewTypeClass:**type**

3.9 Type Specifications

jml−declaration ::= modifiers **invariant**
| modifiers **history−constraint**
| modifiers represents−decl
| modifiers **initially**−clause
| modifiers monitors−for−clause
| modifiers readable−if−clause
| modifiers writable−if−clause
| axiom−clause

JML offers a way to specify abstract data types, namely type specifications. They are useful to state any behavior for the plain data structure, as for example, `invariant` clauses have to hold in all visible states. Each instantiated object has to respect its type specification.

3.9.1 Initially

initially−clause ::= **initially** predicate

Any `initially` clause states what must hold after instantiating an object of this specific type. Thus, all non-helper constructors have to preserve predicates of `initially` clauses. In section 3.2.5, we currently restrict the expressiveness of `invariant`, `constraint` and also `initially` clauses. The predicates of an `initially` clause is only allowed to depend on fields that are defined in the class that declares the `initially` clause. The admissibility of `initially` clauses will be checked during the translation process[11]. We conjoin a translated `initially` predicate to each non-helper constructor's postcondition Ψ and exceptional postcondition Ψe. The next example shows the translation of a `initially` clause . The translated FOL term only gets conjoined to the second constructor since it is the only non-helper constructor.

```
1  public class A {
2    int x;
3    int y;
4
5    //@ initially x > 0;
6    //@ initially y != 2;
7
8    private /*@helper*/ A() {...}
9
10   public A(int z) {...}
11
12   public void foo() {...}
13
14 }
```

[11] See section 3.11 for more details about subset checking.

Lookup.postcondition.A (non-helper)

```
%and(
  %int−pred predGT(
    %valueToInt(
      %dynSelect(
        ?heap:map,
        ?this:ref,
        ?A?xFieldSignature:int
      ):value
    ):int,
    0
  ):PRED,
  %not(
    %int−pred predEQ(
      %valueToInt(
        %dynSelect(
          ?heap:map,
          ?this:ref,
          ?A?yFieldSignature:int
        ):value
      ):int,
      2
    ):PRED
  ):PRED
):PRED
```

3.9.2 History Constraint

history−constraint ::= **constraint**−keyword predicate [for constrained−list] ;
constraint−keyword ::= **constraint** | constraint_redundantly
constrained−list ::= method−name−list | **everything**

The `history constraint` feature belongs to JML level 1. We will simply call them `constraints`. `Constraints` are related to the object `invariants`. Similar to the behavior of `invariants`, to hold in all visible state, `constraints` act as relationships between several visible states. It is a combination of each visible state and any visible state later in the program's execution. The JML feature `old` is usually used to constrain the way how values may change over the whole program execution. One can optionally declare a list of methods (method-name-list). All listed methods have to fulfill the `constraint` predicates. If there is no method listed in a `constraint` clause, the default value "everything" will be used. In this case, all non-helper methods of the enclosing class have to fulfill their predicates. This optional argument is not yet available and will be referred as future work (see section 4.2). In the implementation of this thesis, we use the keyword "everything" and thus, every non-helper method consider the predicate of `constraint` clauses.

`Constraints` do not have to hold in the poststate of constructors because they do not have a prestate. Neither do they have to hold in destructors, because they do not have a poststate. In the case of an exceptional termination of a method, the `constraints` still have to hold for the current state. Thus, we conjoin the translated FOL term of `constraint` predicates to postcondition Ψ and to exceptional postcondition Ψe. Since `constraints` are related to `invariants`, we also do a subset check.

3.9.3 Example of History Constraint

The next example shows three routines of which only the last method has to fulfill the predicate of the `constraint` clause. The predicate gets translated into a FOL term and conjoined to that method's postcondition.

```
public class A {

  int count = 0;

  //@ constraint \old(count) > count;

  public A() {...}
  private /*@helper*/ doSomething() {...}
  public doNewStuff() {...}
}
```

Lookup.postcondition.doNewStuff

```
%int−pred predGT(
  %valueToInt(
    %dynSelect(?\pre_heap:map, ?this:ref, ?A?countFieldSignature:int):value
  ):int,
  %valueToInt(
    %dynSelect(?heap:map, ?this:ref, ?A?countFieldSignature:int):value
  ):int
):PRED
```

3.10 Invariant

invariant ::= **invariant**−keyword predicate ;
invariant−keyword ::= **invariant** | invariant_redundantly

An `invariant` describes a behavior that must hold in every visible state of an object. Object `invariants` specify the consistent states of objects. All object `invariants` in the system are assumptions in a routine's prestate and have to be proven in its poststate. Thus, `invariants` get generally conjoined to precondition Φ, postcondition Ψ, and exceptional postcondition Ψe. There are two exceptions: (1) constructors do not have to preserve the own object `invariants` in their prestate and (2) `helper` routines do not have to hold `invariants`, neither in their pre-, nor in their poststate. To simplify the proof of all `invariants` in the poststate, we only prove `invariants` of modified objects. Objects that are not modified, do not violate their `invariants`. Therefore, we handle the proof of `invariants` in the poststate in a different way. We use the predicate "inv(heap:map, obj:ref, t:type)"[12] to denote to proof the `invariant` of object x of type t. To translate `invariant` clauses, we meet three different cases that we have to handle in different:

- **Object Invariants:** We store all object `invariants` in the lookup table.
- **Preconditions:** To express the preservation of the object `invariant`, and of all generated object `invariants`, we generate a new predicate term and conjoin it to the precondition of all non-helper methods.
- **Postconditions:** Postconditions state a similar case as for method's preconditions. However, only `invariants` of generated and modified objects are relevant.

The next three sections describe these three cases in more details.

3.10.1 Object Invariant

Object `invariants` have to hold in all visible states of the class that declares the `invariants`. We restrict the expressiveness of `invariant` predicates, since they are only allowed to depend on fields of the enclosing class. The admissibility of `invariant` predicates is checked by a subset check. The subset check is an optional feature and can be turned on and off by a command line entry. We discuss the subset check for `invariant`, `initially` and `constraint` predicates in chapter 3.11. A subset check either returns the boolean value *true* for a successful check or *false* otherwise. After a successful subset check for `invariant` predicate, we conjoin the translated FOL term of the predicate to the specific lookup table entry. We do not conjoin the ξ(e) to any pre-, post-, or exceptional postcondition of a routine. We have introduced a new predicate "inv(heap, object, type)" that refers to the invariant term in the lookup table for a given object type.
The next example shows the translation of one `invariant` clause.

[12] See section 3.3.3 for more details about the predicate "inv".

```
1 public class A {
2   B b;
3   //@ invariant b != null;
4
5 }
```

Lookup.invariant.A

```
%not(
  refEQ(
    %valueToRef(
      %dynSelect(?heap:map, ?this:ref, ?A?bFieldSignature:ref):value
    ):ref,
    null
  ):PRED
):PRED
```

3.10.2 Invariant Term for Precondition Φ

A method has to guarantee the satisfaction of all object `invariants` in its prestate. In this context, "all" refers to every allocated object of any type in the whole system. To express this, we generate the following term:

$$\forall\ \mathbf{r{:}ref,\ t{:}type\ (isAssignCompat(heap,\ r,\ t)} \wedge \mathbf{\ isAlive(heap,\ r)} \rightarrow \mathbf{\ inv(heap,\ r,\ t))}$$

Conjoining this term to precondition Φ of all non-helper methods, we can rely on the condition that all `invariants` have to hold in their prestate. The reader should pay attention to the fact that the `invariant` predicate of the enclosing class does not get conjoined to the method's precondition.

The treatment of `invariants` in constructor prestates is analogous to that of methods, except that the class declaring the constructor is excluded in the generated term since its `invariant` has not yet been established. Thus, we conjoin the following term to a constructor's precondition, in which the "this" construct denotes the object itself:

$$\forall\ \mathbf{r{:}ref,\ t{:}type\ (isAssignCompat(heap,\ r,\ t)} \wedge \mathbf{\ isAlive(heap,\ r)} \wedge \mathbf{\ (r\ !{=}\ this)} \rightarrow \mathbf{\ inv(heap,\ r,\ t))}$$

The next example shows the usage of the explained terms in method's and constructor's preconditions:

```
1  public class A {
2    int x;
3    //@ invariant x != null;
4
5    //@ requires true;
6    public A() {...}
7
8    //@ requires true;
9    public void foo() { ...  }
10
11 }
```

Lookup.invariant.A

```
%not(
  %int−pred predEQ(
    %valueToInt(
      %dynSelect(?heap:map, ?this:ref, ?A?xFieldSignature:int):value
    ):int,
    0
  ):PRED
):PRED
```

Lookup.precondition.A

```
%and(
  %isTrue(true):PRED,
  %forAll [#r1:ref, #x0:type]
  %implies(
    %and(
      %and(
        %isAlive(?heap:map, ?#r1:ref):PRED,
        %assignCompat(?heap:map, ?#r1:ref, ?#x0:type):PRED
      ):PRED,
      %not(
        refEQ(
          ?#r1:ref,
          ?this:ref
        ):PRED
      ):PRED
    ):PRED,
    %inv(?heap:map, ?#r1:ref, ?#x0:type):PRED
  ):PRED
):PRED
```

Lookup.precondition.foo

```
%and(
  %assignCompat(
    ?heap:map,
    ?this:ref,
    ?(ReferenceType (ClassType A.className)):type):PRED,
  %forAll [#r9:ref, #x6:type]
  %implies(
    %and(
      %isAlive(?heap:map, ?#r9:ref):PRED,
      %assignCompat(?heap:map, ?#r9:ref, ?#x6:type):PRED
    ):PRED,
    %inv(?heap:map, ?#r9:ref, ?#x6:type):PRED
  ):PRED
):PRED
```

3.10.3 Invariant Term for Postcondition Ψ

Each routine has to satisfy the `invariants` of all objects in its poststate. As the routine's precondition already guarantees the `invariant` of all objects, we only have to consider object that might have been modified. This is evaluated by using a static analysis that makes an overestimation by collecting all types that can be assigned to. Only `invariants` of modifiable objects could have been broken during the execution of that routine. We generate the following term that we conjoin to each routine's post- and exceptional postconditions:

$$\forall\ \textbf{r:ref, t:type (isAssignCompat(heap, r, t)} \wedge \textbf{isAlive(heap, r)} \wedge \textbf{isVisibleIn(t, store-ref-list)} \rightarrow \textbf{inv(heap, x, t))}$$

The special notation "isVisibleIn(t, store-ref-list)"[13] compares the quantified object reference "r" with each type that can be modified during the execution of the routine. We need a list of all modifiable objects in a method. The method for collecting all modifiable objects will be described in more detail in section 3.10.4 (Type Collector). The next example shows the usage of the explained term. Below the example, the method's postcondition consisting of the own postcondition and the newly generated term is shown:

```
1  public class A {
2    B b = new B();
3    //@ invariant b != null;
4
5    //@ ensures true;
6    public void foo() { }
7
8  }
```

[13] See section 3.3.4 for more details about the "isVisibleIn" notation.

Lookup.invariant.A

```
%not(
  refEQ(
    %valueToRef(
      %dynSelect(
        ?heap:map,
        ?this:ref,
        ?A?bFieldSignature:ref
      ):value
    ):ref,
    null
  ):PRED
):PRED
```

Lookup.postcondition.foo

```
%and(
  %isTrue(true):PRED,
  %forAll [#r10:ref, #x7:type]
  %implies(
    %and(
      %and(
        %isAlive(?heap:map, ?#r10:ref):PRED,
        %assignCompat(?heap:map, ?#r10:ref, ?#x7:type):PRED
      ):PRED,
      %anyEQ(
        ?#x7:type,
        ?(ReferenceType (ClassType A.className)):type
      ):PRED
    ):PRED,
    %inv(?heap:map, ?#r10:ref, ?#x7:type):PRED
  ):PRED
):PRED
```

3.10.4 Type Collector

We use a type collector to gain all modifiable object types within a routine. All collected types are stored in a set data structure and are reused in the special notation "isVisibleIn"[14]. This special notation compares each type of the set with the quantified variable type in order to check the `invariant` of those collected types. Objects that are not modified, do not violate their own `invariants` if they hold in the prestate of a routine. Thus, we restrict the scope of possible objects and only demand the `invariants` of modifiable objects in the poststate to be checked. To range this scope, we collect all modifiable objects types within a routine. Our algorithm covers the collection of all modifiable objects within a certain routine.

[14] See section 3.3.4 for more details about this special notation.

For the collection part, we use a separate visitor to avoid any interference with the common visitor as shown in figure 3.5.

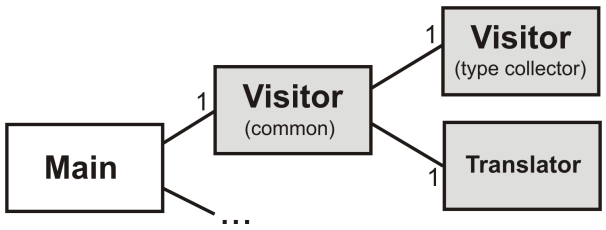

Figure 3.5: Class diagram of visitors and translator class

Modifying an object takes place in a body of a routine that consists of several statements. For this part, we ignore JML annotations and treat assignable expressions and method invocations therefore. That's the reason why we split up the algorithm and use a new class as an independent visitor.
We first describe the algorithm to collect object types of assignable expressions and later the handling of method invocations.

Assignable Expressions:

Now, we take a deeper look at the modification of an object. There is only one way to modify an object, namely by using any assignable expression. Assignable expressions are binary statement expressions and possess a left and a right part. The left part consists of a target and a field as illustrated in figure 3.6.

Figure 3.6: A field access requires a target and a field

Our algorithm collects the type of the target object. There are nine different assignment expressions (assign, asgmul, asgdiv, asgrem, asgadd, asgsub, asgrshift, asgurshift, asgbitand). To state, that we need the type of the left part, we use a boolean flag, called "assign", within the property object and assign the default value *true*. For the right part, we assign the value *false*.
If we reach one of these assignable expression, we do the following steps:

1. Upon reaching assignable expression, set flag to *true*.
2. Visit the left part of the assignment.
3. When done, visiting left part, set flag to *false*.
4. Visit the right part of the assignment .

These four steps lay the foundation for the collecting part. The algorithm visits both sides, because there could be another assignable expression on the right side. The main type collection happens during visiting field accesses. The algorithm checks if the flag is equal to *true*. In that case, we are visiting the left part of an assignable expression and we collect the typeof the field access type by storing it to the overall type collection set. But anyway, we have to visit the target of a field access as well, since there could be another assignable expression. This special case is demonstrated in the next example:

```
(a.b1 = b2).count = 3;
```

There are two object modifications in this example. The field b of object a:A and the field count of object a.b2:B get modified. Thus, the collection type set contains [A, B, THIS] where "THIS" refers to the own object type. The own object type is added as a default state since the own `invariant` has to be checked as well.

Method Invocations:

Consider a method call between a caller and a callee. The callee may again modify some objects, and the caller also has to guarantee the satisfaction of those types `invariants`. The `assignable` clause of a routine defines, which object might be modifiable within that method. We add all object types of the `assignable` reference list of the callee into the callers collection type set. As we have seen in chapter 3.5.5, there are three different kinds of store-ref-lists in an `assignable` clause. We take a look at each one of them, since they are treated separately:

1. **\nothing:** The callee does not modify any object and thus no `invariant` might be violated. In this case, we add no object type to our type set.

2. **\everything:** Using the "everything" keyword, which is in fact the default value if omitted, we have to check the `invariant` of all allocated objects in the system. In theory, we could break up the collection part at this point, but since this is not possible, we state this behavior by setting a flag, called "everything", in the property object. At the very last step, before we generate the term, we check this flag. In that described case, we generate the following term:

   ```
   ∀ o:ref, t:type (isAlive(heap, o) ∧ isAssignCompat(heap, o, t)
                                          → inv(heap, o, t))
   ```

3. **ref-loc:** In this case, we just add all listed object types to our collecting type **set**. All `invariants` within this type set get checked afterwards. The generating FOL term looks like:

   ```
   ∀ o:ref, t:type (isAlive(heap, o) ∧ isAssignCompat(heap, o, t)
                                  ∧ isVisibleIn(o, set) → inv(heap, o, t))
   ```

Example of Type Collection

```
public class Test{
  A a;
  B b;
  C c;
  D d;

  //@ assignable a, b, c;
  public void foo(){
    a.count = 8;
    doSomething();
  }

  //@ assignable d;
  public void doSomething(){...}

}
```

This example demonstrates the collection and the translation of modified object types. We consider the method "foo()" in the class "Test". Our goal is to generate an `invariant` term for the postcondition of that method. The special notation "isVisibleIn" collects all modifiable objects within the method "foo()" and all heap locations in the `assignable` clauses of invoking methods. The own object type [Test] is added to the `assignable` set as a default state, because the own object `invariant` has to hold in all visible states of object "foo()".

Assignable Objects: The method "foo()" contains one assignable expression and thus, one modifiable object type. We do not consider the `assignable` clause of the "foo()" method. Newly added type set is [A].

Method Invocations: By calling the method "doSomething()", we add the object types of all heap locations within that `assignable` clause. Newly added type set is [D].

After collecting all modifiable types within that method, we get the following set: **[Test, A, D]**. The "isVisibleIn(t, {Test, A, D})" predicate looks like:

$$(t = \mathbf{Test}) \vee (t = \mathbf{A}) \vee (t = \mathbf{D})$$

The overall **invariant** term for the postcondition is:

Lookup.postcondition.foo

```
%forAll [#r:ref, #t:type]
%implies(
  %and(
    %and(
      %isAlive(?heap:map, ?#r:ref):PRED,
```

```
      %assignCompat(?heap:map, ?#r:ref, ?#t:type):PRED
    ):PRED,
    %or(
      %or(
        %anyEQ(?#t:type, ?(ReferenceType (ClassType Test.className)):type):PRED,
        %anyEQ(?#t:type, ?(ReferenceType (ClassType A.className)):type):PRED
      ):PRED,
      %anyEQ(?#t:type, ?(ReferenceType (ClassType D.className)):type):PRED
    ):PRED
  ):PRED,
  %inv(?heap:map, ?#r:ref, ?#t:type):PRED
):PRED
```

3.11 JML Subset Checker

```
//@ invariant <P>
//@ initially <P>
//@ constraints <P>
```

Predicates of type specifications as `invariant`, `initially`, and `constraint` can be any kind of expression. We restrict the expressiveness of these predicates. See [15] for more details about `invariant` field dependences on superclass fields. Predicates of `invariant`, `initially`, or `constraint` clauses are only allowed to depend on fields that are defined in the class that declares these specifications. The admissibility of those clauses will be checked during the translation process. We provide a new option to manually switch on and off. The options name is either **doSubsetChecking** or the shortcut **dsc**. The subset checking method gets involved before the predicate gets translated. By calling the subset checking method, we receive a boolean value whether or not all field accesses follow our restriction. In the case of a failing subset check, we inform the user by a message of a similar form as the following:
Subset checking: failed! The field "count" is a field of class "O", and not as expected of class "A"!;
In this example, the `invariant` accesses a field "count" of an object of type "O". We do the check for every single type specification predicate. Thus, we collect and store all field accesses of these predicates in a hash map in order to do the subset check.

Subset checking method
We have access to the own class declaration and to the collected field accesses of the object `invariant` expression. By comparing the parent of every single field access and the own class declaration, we find out whether or not the field access is allowed. If there is a fault, ESC/Java2 prints out the explained messages above and return a boolean value to express the failing of the check. In a successful check, we continue the usual way as conjoining the translated FOL term to the type specification.

3.11.1 Example of Subset Checker

We do a subset check for the following type specification.

```
public class A {
  B b;
  int count = 0;

  //@ invariant b.num > 4;
  //@ initially  count == 0;
  //@ constraint count > 0;
}

public class B { int num;}
```

The generated output is:
Subset checking: failed! The field "num" is a field of class "B", and not as expected of class "A"!;

3.12 Summary of Translations

In this chapter, we have explained the translation of most of JML level 0 features. Table 3.2 illustrates the handling of the type and method specifications. There is given a Java code example in appendix A and the translated FOL terms for each constructor and method.

	non-helper constructor		`helper` constructor		non-helper method		`helper` method	
	Φ	Ψ,Ψe	Φ	Ψ,Ψe	Φ	Ψ,Ψe	Φ	Ψ,Ψe
`requires`	√	×	√	×	√	×	√	×
`ensures`	×	√	×	√	×	√	×	√
`signals`	×	√	×	√	×	√	×	√
`invariant`	×	√	×	×	√	√	×	×
`initially`	×	√	×	×	×	×	×	×
`constraint`	×	×	×	×	×	√	×	×
`assignable`	×	√	×	×	×	√	×	×
`fresh`	×	√	×	×	×	√	×	×

Table 3.2: Handling of type and method specifications

4 Conclusion and Future Work

4.1 Conclusion

We have presented the translation of JML annotations into FOL terms. We first defined and then implemented the translation that now works in Mobius PVE. The translation operation supports most JML level 0 features, and in addition, history constraints, a JML level 1 feature.

We did some simplifications to keep the translation modular and sound. One simplification is applied to field accesses. We allow type specifications only to depend on fields that are defined in the class declaring the specification.

Thus, we could keep the modularity. In order to be sound, we have to check the admissibility of the type specifications, that can be turned on and off.

4.2 Future Work

4.2.1 Implementation of missing JML level 0 Translations

The current translation does not cover all JML level 0 annotations. There are still some few annotations left, which are required for the full version of the MOBIUS project. Section 2.1.1 discloses the missing annotations. Some JML annotations have optional features. The **constraint** clauses provide an optional method to list, which non-helper methods have to fulfill the `constraint` predicate. `Assume` and `assert` annotations have an optional error message, that will be printed if the assertion fails. Both optional features are not implemented yet.

4.2.2 Assignable Semantics

In this thesis we treat the semantics of `assignable` clauses in the same way as of `modifiable` clauses. The semantics of a `modifiable` clause allows objects to be modifiable within the body of the routine. Thus, assign and re-establish the old value to any object field is allowed for objects mentioned in the `modifiable` clause. We check this behavior not until the routine's termination. The JML semantics of the `assignable` clause however forces to check the assign permission before every particular assignable execution. This semantics can be translated by decorating every field update by an `assert` annotation. This annotation yields a term that checks if the `assignable` heap location is mentioned in the `assignable` clause.

4.2.3 Different Frontend/Backend

To scan a Java code related AST, we use a visitor pattern. If the Mobius PVE is using a different JavaFE at any time, the visitor pattern needs only some few changes in order to work with that new JavaFE. Even if the backend of ESC/Java2 will change in future, there are only some few changes needed to work again.

4.2.4 Acknowledgements

I would like to thank my supervisor Hermann Lehner and Prof. Peter Müller and all reviewers for helpful comments. Special thanks go to my family, who gave me support during my whole studies at the ETH and made my dreams true. Thanks for everything!

Bibliography

[1] Patrice Chalin. Early detection of jml specification errors using esc/java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 25–32, New York, NY, USA, 2006. ACM Press.

[2] Mobius Consortium. Deliverable 6.1: Dissemination and training plan. Available online from http://mobius.inria.fr, March 2006.

[3] INRIA: http:///www.inria.fr.

[4] Software Component Technology Group of ETH Zurich: http://se.inf.ethz.ch.

[5] ETH Zurich: http://www.ethz.ch.

[6] A. Schubert and J. Chrząszcz. ESC/Java2 as a tool to ensure security in the source code of Java applications. In *Software Engineering Techniques: Design for Quality*, IFIP, Warsaw, 2006. Springer-Verlag.

[7] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[8] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.

[9] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, mars 2004. http://coq.inria.fr/doc/main.html.

[10] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.

[11] The Java Programming Language: http://java.sun.com/.

[12] The Java Runtime Environments: http://java.sun.com/j2se/desktopjava/jre/index.jsp.

[13] The Java Modeling Language: http://jmlspecs.org.

[14] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[15] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

A Translation Example of most JML Level 0 Features

This appendix shows the translation of JML annotations: `invariant`, `constraint`, `initially`, `helper`, `requires`, `ensures` and `old`.

```
public class One {
  int count = 2;

  //@ invariant count > 0;
  //@ constraint \old(count) > count;
  //@  initially  count != 0;

  //@ requires count > 0;
  private /*@helper*/ One(){ }

  //@ ensures count < 3;
  public One(int e){ }

  private /*@helper*/ void foo1() { }

  //@ ensures \old(count) > count;
  public  void foo2() { }
}
```

Lookup.precondition.One (with /*@helper*/)

```
%int−pred predGT(
  %valueToInt(
    %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
  ):int,
  0
):PRED
```

Lookup.precondition.One (without /*@helper*/)

```
%and(
  %isTrue(true):PRED,
  %forAll [#r2:ref, #x0:type]
  %implies(
    %and(
      %and(
        %isAlive(?heap:map, ?#r2:ref):PRED,
        %assignCompat(?heap:map, ?#r2:ref, ?#x0:type):PRED
      ):PRED,
      %not(refEQ(?#r2:ref, ?this:ref):PRED):PRED
    ):PRED,
    %inv(?heap:map, ?#r2:ref, ?#x0:type):PRED
  ):PRED
):PRED
```

Lookup.precondition.foo

```
%isTrue(true):PRED
```

Lookup.precondition.foo2

```
%and(
  %isTrue(true):PRED,
  %forAll [#r8:ref, #x7:type]
  %implies(
    %and(
      %isAlive(?heap:map, ?#r8:ref):PRED,
      %assignCompat(?heap:map, ?#r8:ref,?#x7:type):PRED
    ):PRED,
    %inv(?#r8:ref, ?#x7:type):PRED
  ):PRED
):PRED
```

Lookup.postcondition.One (with /*@helper*/)

```
%isTrue(true):PRED
```

Lookup.postcondition.One (without /*@helper*/)

```
%and(
  %and(
    %int−pred predLT(
      %valueToInt(
        %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
      ):int,
      3
    ):PRED,
    %forAll [#r3:ref, #x1:type]
    %implies(
      %and(
        %and(
          %isAlive(?heap:map, ?#r3:ref):PRED,
          %assignCompat(?heap:map, ?#r3:ref, ?#x1:type):PRED
        ):PRED,
        %anyEQ(
          ?#x1:type,
          ?(ReferenceType (ClassType One.className)):type
        ):PRED
      ):PRED,
      %inv(?heap:map, ?#r3:ref, ?#x1:type):PRED
    ):PRED
  ):PRED,
  %not(
    %int−pred predEQ(
      %valueToInt(
        %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
      ):int,
      0
    ):PRED
  ):PRED
):PRED
```

Lookup.postcondition.foo

```
%isTrue(true):PRED
```

Lookup.postcondition.foo2

```
%and(
  %and(
    %implies(
      %assignCompat(
        ?heap:map,
        ?this:ref,
        ?(ReferenceType (ClassType One.className)):type
      ):PRED,
```

```
      %int−pred predGT(
        %valueToInt(
          %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
        ):int,
        %valueToInt(
          %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
        ):int
      ):PRED
    ):PRED,
    %forAll [#r8:ref, #x4:type]
    %implies(
      %and(
        %and(
          %isAlive(?heap:map, ?#r8:ref):PRED,
          %assignCompat(?heap:map, ?#r8:ref, ?#x4:type):PRED
        ):PRED,
        %anyEQ(
          ?#x4:type,
          ?(ReferenceType (ClassType OneType.className)):type
        ):PRED
      ):PRED,
      %inv(?heap:map, ?#r8:ref, ?#x4:type):PRED
    ):PRED
  ):PRED,
  %int−pred predGT(
    %valueToInt(
      %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int,
    %valueToInt(
      %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int
  ):PRED
):PRED
```

Lookup.exceptionalpostconditions.One (with /*@helper*/)

```
%implies(
  %assignCompat(
    ?heap:map,
    ?#r0:ref,
    ?(ReferenceType (ClassType java.lang.ExceptionType.className)):type
  ):PRED,
  %implies(
    %int−pred predGT(
      %valueToInt(
        %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
      ):int,
      0
    ):PRED,
```

```
    %isTrue(false):PRED
  ):PRED
):PRED
```

Lookup.exceptionalpostconditions.One (without /*@helper*/)

```
%and(
  %and(
    %implies(
      %assignCompat(
        ?heap:map,
        ?this:ref,
        ?(ReferenceType (ClassType OneType.className)):type
      ):PRED,
      %int−pred predGT(
        %valueToInt(
          %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
        ):int,
        %valueToInt(
          %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
        ):int
      ):PRED
    ):PRED,
    %forAll [#r8:ref, #x4:type]
    %implies(
      %and(
        %and(
          %isAlive(?heap:map, ?#r8:ref):PRED,
          %assignCompat(?heap:map, ?#r8:ref, ?#x4:type):PRED
        ):PRED,
        %anyEQ(
          ?#x4:type,
          ?(ReferenceType (ClassType OneType.className)):type
        ):PRED
      ):PRED,
      %inv(?heap:map, ?#r8:ref, ?#x4:type):PRED
    ):PRED
  ):PRED,
  %int−pred predGT(
    %valueToInt(
      %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int,
    %valueToInt(
      %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int
  ):PRED
):PRED
```

Lookup.exceptionalpostconditions.foo1

```
%implies(
  %assignCompat(
    ?heap:map,
    ?#r5:ref,
    ?(ReferenceType (ClassType java.lang.ExceptionType.className)):type
  ):PRED,
  %isTrue(false):PRED
):PRED
```

Lookup.exceptionalpostconditions.foo2

```
%and(
  %and(
    %implies(
      %assignCompat(
        ?heap:map,
        ?#r6:ref,
        ?(ReferenceType (ClassType java.lang.ExceptionType.className)):type
      ):PRED,
      %implies(
        %assignCompat(
          ?heap:map,
          ?this:ref,
          ?(ReferenceType (ClassType OneType.className)):type
        ):PRED,
        %isTrue(false):PRED
      ):PRED
    ):PRED,
    %forAll [#r9:ref, #x5:type]
    %implies(
      %and(
        %and(
          %isAlive(?heap:map, ?#r9:ref):PRED,
          %assignCompat(?heap:map, ?#r9:ref, ?#x5:type):PRED
        ):PRED,
        %anyEQ(
          ?#x5:type,
          ?(ReferenceType (ClassType OneType.className)):type
        ):PRED
      ):PRED,
      %inv(?heap:map, ?#r9:ref, ?#x5:type):PRED
    ):PRED
  ):PRED,
  %int−pred predGT(
    %valueToInt(
      %dynSelect(?\pre_heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int,
```

```
    %valueToInt(
      %dynSelect(?heap:map, ?this:ref, ?One?countFieldSignature:int):value
    ):int
  ):PRED
):PRED
```

B JML Predicate Syntax

jml−predicate−keyword ::= \TYPE
| \bigint | \bigint_math | \duration
| \elemtype | **everything** | **exists**
| **forall** | **fresh**
| \into | \ invariant_for | \ is_initialized
| \java_math | \lblneg | \lblpos
| \lockset | \max | \min
| \nonnullelements | \not_assigned
| \not_modified | \ not_specified
| \nothing | \nowarn | \nowarn_op
| \num_of | **old** | \only_accessed
| \only_assigned | \only_called
| \only_captured | **pre**
| \product | \reach | \real
| **result** | \same | \safe_math
| \space | \such_that | \sum
| **typeof** | **type** | \warn_op
| \warn | \working_space
| jml−universe−pkeyword
jml−universe−pkeyword ::= \peer | \readonly | \rep
jml−keyword ::= abrupt_behavior | abrupt_behaviour
| accessible | accessible_redundantly
| also | assert_redundantly
| **assignable** | assignable_redundantly
| **assume** | assume_redundantly | axiom
| behavior | behaviour
| breaks | breaks_redundantly
| callable | callable_redundantly
| captures | captures_redundantly
| choose | choose_if
| code | code_bigint_math |
| code_java_math | code_safe_math
| **constraint** | constraint_redundantly
| constructor | continues | continues_redundantly
| decreases | decreases_redundantly
| decreasing | decreasing_redundantly
| diverges | diverges_redundantly
| duration | duration_redundantly
| **ensures** | **ensures_redundantly** | example
| exceptional_behavior | exceptional_behaviour

| exceptional_example
| **exsures** | **exsures_redundantly** | extract
| field | **forall**
| for_example | **ghost**
| **helper** | hence_by | hence_by_redundantly
| implies_that | in | in_redundantly
| initializer | **initially** | instance
| **invariant** | invariant_redundantly
| **loop_invariant** | **loop_invariant_redundantly**
| **maintaining** | **maintaining_redundantly**
| maps | maps_redundantly
| measured_by | measured_by_redundantly
| method | model | model_program
| modifiable | modifiable_redundantly
| modifies | modifies_redundantly
| monitored | monitors_for | non_null
| normal_behavior | normal_behaviour
| normal_example | nowarn
| nullable | nullable_by_default
| **old** | or
| **post** | **post_redundantly**
| **pre** | **pre_redundantly**
| pure | readable
| refine | refines | refining
| represents | represents_redundantly
| **requires** | **requires_redundantly**
| returns | returns_redundantly
| **set** | **signals** | **signals_only**
| signals_only_redundantly | **signals_redundantly**
| spec_bigint_math | spec_java_math
| spec_protected | spec_public | spec_safe_math
| static_initializer | uninitialized
| unreachable | weakly
| when | when_redundantly
| working_space | working_space_redundantly
| writable
| peer
| readonly
| rep

C Expression Syntax

Syntax of an expression:

```
expression ::= assignment−expr
assignment−expr ::= conditional−expr
                    [ assignment−op assignment−expr ]
assignment−op ::= = | += | −= | ∗= | /= | %= | >>=
        | >>>= | <<= | &= | '|=' | ^=
conditional−expr ::= equivalence−expr
                     [ ? conditional−expr : conditional−expr ]
equivalence−expr ::= implies−expr
                     [ equivalence−op implies−expr ] ...
equivalence−op ::= <==> | <=!=>
implies−expr ::= logical−or−expr
                 [ ==> implies−non−backward−expr ]
        | logical−or−expr <== logical−or−expr
                 [ <== logical−or−expr ] ...
implies−non−backward−expr ::= logical−or−expr
                 [ ==> implies−non−backward−expr ]
logical−or−expr ::= logical−and−expr [ '||' logical−and−expr ] ...
logical−and−expr ::= inclusive−or−expr [ && inclusive−or−expr ] ...
inclusive−or−expr ::= exclusive−or−expr [ '|' exclusive−or−expr ] ...
exclusive−or−expr ::= and−expr [ ^ and−expr ] ...
and−expr ::= equality−expr [ & equality−expr ] ...
equality−expr ::= relational−expr [ == relational−expr] ...
        | relational−expr [ != relational−expr] ...
relational−expr ::= shift−expr < shift−expr
        | shift−expr > shift−expr
        | shift−expr <= shift−expr
        | shift−expr >= shift−expr
        | shift−expr <: shift−expr
        | shift−expr [ instanceof type−spec ]
shift−expr ::= additive−expr [ shift−op additive−expr ] ...
shift−op ::= << | >> | >>>
additive−expr ::= mult−expr [ additive−op mult−expr ] ...
additive−op ::= + | −
mult−expr ::= unary−expr [ mult−op unary−expr ] ...
mult−op ::= ∗ | / | %
unary−expr ::= ( type−spec ) unary−expr
        | ++ unary−expr
        | −− unary−expr
        | + unary−expr
```

```
        | − unary−expr
        | unary−expr−not−plus−minus
unary−expr−not−plus−minus ::= ~ unary−expr
        | ! unary−expr
        | ( built−in−type ) unary−expr
        | ( reference−type ) unary−expr−not−plus−minus
        | postfix−expr
postfix−expr ::= primary−expr [ primary−suffix ] ... [ ++ ]
        | primary−expr [ primary−suffix ] ... [ −− ]
        | built−in−type [ '[' ']' ] ... . class
primary−suffix ::= . ident
        | . this
        | . class
        | . new−expr
        | . super ( [ expression−list ] )
        | ( [ expression−list ] )
        | '[' expression ']'
        | [ '[' ']' ] ... . class
primary−expr ::= ident | new−expr
        | constant | super | true
        | false | this | null
        | ( expression )
        | jml−primary
built−in−type ::= void | boolean | byte
        | char | short | int
        | long | float | double
constant ::= java− literal
new−expr ::= new type new−suffix
new−suffix ::= ( [ expression−list ] ) [ class−block ]
        | array−decl [ array− initializer ]
        | set−comprehension
array−decl ::= dim−exprs [ dims ]
dim−exprs ::= '[' expression ']' [ '[' expression ']' ] ...
array− initializer ::= { [ initializer [ , initializer ] ... [ , ] ] }
initializer ::= expression
        | array− initializer
```