Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jive/JML

Generating Proof Obligations

From JML Specifications

Ghislain Fourny
gfourny@student.ethz.ch

Swiss Federal Institute of Technology Zurich
Software Component Technology Group

Semester Project
March 2005

Supervisors:
Professor: Prof. Dr. Peter Müller
Assistant: Ádám Darvas

Table of Contents

Abstract

The new version on Jive takes JML-annotated Java programs as input. After an introduction to Jive and details about the JML specification and implementation, this paper describes the internal implementation of the Proof Obligation Generator in Jive's front-end, as well as its external (input and output) interfaces. We then give some examples of code we performed tests against.

If you would like a quick start, you are advised to jump directly to part III, paragraph 3.7.

I. Introduction

1. Behavioral interface specification

The first generations of information systems were relatively small, so that one could write and debug programs while having a high control of the situation. Should one find a flaw, one would just try to correct it and test again.

Over the past few decades, computer got used everywhere, including where bugs could have severe consequences, either because human lives are at stake, or for financial reasons. Not only does this mean that the final version of a system should have no bugs, but also that one does not even have the opportunity to test and debug. You cannot (yet) send a robot on mars for each intermediate version of each module of each programmer. The first version should be final, and tests can only be simulated.

The software used in control towers for flight navigation is still written in old languages like Fortran or Cobol: who would like to take the responsibility to convert this in a safer, more modern and efficient language like Java or C++, possibly introducing flaws? And even if we keep these respectable languages, only statistics protect us: if only a limited number of bugs occurred in the last thirty years, one hopes it will remain stable in the future.

Programs are so large, that no human being can have a sufficiently clear overview of it to be able to say: this will work the way we want it to. Computers? Merely knowing whether a program will terminate instead of looping, or whether a given line of code will ever be used, has been proved undecidable!

Though, there still remains a solution: cooperation between programmers and computers. Programmers subdivide their work in modules and annotate each method or variable in their language. If each method exactly does what it should, the whole program should work as intended.

Hence if a computer could process each method and show that it behaves according to its annotation, the program would be proven correct, and the only program to be proven “by hand” would be this unique prover. This requires that the computer understands our language, which we know to have its flaws and ambiguities. Hence, and this is the basic idea of Jive at its beginning, the programmer could write annotations in a formal, machine-checkable language, unambiguously.

Annotations could be contracts between the implementer of the method and its caller. If a certain precondition holds, which the caller has to make sure and the implementer can assume, then the implementer’s job is to make sure a certain postcondition will hold after termination of the method, which the caller then can assume.

2. Introduction to Jive

In this part, we explain shortly the architecture of Jive. More information can be found in [5].

Jive stands for Java Interactive Verification Environment. The programmer annotates his programs and these annotations are processed by Jive. This leads to safer software for at least two reasons:
- Because the programmer using such annotations has to think about correctness of his programs.
- Because these annotations are then converted to proof obligations by Jive, which are sent to a theorem prover.

Jive is bound to the Java language. This can be considered a drawback, since it would have to be re-implemented, should one want to reuse it in another (not Java-like) language. But it also is a great advantage, since it knows Java in depth.

In its first version, Jive relied on first-order logic for this is the standard as an input to theorem provers. Programmers were also required to enter their annotations in first-order logic, so that the computer could interpret preconditions and postconditions. This means that lots of developers would have to learn a new, abstract, syntax, and a way of thinking which does not correspond to the way they program.

Hence, a compromise has to be found, and this compromise is JML. JML is a behavioral interface specification language developed by Gary Leavens and his group at the Iowa State University. The expressive power of JML is potentially at least the same as first-order logic, however its syntax is pretty close to that of Java. JML also allows one to specify invariants.

The second version of Jive will take JML-annotated Java programs as input. The front end (proof obligations and invariants generation) is handled by the Software Component Technology group, and the backend (proving) by the University of Kaiserslautern. The aim of the project, and the topic of this article, is to describe the part of the front end which converts JML specifications into proof obligations in first-order logic, the Proof Obligation Generator.

3. Related work in program correctness

If Jive is able to prove a specification true, it does not give much assistance to the user should the proof fail.

In the state-of-the-art, it is possible to generate counter-examples showing that the specification does not hold in the case of model checking. Work is in progress to do the same in theorem proving.

Research is also been carried out in how to detect common programming errors, such as null pointers, illegal array accesses. One example of such a tool is ESC (Extended Static Checker). It is above the decidability ceiling, so that the approach is neither sound nor complete. But it can be of good help to programmers in the debugging stage. It is explained in [8].

Another topic of research is about type universes. Each type is associated with three "modifiers": peer, rep and read-only, which organize instances of classes in universes. Each universe is owned by one object, which is the only one outside the universe which has write access rights on elements in this universe. Element in a universe have also mutual write access rights. This provides a framework for a better encapsulation for groups of cooperating objects [9].

II. JML Background

1. JML Basics

The Java Modeling Language is a behavioral interface specification language. Behavioral, because it describes rigorously the way a method behaves, in terms of contract between the user and the programmer. Interface, for it describes - just like Java, actually completing it - what methods give access to a given object and how they are to be used (signatures).

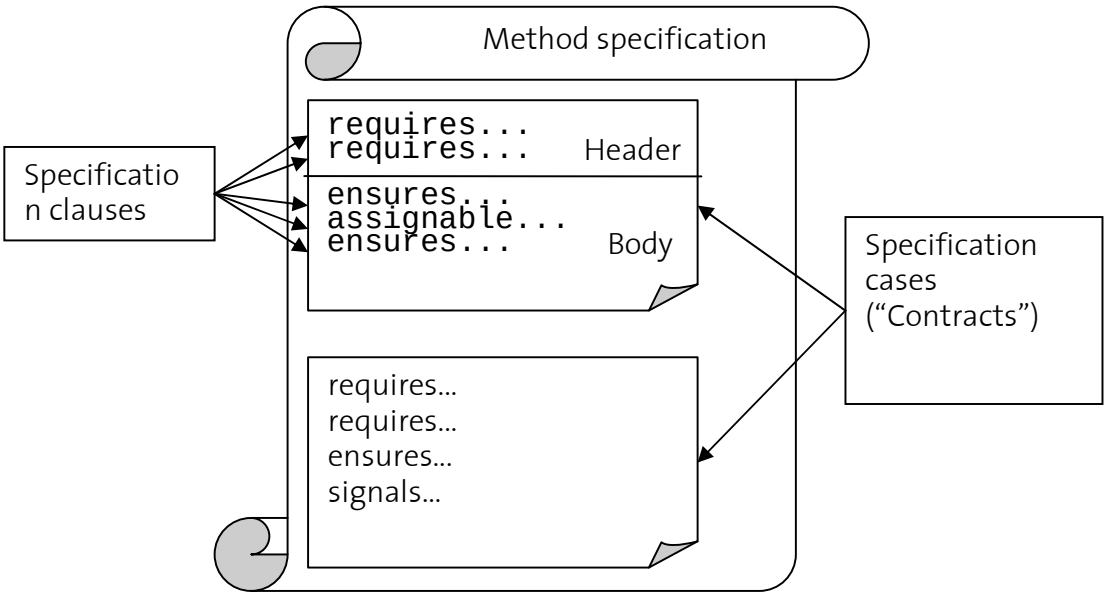
The notion of contract is described in depth in [1]. Basically, it is as if the user and the programmer took the following agreement:

- the user has to make sure the precondition holds before calling the method. He then can be sure that the postconditions hold, whether the method successfully returns or throws an exception.
- The programmer of the method can assume that the preconditions hold, and is then required to make sure the postconditions will hold at the end of the call both in case of failure and of success.

A more detailed insight in JML can be found in the article written by their designers [2].

2. JML Specification: grammar

In this section, we get familiar with the concepts of JML according to its grammar (see paragraph 19.6, [5]), and give the underlying semantics. We give these concepts bottom-up; they are summed up on the following picture.



A sequence of specification clauses builds a specification case. Specification cases are independent from one another, and their sequence builds a method specification. It is not our goal in this part to repeat the JML documentation (the reader can find it in [2], part 9), we rather wish to focus on how we interpret the grammar to implement the proof obligation generator.

2.1. Specification clauses

We first consider the atomic JML elements: specification clauses. Each of them gives a particular contribution in building a specification case.

2.1.1. Header

In the header, specification clauses indicate preconditions. Such a clause begins with the keyword `requires`.

If there is no such clause, and as described in the JML documentation, nothing is promised to the implementor: the precondition is *true*. This means that whatever the context in which the method is called, the postconditions should hold after the call.

2.1.2. Body

Body specification clauses specify the behavior of the method *provided that* the user respected the preconditions in the header.

The behavior of a method can be described with the following clauses:

- `assignable` clauses, along with the only variables the method can assign to. If there are several assignable clauses, we group all of the variables. We will however not strictly respect the JML documentation, in that we actually check whether the method modifies the variables or not, i.e. whether the value of the variables are the same in the prestate and in the poststate. It remains possible for the program to assign to these variables and restore their prestate values, without being detected by Jive (this is a different semantics than that of JML)
- `ensures` clauses, along with the postconditions. If there is no such clause, as described in the JML documentation, the postcondition is true, i.e. no particular condition has to hold after the execution of the method.
- `signals` clauses, followed by a type of exception and an exceptional postcondition the poststate has to satisfy, should an exception of this type or of a subtype be thrown. If there is no such clause, there is no particular constraint when an exception is thrown.
- `diverges`, `when`, `working-space-clause`, `duration`, which we do not support.

For each keyword in the body, clauses are in conjunction: if several of the same kind appear, we will group them with the boolean operator AND.

2.1.3. Nested specification cases

Sometimes, several specification share common preconditions. In this case, it is possible to nest them.
In this case, the body is not a sequence of clauses, but a collection of nested specification cases:

```
header
{|
        Specification 1;
also
        Specification 2;
         ...
|}
```

Where specification1, 2... also contain a header and a body (possibly empty). Nested specifications should be of the same sort that the external one: if it is heavyweight and normal behavior, all of them are implicitely heavyweight and normal behavior as well.

Higher level of nesting is allowed.

If desugared, as explained in [3], the above specification is equivalent to:

```
        header
        Specification 1;
also
        header
        Specification 2;
         ...
```

2.2. Specification cases

A specification case is a contract between the caller and the implementer (see introduction and [1]). We can reformulate our definition of the contract in terms of JML: If the caller makes the preconditions (requires) true, he can then assume that the postconditions (ensures, signals, assignable) are true after the call. The implementer can assume the precondition, has to make the postcondition true, in case of normal termination (ensures) but also when an exception is thrown (signals), and has to ensure that no location which should not be assigned has been assigned.

Any specification case consists of a header (possibly empty) and a body (possibly empty).
Now, there are three types of specification cases:

- Exceptional specification case: there can be no ensures clause since an exception has to be thrown as soon as the precondition holds.
- Normal specification case: there can be no signal cause since no exception can be thrown when the precondition holds.
- Generic specification case: no particular constraint, the method can terminate normally or throw an exception.

The nature of a specification case is specified by the behavior in which it is enclosed, as explained in the next subsection.

2.3. Heavyweight Vs lightweight specification cases

There are four ways to present specification cases.

- Heavyweight: a behavior specification which is a complete specification case of any kind (normal, exceptional, generic). It is put in a sort of “shell” with extra information about this kind and the privacy.
- Lightweight (incomplete): a generic specification case, possibly incomplete.
- Model program which we do not handle.
- Code contracts which we do not handle.

We now focus on behavior specifications and lightweight specification cases. Then we explain how they can all be “desugared” to a heavyweight behavior specification and how this will affect our work.

2.3.1. Heavyweight specification cases

The easiest specification cases to handle are heavyweight specification cases. For them indeed, the JML convention (see [2]) completely defines the behavior of the method.

There are three sorts of heavyweight specification cases:

- Behavior, which is actually a generic specification case introduced by the keyword `behavior`.
- Normal behavior, which is a normal specification case introduced by the keyword `normal_behavior`
- Exceptional behavior, which is an exceptional specification case introduced by the keyword `exceptional_behavior`.

A heavyweight specification case begins with a privacy keyword which gives information about whom the specification is intended for.

This privacy statement is followed by one of the following keywords: `behavior`, `normal_behavior`, `exceptional_behavior`, depending on the type of heavyweight specification case, and eventually by the specification case header and body.

2.3.2. Lightweight

In a lightweight specification case, there is neither a privacy statement nor a behavioral keyword. It is merely a generic specification case.

Since it is not a behavior specification, we neither know whether the method throws any exception, nor how to interpret missing statements. JML considers them as not specified and let implementations adapt this to their needs, although it recommends some possible interpretations. We explain in the following paragraph which one we took.

2.3.3. Desugaring lightweight

or generic specification cases

As far as we are concerned and as described in [4], we will always consider the absence of any keyword `requires, ensures, signals` clause to mean that the associated condition is `true(`whatever the exception for the `signals` clause). This implies that if there is no requires clause, the implementation has to deal with loose preconditions to satisfy the postconditions, and that on the other hand if there is no ensures or signal clause, the degree of freedom in the behavior is high and the caller cannot assume anything on the result.

As specified in [1], the privacy of a lightweight specification is always that of the method.

2.4. Method specification

There can of course be several different contracts, such that a method specification can consist of several specification cases.

Basically, a method specification is composed, in this order, of:

- a sequence of specification cases separated with `also` (heavyweight or lightweight, described in the last paragraphs)
- `implies_that` followed by a sequence of specification cases (which should be logically implied by their predecessors)
- `for_example` followed by a sequence of example specification cases (which should also be implied by their predecessors)

where any part can be omitted. We actually do not support implies_that and for_example.

In case of overriding, the specification extends this of the overridden method and has to be prefixed with `also`. This means that the specification of the overridden method still has to hold and that the current one extends it. Given that the logic of Jive handles these verifications, the generated proof will not contain this super-specification.

2.5. Other JML Features

We also support the following features, which we actually "desugar" and convert to former clauses:

- `pure`: a pure method is a method which cannot modify any variable. This is equivalent to an `assignable \nothing` clause.
- `non_null`: for a parameter x it is equivalent to `requires x != null` and for a method result, to `ensures \result != null`

2.6. JML keywords

JML formulas may contain following keywords:

- \old(expression): in a postcondition, this means that we refer to the value of the expression in the prestate.
- \result: in a postcondition, this is to involve the return value of the method.
- \forall, \exists: in any JML formula, allow universal or existential quantification
- helper: a method modifier which means that, as a helper, it may break invariants of the class.

3. JML 5.0 Classes: Implementation of the specification

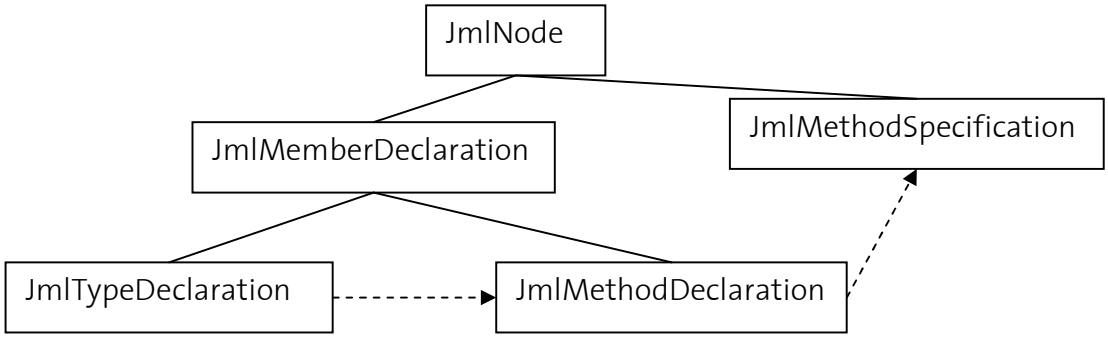
In this section, we explain how the grammar above is implemented in the JML source code and introduce the interface used in the proof obligation generator. The reader interested in the source code should then be able to understand it without getting the JML source code. It could also help a programmer willing to make use of the JML output.

Elements of the abstract syntax tree of the specifications built by the JML parser are in the package org.jmlspecs.checker. It actually completes the AST's interface of Multijava (which extends the syntax of Java).

3.1. The (internal) output of the JML Checker

We start with a hash map containing the set of all parsed classes (types), the key of which is a String with the corresponding name.

Hence, an iteration on the keys gives access to each of the types that the Proof Obligation generator has to process. Each such type is an instance of JmlTypeDeclaration.



Interface of JmlTypeDeclaration
ArrayList methods();
JmlDataGroupeMemberMap getDataGroupMap();

For each type, an iteration on the list of the methods returned by method() gives access to each of them, as instances of JmlMethodDeclaration.

Interface of JmlMethodDeclaration
String ident();
int modifiers();
boolean isPublic();
boolean isPrivate();
boolean isConstructor();
boolean isHelper();
boolean hasSpecification()
JFormalParameters[] parameters();

JmlMethodSpecification methodSpecification();

The modifiers of the method are handled as follows :
Utils.hasFlag(method.modifiers(), ACC_PROPERTY) returns true if :

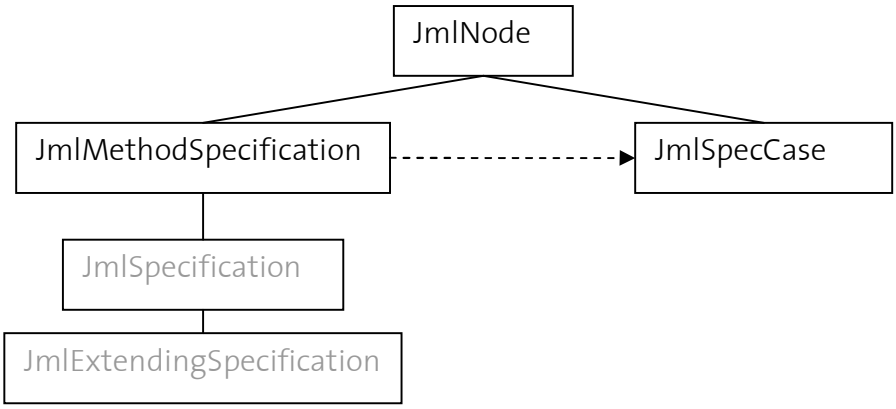
- the method is public when PROPERTY=PUBLIC
- the method is private when PROPERTY=PRIVATE
- the method is protected when PROPERTY=PROTECTED
- the method is pure when PROPERTY=PURE
- the method returns a non-null value when PROPERTY=NON_NULL

and false otherwise.

Interface of JmlFormalParameters (which extends JFormalParameters)
boolean isNonNull();
String ident();
CType getType();

To know whether parameters should be non-null, we call parameters(), cast each element of the array to JmlFormalParameter and call isNonNull().
For each method, besides its properties, we then access its specification.

3.2. Method specification

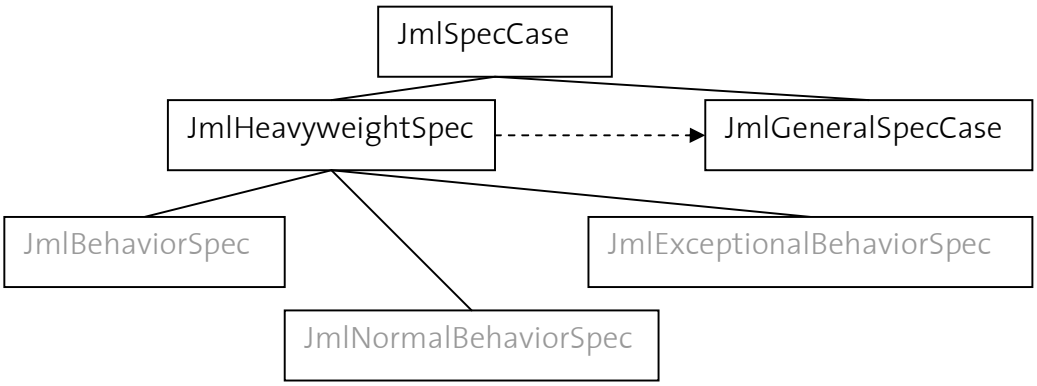


Interface of JmlMethodSpecification
boolean hasSpecCases();
JmlSpecCase[] specCases();

Then, after a test, we can access each specification case of the method.

3.3. Heavyweight and lightweight specification cases

A cast test on each element in the array of the method specification determines whether we have a heavyweight or a lightweight specification case.
If it is heavyweight, it is an instance of JmlHeavyweightSpec, and more particularly one of the three following subclasses:



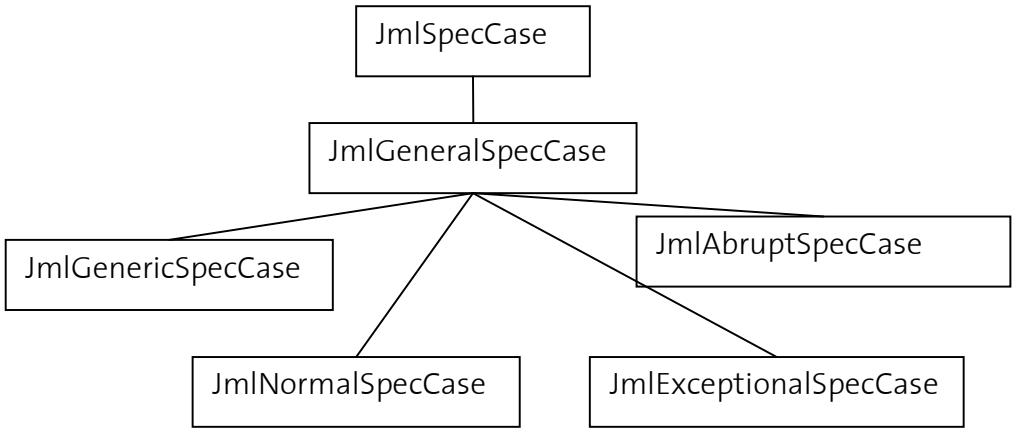
Interface of JmlHeavyweightSpec:
int privacy();
JmlGeneralSpecCase specCase();

The method specCase() then removes the shell and returns the associated specification case (general in the name is not to be mistaken with generic, which is a particular case of specification case).

If the cast fails, this element is already a general specification case and a simple cast is enough to access it.

3.4. General specification cases

Without the shell, lightweight or heavyweight specification cases are all instances of JmlGeneralSpecCase, and in particularly of one of four subclasses:



We can then work with a variable, the static type of which is JmlGeneralSpecCase, since they all provide the same interface.

Interface of JmlGeneralSpecCase
boolean hasSpecHeader()
boolean hasSpecBody()

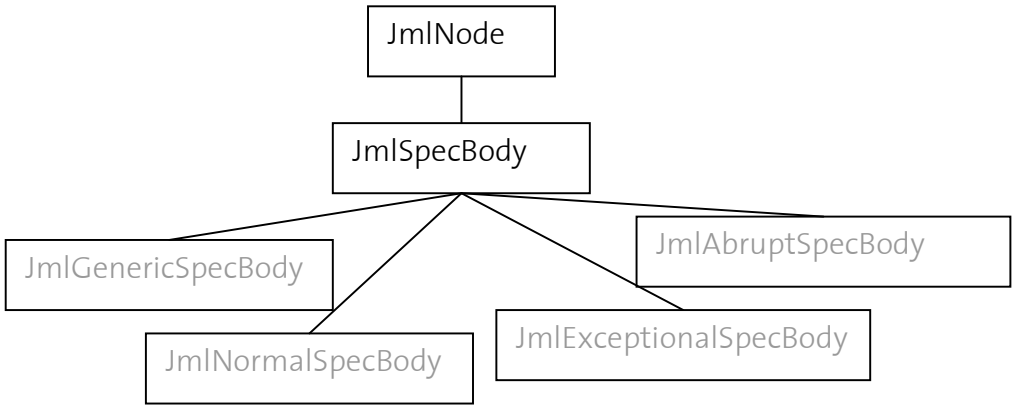
JmlRequiresClause[] specHeader()
JmlSpecBody specBody()

When necessary, a simple instanceof test determines if we have a normal or exceptional behavior.

3.5. Specification clauses, nested specification cases

3.5.1. Body

If the header is already an array of require clauses, accessing the body is somehow more intricate. JmlSpecBody gives access either to the body clauses, or to nested specification cases which all share the header.

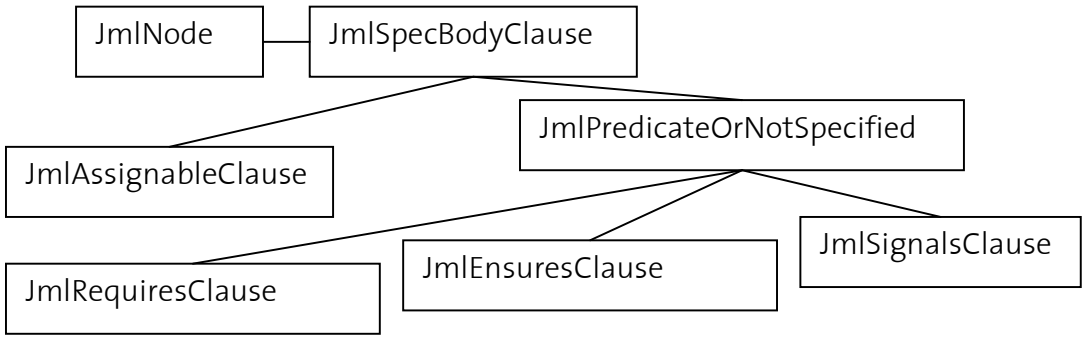


Interface of JmlSpecBody
boolean isSpecClauses()
JmlSpecBodyClause[] specClauses()
boolean isSpecCases()
JmlGeneralSpecCase[] specCases()

In any case, the interface allows us to determine in which case we are, and get either the body clauses or the nested specification cases. In the latter case, we already know the interface of them (see former paragraph).

3.5.2. Clauses (header and body)

All body clauses (assignable, signals, ensures) are instances of JmlSpecBodyClause. Curiously, and likely because of the similarity in implementation, requires clauses also are instances of this class.



3.5.3. Assignable clauses – assignable field set

With the assignable clause, we can access the set of all locations declared assignable.

Interface of JmlAssignableClause
JmlAssignableFieldSet getAssignableFieldSet();

This set provides additional facilities, and handles special cases like universal set or empty set. An iterator allows accessing its elements, which are instances of class JmlSourceField.

Interface of JmlAssignableFieldSet
JmlAssignableFieldSet();
boolean addAll(JmlAssignableFieldSet();
boolean isUniversalSet();
boolean isEmpty();
Iterator iterator();

A datagroup map is associated with each type. It gives information, when a given variable is allowed to be modified, about which other variables may be modified as well (the members of its so-called datagroup).

Interface of JmlDataGroupMemberMap
Iterator keyGroupIterator();
JmlAssignableFieldSet getMembers(JmlSourceField);

3.5.4. Other clauses (predicates)

Interface of JmlPredicateOrNotSpecified
boolean isNotSpecified();
JmlPredicate predOrNot();

For each requires, ensures or signals clause, we can get the associated predicate (after testing whether it is specified). Without entering in further details, invoking

the series of methods predOrNot().specExpression().expression() on such a clause gives the expression.

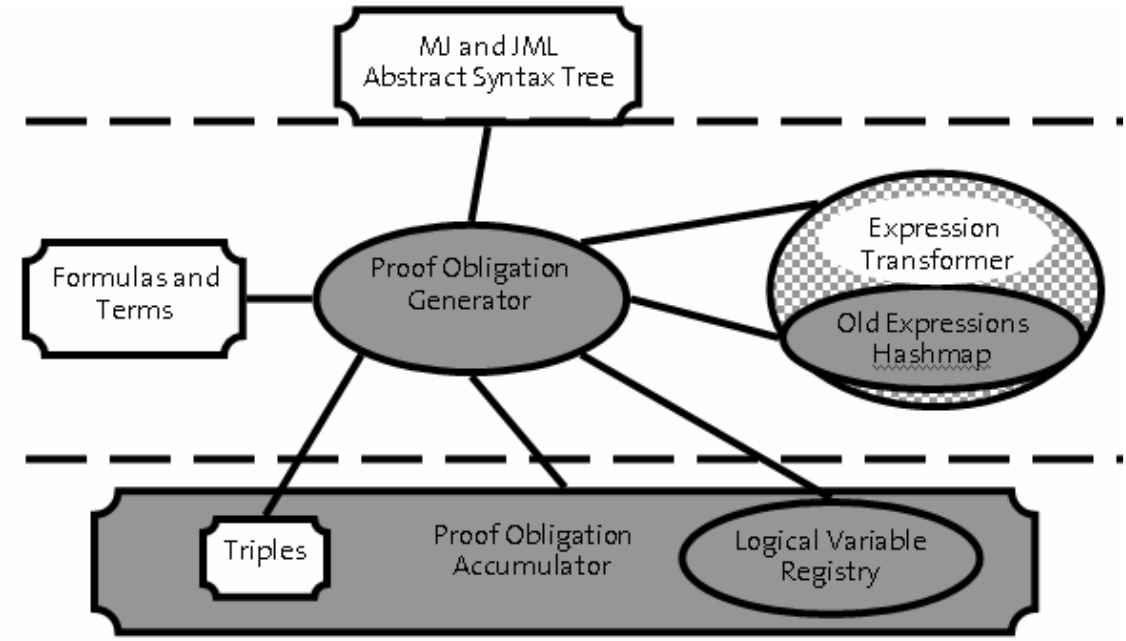
For signals clauses, we still need some more information about the type of exception handled in the condition, and the bound variable representing the exception in the condition.

Interface of JmlSignalsClause
String ident();
CType() type();

III.Proof obligation generator

In this part, we introduce the interface that the proof obligation generator uses to create proof obligations, then explain how the proof obligation generator has been implemented.

1. Proof generations Interface



The Proof Obligation Generator is interfaced with:

- The AST, in the JML format (which also relies on MultiJava), which is the input.
- The formulas and terms it has to generate as output (Interface generated by Katja)
- The expression transformer, which is another module. Its aim is to convert JML formulas to Katja formulas. We slightly modified it so that it supports access to \old expressions (see paragraph 2.5.5.4)
- The Triple interface, which implements a Hoare triple
- The Logical Variable Registry, which contains logical variables created by the Proof Obligation Generator. We implemented it.
- The Proof Obligation Accumulator, which is the assistant of the Proof Obligation Generator and helps it create Proof Obligations. We implemented it as well.

1.1. Formulas and terms

Classes for generating formulas and terms were automatically generated by Katja.
Terms are general logical expression associated with types (Boolean, object, location, store, ...).

Preconditions and postconditions in Hoare triples are all formulas (a statement that can be true or false) and formulas can be built with terms just like in mathematics.

A detailed interface of Formulas and Terms is given in the annex.

1.2. Triples

A triple is nothing but a triplet composed of:
- a precondition (of type Formula)
- a reference to a method (of type MethodRef)
- a postcondition (of type Formula)

It is the implementation of a contract.
We create it straight-forward with the constructor:
Triple(Formula, MethodRef, Formula)

1.3. Logical variable registries

Together with a triple, we need to include a table containing all of the logical variables created in order to:

- refer to /old references in the postcondition
- save the store
- refer to the assignable field set
- refer to free variables

We store each of these tables as an instance of LogicalVarRegistry, which behaves like a hashmap associating a String key (the name of the variable) to a (katja) Type (merely its type, expressed the Katja way).

The constructor is the default constructor LogicalVarRegistry() and the method put(String, Type) adds new elements.

1.4. Expression transformer

In this paper, we focus about the structure of JML behavioral specifications and how it is mapped into Hoare Triples. Translating expressions found in pre- and postconditions constitutes a project in itself and is performed in another class, ExpressionTransformer.
This class basically implements the σ function mentioned in [4, section 3]. The corresponding method takes a JML expression (which is always an instance of a certain class JExpression) and returns a formula.

Actually, the expression transformer's interface contains two methods transformTerm and transformFormula, the both of which take an instance of JExpression as argument and return a term or a formula, depending on the call. If the expression corresponds to a boolean, calling transformFormula "wraps" the result in a formula using the IsTrue class, as explained in 1. This simplifies the code in the caller.

To complement the expression transformer and allow substituting in formulas, we also have a method jive.ContainerInterface.Util.substitute() which takes as argument a formula (already converted to first-order logic), the variable to replace and the expression to replace this variable for.

1.5. Expression transformer old hash map

It is possible that a JML expression in a postcondition contains a special keyword \old. It means that we want to access, from the poststate, a value in the prestate.
In this case, translating the expression to a Formula is not enough! We need to have access to the value of the expression in brackets BEFORE the call. To make this process easier, the expression transformer has been modified such that, when an \old expression is met:

- it is replaced with a new logical variable.
- this logical variable is added to a hashmap together with the abstract syntax tree of the expression in brackets and its type.

An interface then provides access to this information after calling the expression transformer:

```
public static void emptyOldHashSet();
```
empties the hash set before an expression transformer call.
```
public static boolean hasOldHashSet();
```
tests whether the hast set contains an element after the call.
```
public static Iterator getIteratorOnOldHashSet();
```
gets an iterator to obtain these elements, if any.

And:

```
public static void setInPostCond();
public static void setInPreCond();
```

which tell the expression transformer whether one is in a pre- or postcond (this is for proper handling of parameters, see paragraph 2.5.5.5)

Precise explanations about \old expression are given later in this paper.

1.6. Proof obligation Accumulator

The class ProofObligationAccumulator is designed so as to assist the class ProofObligationGenerator in building proof obligations. It stores all of the generated proof obligations, and provides a transaction-like interface to generate a new one and store it as well.

Only once in the generation process, an instance of this class - which will eventually contain all of the obligations - is created by the Proof Obligation Generator with the default constructor. It is then empty.

1.6.1. Life cycle of the Proof Obligation Accumulator

A new triple for a given method is created by newPO(MethodRef). The argument is a reference to the current method.

Then, it can be modified by:

- adding a new precondition with appendToPreCond(Formula), which conjuncts the formula with the existing precondition (or creates it if there was none). The new precondition is then (former precondition) AND (new formula).
- adding a new postcondition with appendToPostCond(Formula), which conjuncts the formula with the existing postcondition (or creates it if there was none). The new postcondition is then (former postcondition) AND (new formula).
- Registering a new variable in the logical variable registry, with registerVarInSymbolTable(String, Type), which returns the corresponding logical variable of type LogicalVar.
- Setting/unsetting flags about assignable locations, with setAssignableEverything(), setAssignableNothing(). These flags are not stored, but allow communication between parts of the generator handling assignable clauses.

Eventually, it is commited with commit().

1.6.2. Utility methods

Besides, utility methods help further during the Proof Obligations generation:

- getTypeOf(String) returns the type of an already registered logical variable for the current Proof Obligation.
- appendToCond(Formula, Formula) (which is static!) computes the conjunction of two formulas (semantically: adds the second one to the first one) and returns it.
- isEverythingAssignable(), isNothingAssignable() gives the values of the flags previously set (or not) for the current Proof Obligation.

1.6.3. Handling nested specifications

Nested specifications, in a way, “break” the normal life cycle of the generation of proof obligations.

In the normal life cycle, the proof obligation generator creates a new proof obligation p, reads a specification, modifies p, and eventually commits p. Then it can generate another proof obligation and so on.

However, imagine that, after creating a new proof obligation p, after reading the header of a specification and after modifying p accordingly, it does not find a body, but nested specification cases. It cannot simply read the first one, complete p and

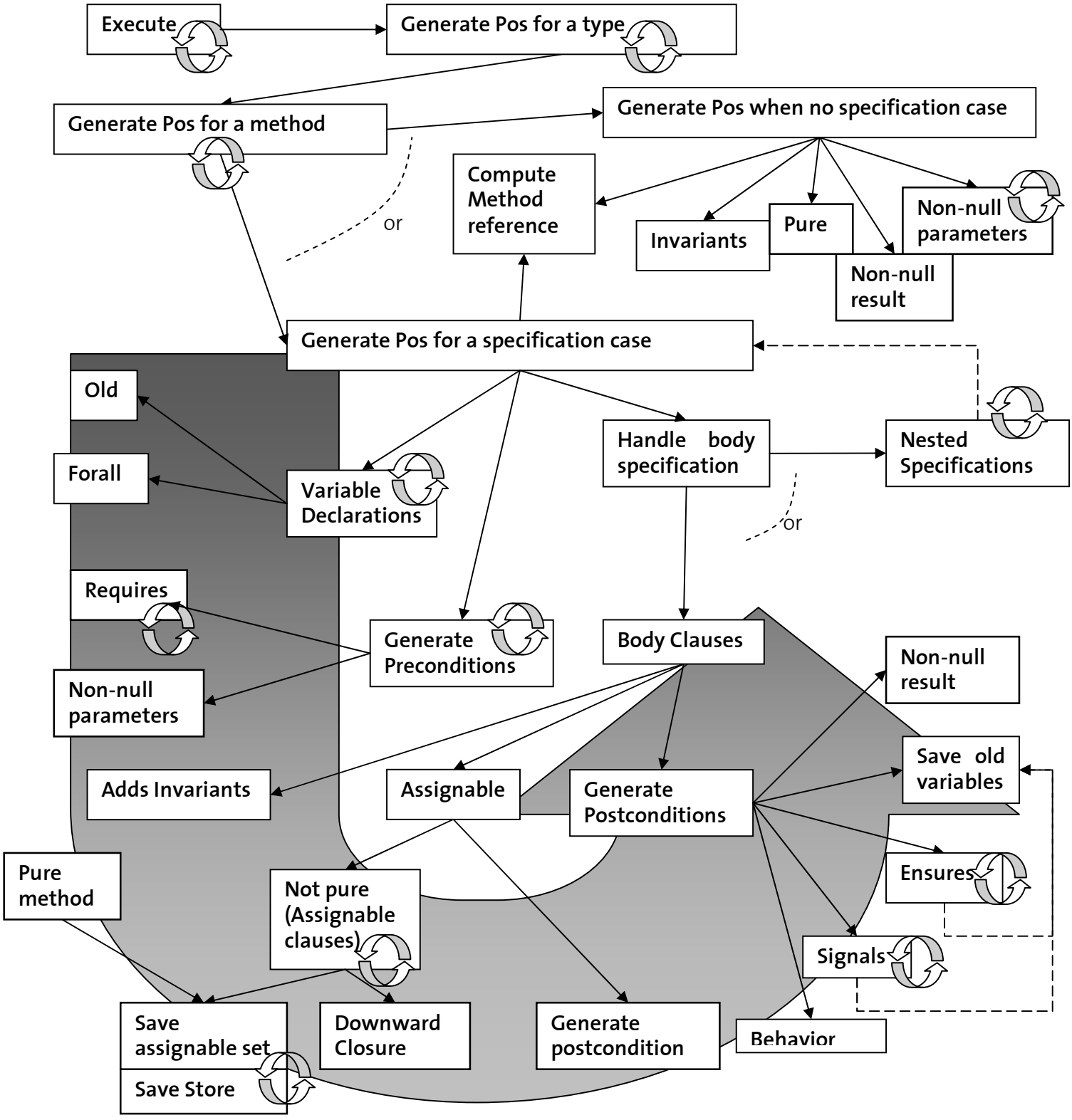
commit it! When processing further nested specification cases, everything in the common header would be lost...

The Proof Obligation Accumulator provides tools to solve this problem: a stack allows one to push an incomplete proof obligation at any time (pushPO()), like a snapshot of the situation, and to pop it (popPO()), when one is done with it.

To allow this functionality, the semantics of newPO() has to modified in the following way: if the stack is empty, the behavior does not change; if the stack is not empty, the call to newPO() has a different effect: the newly generated PO is not empty, but is a copy (conditions and declared logical variables) of the PO on top of the stack. It can then be completed, and normally commited.

2. ProofObligationGenerator

2.1. Overview - Architecture



The graphics sums up how the different parts of the Proof Obligation Generator interact with one another, and it can be used as support for the rest of this part, in which we explain each of these functionalities in details.

2.2. Methods in the Proof Obligation Generator

In the next parts, we describe the implementation and how all of these methods work together. The intent of this overview is not to simply list the source code - the version here, given as a support to the implementation issues, is actually very simplified, comments and display commands were removed - and the reader wanting to have a deeper insight might want to read the actual code.

If you wish to make use of the proof obligation generator, you do not need to know these details and you might rather want to refer to paragraph 3.

The names of all of the methods in this architecture have been chosen so that reading the code gives a semantical - rather than technical - overview, just like you would read a book. The "technical" code is left to helper methods.

2.3. External loop

There is only one instance (singleton) of the class ProofObligationGenerator, which you get by calling the static method getSingleton(). The forest of abstract syntax trees generated by the JML checker is then passed to the method setTopLevelTypes() of this instance. The initiation of the process is nothing but a series of iterations, from types to specifications.

- The execute() method performs an iteration on this forest, calling generatePOsForType() for each of the trees - one for each type. This is the only public method (together with method setTopLevelTypes()) in the interface of the ProofObligationGenerator instance. It returns the Proof Obligation Accumulator containing all proofs.

- generatePOsForType() then calls generatePOsForMethod() for each method defined in the type.

- generatePOsForMethod() checks whether the method contains a specification or not, and calls either generatePOForNoSpecCase() or iterates over the specification cases and calls for each of them generatePOForSpecCase(). It eventually checks purity and generates a unique and additional proof obligation if the method is pure.

Not that if a method is pure, we assume it never modifies a location, i.e. it is associated with a "require true" precondition (possibly conjoined to the invariant in the triple), whatever the preconditions of the specification. This is a discussed issue because there exists an opposite alternative (reusing preconditions from the specification).

Furthermore, should one want to specify that no location is to be modified under certain conditions, this is handled by \assignable nothing.

```
Private ProofObligationAccumulator POs;

ProofObligationAccumulator execute() {
  POs = new ProofObligationAccumulator();
  ExpressionTransformer.setJMLMode(true);
    Iterator iter = topLevelTypes.keySet().iterator();
    while (iter.hasNext()) {
      String st = (String) iter.next();
      generatePOsForType((JmlTypeDeclaration) topLevelTypes.get(st));
    }
  return POs;
}

private void generatePOsForType(JmlTypeDeclaration type) {
  Iterator iter = type.methods().iterator();
  while (iter.hasNext())
    generatePOsForMethod(type, (JmlMethodDeclaration) iter.next());
}

private void generatePOsForMethod(JmlTypeDeclaration type,
                                  JmlMethodDeclaration method) {
  if (method.hasSpecification()
              && method.methodSpecification().hasSpecCases()) {
    JmlSpecCase[] specs = method.methodSpecification().specCases();
    for (int i = 0; i < specs.length; i++)
           generatePOForSpecCase(type, method, specs[i]);
  }
  else
    generatePOForNoSpecCase(type, method);
  generatePOForPureMethod(type, method);
```

2.4. No specification

If the method has no specification, i.e. generatePOsForNoSpecCase() was called, we:

- generate the invariants (by calling generatePrePostInvariant(), which analyses whether to add them or not)
- generate preconditions for non-null results or parameters (which the method could have...)

```
private void generatePOForNoSpecCase(JmlTypeDeclaration type,
                                     JmlMethodDeclaration method) {
  POs.newPO(MethodRef.getMethodRef(type, method));
    generatePrePostCondInvariant(method);
    generatePreCondNonNullParameters(type, method);
    generatePostCondNonNullResult(method);
  POs.commit();
}
```

2.5. Specification

If the method has a specification, we handle each specification case.

If it is not supported (model program, code contract), a runtime exception is thrown. Else, whatever the case, we extract the general spec case with header and body.

One first computes a reference to the method, and handle variable declarations followed by standard preconditions (requires, ...). Then depending on the type of body (whether it is nested, or consists of clauses), we call the corresponding method.

In the upcoming paragraphs, we describe all of these parts, except variable declarations and nested specification cases, which are explained later.

```
private void generatePOForSpecCase(JmlTypeDeclaration type,
            JmlMethodDeclaration method, JmlSpecCase specCase) {
  if (specCase instanceof JmlModelProgram)
    throw new RuntimeException("Model Programs not supported!");
  else if (specCase instanceof JmlCodeContract)
    throw new RuntimeException("Code Contract not supported!");

  JmlGeneralSpecCase generalSpecCase;
  int privacy;
  if (specCase instanceof JmlGeneralSpecCase) {
    privacy = method.modifiers();
    generalSpecCase = (JmlGenericSpecCase) specCase;
  } else {
    privacy = ((JmlHeavyweightSpec) specCase).privacy();
    generalSpecCase = ((JmlHeavyweightSpec) specCase).specCase();
  }

  POs.newPO(getMethodRef(type, method, privacy));
  generatePreCondSpecVarDecls(generalSpecCase);
  generatePreConditions(type, method, generalSpecCase);

  if (generalSpecCase.hasSpecBody()
                && generalSpecCase.specBody().isSpecCases())
    generatePOsForNestedSpecCases(type, method, generalSpecCase);
  else
    generatePOsForBodyClauses(type, method, generalSpecCase);
}
```

2.5.1. Method reference

Calling a special method (MethodRef.getMethodRef()) with type and method as argument gives a (unique) reference to the current method.

The reference is bound either dynamically (type:method()) if the method is public or protected, or statically (type@method()) if it is private or static.

However, if the privacy of a specification cannot be more visible than this of the method, it is possible that, for instance, the method be public (hence dynamically bound) but the specification private! In this case, we need a method reference which is statically bound and have to generate it manually. To this aim, we replace

getMethodRef() by our version, in the same class. If the method is abstract and the specification private, we throw an exception.
Note that if the specification case is lightweight, the privacy of it is considered the same as that of the method.

```
private MethodRef getMethodRef(JmlTypeDeclaration type,
                JmlMethodDeclaration method, int privacy) {
  MethodRef result = null;
  MethodRef mr = (MethodRef.getMethodRef(type, method));
  if (Utils.hasFlag(privacy, ACC_PRIVATE))
    if(mr instanceof ConcreteVirtualMethod)
      result = Main.currentSession().getUnique(new Implementation(type, method));
    else if (mr instanceof AbstractVirtualMethod)
      throw new RuntimeException("Private specification for an abstract method");
  return result == null ? MethodRef.getMethodRef(type, method) : result;
}
```

2.5.2. Headers

In all cases, we process the header of the specification case, if it exists. We handle requires clauses, and methods with non-null parameters (if nested, which is tackled later in this paper, we do not do it again since it was already done at level o of nesting).

First of all, we need to tell the expression transformer that we are in a precondition (because of handling of parameters - for more explanations, see 2.5.5.5)

```
private void generatePreConditions(JmlTypeDeclaration type,
            JmlMethodDeclaration method, JmlGeneralSpecCase specCase) {
  ExpressionTransformer.setInPreCond();
  if(specCase.hasSpecHeader())
    generatePreCondRequires(specCase);
  if(!POs.isNested())
    generatePreCondNonNullParameters(type, method);
}
```

2.5.2.1 Requires clauses

To handle requires clauses, we merely iterate on them, convert expressions with the expression transformer and append the result to the already generated preconditions.

As explained in [4], in case of multiple preconditions
requires P_1;
requires P_2;
...
requires P_n;

we conjoin them. Hence, the generated precondition is:
$\sigma(P_1) \wedge \sigma(P_2) \wedge ... \wedge \sigma(P_n)$

where σ is the conversion to a first-order logic formula performed by the Expression Transformer.

This precondition will itself be conjoined to the existing precondition of the current Hoare Triple in the Proof Obligation Accumulator.

```
private void generatePreCondRequires(JmlGeneralSpecCase specCase) {
  JmlRequiresClause[] preconds = specCase.specHeader();
  for (int k = 0; k < preconds.length; k++) {
    if (preconds[k].isNotSpecified())
      continue;
    JExpression requires = (JExpression) preconds[k].predOrNot()
                                          .specExpression().expression();
    Formula tRequires = ExpressionTransformer
                                             .transformFormula(requires);
    POs.appendToPreCond(tRequires);
  }
}
```

2.5.2.2 Non-null parameters

We simply iterate on the arguments of the method, and add a precondition if it is flagged as non-null.

If we find a non-null argument, say:

public void method(**/*@ non_null @*/** *type* a) ;

Then we conjoin the following precondition as explained in [4], paragraph 4.4:

$a \neq null$

in which a is passed as a ProgVar instance. For more explanations about differences between program variables and logical variables, you could refer to paragraph 2.8.

```
private void generatePreCondNonNullParameters(JmlTypeDeclaration type,
                JmlMethodDeclaration method) {
  JFormalParameter[] parameters = method.parameters();
  for (int i = 0; i < parameters.length; i++) {
    if (((JmlFormalParameter) parameters[i]).isNonNull()) {
      Type t = CompRef.getKatjaType(parameters[i].getType());
      POs.appendToPreCond(new IsTrue(new BinOpTermTerm(
                          new ProgVar(parameters[i].ident(), t),
                          JNotEq.INSTANCE, new Null(t), JBoolean.INSTANCE)));
    }
  }
}
```

2.5.3. Body clauses

When there is no nesting, i.e. the body is simply a sequence of clauses, we process in the following order to analyze clauses:

- We first add invariants (see paragraph 2.6)

- Then we handle assignable locations (whether pure method or assignable clause) (paragraph 2.5.4)
- And we finally generate standard postconditions (requires, signals, non-null result, ...) (paragraph 2.5.5)

```
private void generatePOsForBodyClauses(JmlTypeDeclaration type,
  JmlMethodDeclaration method, JmlGeneralSpecCase specCase) {
  generatePrePostCondInvariant(method);
  generatePrePostCondAssignmentsForSpecCase(type, method, specCase);
  generatePostCondForSpecCase(type, method, specCase);
  POs.commit();
}
```

2.5.4. Assignable locations

Processing of assignable locations depends on the purity of the method. If it is pure, this is handled by a separate Proof Obligation and we leave this step for later. If it is not the case, we refer to assignable expressions:

- we generate a precondition building the set M of assignable locations (if it is not trivial), which we conjoin to the precondition of the current Hoare triple.
- and then generate the postcondition which makes sure that locations in this set are not modified.

Note that the specification of JML indicates that the semantics of an assignable location is that it should never be assigned anything, even if one restores the former value before the method returns. However, as explained in [4], paragraph 4.2, Jive will only check that the location was not modified, i.e. that the location has the same content before and after the call.

```
private void generatePrePostCondAssignmentsForSpecCase(
  JmlTypeDeclaration type, JmlMethodDeclaration method,
  JmlGeneralSpecCase specCase) {

  if (!Utils.hasFlag(method.modifiers(), ACC_PURE)) {
    generatePreCondAssignableNotPure(type, method, specCase);
    generatePostCondAssignable();
  }
}
```

2.5.4.1 Not pure methods

If the method is not pure, we look at the assignable clauses the specification contains. We collect all of them, and compute the downward closure (as defined and explained later in this paragraph). If there is none, as said in [4] paragraph 4.2, we consider that all locations are assignable.

Note that all such locations are instance fields of the class, since JML does not support dynamic dependencies yet (it produces an error if a field of another object is passed).

We then set the flags in the Proof Obligation Accumulator if no location or all locations are assignable, and save the list of them in a precondition otherwise. This optimization, if a flag is set, enables us to skip the precondition and build a simpler postcondition.

```
private void generatePreCondAssignableNotPure(JmlTypeDeclaration type,
            JmlMethodDeclaration method, JmlGeneralSpecCase specCase) {
  boolean foundAssignableLocations = false;
  JmlAssignableFieldSet assignable = new JmlAssignableFieldSet();
  if (specCase.hasSpecBody() && specCase.specBody().isSpecClauses()) {
    JmlSpecBodyClause[] specBodyClauses = specCase.specBody().specClauses();
    for (int i = 0; i < specBodyClauses.length; i++) {
      if (!(specBodyClauses[i] instanceof JmlAssignableClause))
        continue;
      foundAssignableLocations = true;
      assignable.addAll(((JmlAssignableClause) specBodyClauses[i])
               .getAssignableFieldSet());
    }
  }
  if (foundAssignableLocations) {
    if (assignable.isUniversalSet()) {
      POs.setAssignableEverything();
      return;
    } else if (assignable.isEmpty()) {
      POs.setAssignableNothing();
      return;
    } else
      generateDownwardClosure(type, method, assignable);
  } else {
    POs.setAssignableEverything();
    return;
  }
  if (!POs.isEverythingAssignable() && !POs.isNothingAssignable())
    generatePreCondAssignable(termListFromAssignableFieldSet(type, assignable));
}
```

In JML, the user can make use of so-called model fields, which are additional fields that are not part of the implementation, but make the specification easier and solve such problems like:

- assigning new locations when subclassing and overriding, i.e. fields of the subclass
- Keeping private fields… private and expressing specifications on a more abstract level.

A model field m is associated to a data group, the locations in which are also implicitely assignable, should m appear in an \assignable clause. Hence, when subclassing, new fields are added to an existing data group and are hence assignable by corresponding methods.
Private fields are added in data groups of the corresponding model fields. If the implementation changes, one does not need to change model fields (which appear in the interface), but just data groups.

This is transitive, i.e. if x is in y's data group and y is in z's data group, then if z appears in a \assignable clause, y but also indirectly x are assignable.

To compute the downward closure, we first get the hashmap from the type, which contains all of the necessary information about data groups for this type. Then, we add all elements of all data groups of locations in our bag and repeat the operation until no more remains to add.

```
private void generateDownwardClosure(JmlTypeDeclaration type,
  JmlMethodDeclaration method, JmlAssignableFieldSet assignable) {
  JmlDataGroupMemberMap dataGroupMap = type.getDataGroupMap();
  JmlAssignableFieldSet toAdd = null;
  while (toAdd == null) {
    toAdd = new JmlAssignableFieldSet();
    for (Iterator i = assignable.iterator(); i.hasNext();) {
      JmlSourceField field = (JmlSourceField) i.next();
      JmlSourceField correspondingField = null;
      Iterator j = dataGroupMap.keyGroupIterator();
      while (correspondingField == null && j.hasNext()) {
        JmlSourceField candidateField = (JmlSourceField) j.next();
        if (field.toString().equals(candidateField.toString()))
          correspondingField = candidateField;
      }
      if(correspondingField)
        toAdd.addAll((JmlAssignableFieldSet)
                                dataGroupMap.getMembers(correspondingField));
      else
         `toAdd.add(field);
    }
    if (toAdd.size() != assignable.size()) {
      assignable.addAll(toAdd);
      toAdd = null;
    }
  }
}
```

After collecting and computing the downward closure, we still need to convert our set to first-order logic to generate the precondition.

Since dynamic dependencies are not yet supported but JML, we know that all locations corresponding to fields in the instance, i.e. a name n will actually refer to this.n.

This is what we do in the following method.

```
private TermList termListFromAssignableFieldSet(JmlTypeDeclaration type,
    JmlAssignableFieldSet assignable) {
  TermList assignableTermList = null;
  for (Iterator i = assignable.iterator(); i.hasNext();) {
    JmlSourceField field = (JmlSourceField) i.next();
    Type this_type =
      ExpressionTransformer.getJiveType(type.getCClass().getType());
     Term location =
       new FuncAppl(
         PredefinedTerms.LOC,
         new TermList(
           new This(this_type)
         ).appBack(
           new Field(
             field.getIdent(),
             (ClassOrInterfaceType) this_type,
             ExpressionTransformer.getJiveType(field.getType())
         )
       ),
       PredefinedTerms.location_T
     );

    if (assignableTermList == null)
      assignableTermList = new TermList(location);
    else
      assignableTermList = assignableTermList.appBack(location);
  }
  return assignableTermList;
}
```

2.5.4.2 Saving assignable locations

Once we have the TermList of all assignable locations, we can save it in a precondition. The following method has been called above if the set M is not trivial. As always with LogicalVars, we register it in the Symbol Table before we build the precondition and conjoin it to the precondition of the current Hoare triple.

```
private void generatePreCondAssignable(TermList assignableTermList) {
  SetTerm assignableLocations = new SetTerm(assignableTermList, SetType.INSTANCE);
  LogicalVar M = POs.registerVarInSymbolTable("M", SetType.INSTANCE);
  POs.appendToPreCond(new IsTrue(new BinOpTermTerm(M, JEq.INSTANCE,
                  assignableLocations, JBoolean.INSTANCE)));
}
```

2.5.4.3 Checking they were not modified

The last step is to conjoin a postcondition which makes sure no location was modified, except the one mentioned (directly or indirectly) in the specification.

- If we have a non-trivial set (no flag set in the Proof Obligation Accumulator), we conjoin:

$$\forall loc : Location.(loc \notin M \wedge alive(obj(loc), S) \Rightarrow \$(loc) = S(loc))$$

- If nothing is assignable (corresponding flag set in the Proof Obligation Accumulator), we conjoin:

$$\forall loc : Location.(alive(obj(loc), S) \Rightarrow \$(loc) = S(loc))$$

- If everything is assignable (corresponding flag set in the Proof Obligation Accumulator), we do not need to check anything.

```
private void generatePostCondAssignable() {
  LogicalVar M = new LogicalVar("M", SetType.INSTANCE);
  LogicalVar S = new LogicalVar("S", Store.INSTANCE);
  LogicalVar loc = new LogicalVar("loc", locationType);

  Term loc_dollar =
    new FuncAppl(
      PredefinedTerms.LOOKUP,
      new TermList(new Dollar(Store.INSTANCE)).appBack(loc),
      PredefinedTerms.value_T
    );

  Term loc_S =
    new FuncAppl(
      PredefinedTerms.LOOKUP,
      new TermList(S).appBack(loc),
      PredefinedTerms.value_T
    );

  if (POs.isNothingAssignable()) {
    generatePreCondSaveStore();
    POs.appendToPostCond(
      new BindingFormula(Forall.INSTANCE, new BindingList(new Binding(loc)),
        new BinaryFormula(
          new IsTrue(
            new FuncAppl(
              PredefinedTerms.ALIVE,
              new TermList(
                new FuncAppl(PredefinedTerms.obj, new TermList(loc),
                                          PredefinedTerms.objectType)
              ).appBack(PredefinedTerms.DOLLAR),
              JBoolean.INSTANCE
            )
          ),
          FImplies.INSTANCE,
          PredefinedTerms.eq(loc_dollar,loc_S)
        )
      )
    );
  } else if (!POs.isEverythingAssignable()) {
    generatePreCondSaveStore();
    POs.appendToPostCond(
      new BindingFormula(Forall.INSTANCE, new BindingList(new Binding(loc)),
        new BinaryFormula(
          new BinaryFormula(
            new FNot(
              new IsTrue(
                new BinOpTermTerm(loc,Element.INSTANCE,M,JBoolean.INSTANCE)
              )
            ),
            FAnd.INSTANCE,
            new IsTrue(
              new FuncAppl(
                PredefinedTerms.ALIVE,
                new TermList(
                  new FuncAppl(PredefinedTerms.obj,
                               new TermList(loc),PredefinedTerms.objectType)
                ).appBack(PredefinedTerms.DOLLAR),
                JBoolean.INSTANCE
              )
            )
          ),
          FImplies.INSTANCE,
          PredefinedTerms.eq(loc_dollar,loc_S)
        )
      )
    );
  }
}
```

generatePreCondSaveStore() simply generates a logical var S in which the store is saved.

```
private void generatePreCondSaveStore() {
  display("Saving store", 3);
  LogicalVar S = POs.registerVarInSymbolTable("S", Store.INSTANCE);
  POs.appendToPreCond(new IsTrue(new BinOpTermTerm(
    new Dollar(Store.INSTANCE),
    JEq.INSTANCE,
    S, JBoolean.INSTANCE)));
}
```

2.5.4.4Pure methods

If the method is pure, no location should be assigned. The following method is called after processing each method, and checks purity:

```
private void generatePOForPureMethod(JmlTypeDeclaration type,
            JmlMethodDeclaration method) {
  if (Utils.hasFlag(method.modifiers(), ACC_PURE)) {
    POs.newPO(MethodRef.getMethodRef(type, method));
      generatePreCondAssignablePure(type, method);
      generatePostCondAssignable();
    POs.commit();
  }
}
```

If the method is pure, we call the following method:

```
private void generatePreCondAssignablePure(JmlTypeDeclaration type,
  JmlMethodDeclaration method) {
  if (method.isConstructor()) {
    JFieldDeclarationType[] jfd = type.fields();
    TermList assignableTermList = null;
    for (int i = 0; i < jfd.length; i++) {
      JClassFieldExpression generatedThisField = new JClassFieldExpression(
        null, new JThisExpression(null,type.getCClass()), jfd[i].ident());
      if (assignableTermList == null)
        assignableTermList = new TermList(ExpressionTransformer
                             .transformLocation(generatedThisField));
      else
        assignableTermList = assignableTermList
          .appBack(ExpressionTransformer.transformLocation(generatedThisField));
    }
    generatePreCondAssignable(assignableTermList);
  } else
    POs.setAssignableNothing();
}
```

We simply set the corresponding flag in the Proof Obligation Accumulator so that the adequate postcondition be then generated (see last line of code).

There is however an exception to this rule: if one has a constructor, it should be allowed that it assigns any field in the instance.

If such is the case, we iterate on the fields of the class, which we all put in our bag (a TermList) and generate the precondition saving the set just like non-pure methods.
In both cases, we then generate postconditions as for non-pure methods as well.

2.5.5. Postconditions

Postconditions are found in ensures clauses and signal clauses. We also handle non-null results and \old expressions, as well as normal and exceptional behaviors.

This is done in the following order:

- We determine normal or exceptional behavior if such is the case.
- We process normal postconditions (\ensures clauses)
- We process exceptional postconditions (\signals clauses)
- Should Logical Variables (for accessing prestate values in the poststate) have been used in the last two steps, they are in the "old hast set". We register them and save the prestate value of the corresponding expression thanks to a special precondition.
- We test for non-null result

First of all, we need to empty the "old hash set" and to tell the expression transformer that we are in a postcondition (for proper handling of parameters – see 2.5.5.5)

```
private void generatePostCondForSpecCase(JmlTypeDeclaration type,
  JmlMethodDeclaration method, JmlGeneralSpecCase specCase) {

  ExpressionTransformer.emptyOldHashSet();
  ExpressionTransformer.setInPostCond();
  generatePostCondBehavior(specCase);
  generatePostCondEnsures(specCase.specBody().specClauses());
  generatePostCondSignals(specCase.specBody().specClauses());
  generatePreCondSaveOld();

  generatePostCondNonNullResult(method);
}
```

2.5.5.1 Normal or exceptional behavior

In case of normal or exceptional behavior, one appends a postcondition meaning that the method has to return a value, or that the method has to throw an exception.

- In case of a normal behavior, no exception may be thrown. As specified in [4] paragraph 4.2, we conjoin the following formula to the postcondition of the current Triple in the Proof Obligation Accumulator:

$$\chi \neq \text{Exc}$$

In case of a normal behavior, the method cannot return without throwing an exception. As specified in [4] paragraph 4.2, we conjoin the following formula to the postcondition of the current Triple in the Proof Obligation Accumulator:

$$\chi \neq \text{Normal}$$

```
private void generatePostCondBehavior(JmlGeneralSpecCase specCase) {
  if (specCase instanceof JmlNormalSpecCase)
    POs.appendToPostCond(new IsTrue(new BinOpTermTerm(
      new Chi(Status.INSTANCE),
      JNotEq.INSTANCE,
      new Exc(Status.INSTANCE), JBoolean.INSTANCE)));
            return;
  else if (specCase instanceof JmlExceptionalSpecCase)
    POs.appendToPostCond(new IsTrue(new BinOpTermTerm(
      new Chi(Status.INSTANCE),
      JNotEq.INSTANCE,
      new Normal(Status.INSTANCE), JBoolean.INSTANCE)));
}
```

2.5.5.2 Normal postconditions

As explained in [4], in case of multiple normal postconditions
ensures Q_1;
ensures Q_2;
...
ensures Q_n;

we conjoin them. Hence, the generated normal postcondition is:
$\sigma(Q_1) \wedge \sigma(Q_2) \wedge ... \wedge \sigma(Q_n)$ where σ is the conversion to a first-order logic formula performed by the Expression Transformer.

Since it is a normal postcondition, the following formula is conjoined to the postcondition of the current triple:

$$\chi = Normal \Rightarrow \sigma(Q_1) \wedge \sigma(Q_2) \wedge ... \wedge \sigma(Q_n)$$

If the specification is a normal behavior, an (implemented) optimization removes the implication since the left hand side always holds.

```
private void generatePostCondEnsures(JmlSpecBodyClause[] specBodyClauses) {
  Formula normal = null;
  for (int l = 0; l < specBodyClauses.length; l++)
    if (specBodyClauses[l] instanceof JmlEnsuresClause) {
      JExpression jExp = ((JmlPredicateOrNotSpecified) specBodyClauses[l])
                         .predOrNot().specExpression().expression();
      Formula t = ExpressionTransformer.transformFormula(jExp);
      normal = ProofObligationHashMap.appendToCond(normal, t);
    }

  if (normal != null)
    if(specCase instanceof JmlNormalSpecCase)
      POs.appendToPostCond(normal);
    else
      POs.appendToPostCond(new BinaryFormula(new IsTrue(
        new BinOpTermTerm(
          new Chi(Status.INSTANCE),
          JEq.INSTANCE,
          New Normal(Status.INSTANCE),
          JBoolean.INSTANCE)),
        FImplies.INSTANCE,
        normal));
}
```

2.5.5.3 Exceptional postconditions

Processing signals clauses is slightly more intricate: before building a conjunction of all of them, one has to take the type and name of each exception mentioned in a clause.
With the following clause:
signals (E e) R;

The generated exceptional postcondition is:
$typeof(ExcV) \preceq E \Rightarrow \sigma(R[ExcV / e])$
where σ is the conversion to a first-order logic formula performed by the Expression Transformer.

This means that, if the exception thrown is a direct or indirect instance of E, the condition R holds, where e (the bound variable) designs the actual value of the exception thrown.

As explained in [4], in case of multiple exceptional postconditions
signals (E_1 e_1) R_1;
signals (E_2 e_2) R_2;
...
signals (E_n e_n) R_n;

we conjoin them.

And as we did for normal postconditions, we add a prefix expressing that it is an exceptional postcondition, so that the following formula is conjoined to the postcondition of the current triple:

$$\chi = Exc \Rightarrow typeof(ExcV) \preceq E_1 \Rightarrow \sigma(R_1)[ExcV / e_1]$$
$$\wedge\, typeof(ExcV) \preceq E_2 \Rightarrow \sigma(R_2)[ExcV / e_2]$$
$$\wedge\, \ldots$$
$$\wedge\, typeof(ExcV) \preceq E_n \Rightarrow \sigma(R_n)[ExcV / e_n]$$

During the substitution, a field exceptionBound contains the name of the variable being substituted, so that it be not considered a program var and erroneously "saved" (cf 2.8): The method isLogical() also compares its argument with this field.

Finally, if the specification is an exceptional behavior, an optimization removes the implication since the left hand side always holds.

```
private void generatePostCondSignals(JmlSpecBodyClause[] specBodyClauses) {
  Formula exc = null;
  for (int l = 0; l < specBodyClauses.length; l++)
    if (specBodyClauses[l] instanceof JmlSignalsClause) {
      JExpression jExp = ((JmlPredicateOrNotSpecified) specBodyClauses[l])
                         .predOrNot().specExpression().expression();
      CType exceptionType = ((JmlSignalsClause) specBodyClauses[l]).type();
      String exceptionIdent = ((JmlSignalsClause) specBodyClauses[l]).ident();
      exceptionBound = exceptionIdent;
      Formula t = new BinaryFormula(
        new IsTrue(new BinOpTermType(
          new ExcV(objectType),
          InstanceOf.INSTANCE,
          CompRef.getKatjaType(exceptionType),JBoolean.INSTANCE)),
        FImplies.INSTANCE,
        jive.ContainerInterface.Util.substitute(
          ExpressionTransformer.transformFormula(jExp),
          new ExcV(PredefinedTerms.objectType),
          new LogicalVar(
             exceptionIdent, ExpressionTransformer.getJiveType(exceptionType)
        )
      );
      exceptionBound = null;
      exc = ProofObligationHashMap.appendToCond(exc, t);
    }

if (exc != null)
  if(specCase instanceof JmlExceptionalSpecCase)
    POs.appendToPostCond(exc);
  else
    POs.appendToPostCond(new BinaryFormula(
      new IsTrue(
        new BinOpTermTerm(
          new Chi(Status.INSTANCE),
          JEq.INSTANCE,
          new Exc(Status.INSTANCE), JBoolean.INSTANCE)),
```

2.5.5.4 Accessing prestate values in the poststate

Ensures and signals clauses might contain \old expressions or formal parameters, which both should refer to values before the call. To that aim, we had to slightly modify the expression transformer.

In that case, the expression transformer transforms the corresponding expression to first order logic, but does not put it back in the returned tree. Instead, it creates a new logical variable associated to the converted expression (and its type) in the so called “old hast set”. It is thus this new logical variable which occurs in the returned tree.

For instance, if we have:

ensures $expr_1$ op_1 \old(expr)

a new logical variable (with an auto generated unique name), say !xo, is generated, the expression transformer fills the old hast set:

Name	Type	Tree
!xo	(the type of expr)	σ(expr)

And the returned first-order formula is actually (we only give a feeling):

$\sigma(expr_1)$ $\sigma(op1)$!xo

Hence, one has to complete the work of the expression transformer by generating special preconditions which saves prestate values in those logical variables.

This is done by a special method generatePreCondSaveOld() it checks whether the expression transformer found expressions referring to the prestate, and processes them.

```
private void generatePreCondSaveOld() {
  if (ExpressionTransformer.hasOldHashSet()) {
    Iterator oldsIterator = ExpressionTransformer.getOldIterator();
    while (oldsIterator.hasNext()) {
      ExpressionTransformerOldHashSet.LogicalVarEntry l =
          (ExpressionTransformerOldHashSet.LogicalVarEntry) oldsIterator.next();
      generatePreCondSaveOld(l.getName(), l.getAST());
    }
  }
}
```

The saving precondition is conjoined by the following method. In our example, it conjoins:

!xo=σ(expr)

to the precondition of the current triple.

```
private void generatePreCondSaveOld(String name, Term ast) {
  display("Saving old variable "+name);
  LogicalVar F = POs.registerVarInSymbolTable(name, ast.type());
  POs.appendToPreCond(new IsTrue(new BinOpTermTerm(F, JEq.INSTANCE, ast,
    JBoolean.INSTANCE)));
}
```

2.5.5.5 Formal parameters in postconditions

According to the JML specification, formal parameters in postconditions still implicitely refer to their value in the prestate! Modifications of their values are implementation details and are irrelevant in specifications.

Hence, a new flag in the Expression Transformer tells whether we are in a precondition or in a postcondition. It is set by the Proof Obligation Generator accordingly. When the expression transformer finds a formal parameter
\ensures ... param ...
it treats it as if it were in a \old expression:
\ensures ... \old(param) ...

2.5.5.6 Non-null result

Postconditions are found in ensures and signals clauses, but also in the keyword non-null in front of the method declaration. The latter can actually be desugared to:

ensures \result!=o

which is handled by a dedicated method generatePostCondNonNullResult().

```
private void generatePostCondNonNullResult(JmlMethodDeclaration method) {
  if (Utils.hasFlag(method.modifiers(), ACC_NON_NULL))
    POs.appendToPostCond(new IsTrue(new BinOpTermTerm(
      new ResV(objectType),
      JNotEq.INSTANCE,
      new Null(NullType.INSTANCE), JBoolean.INSTANCE)));
  }
}
```

2.6. Invariant generation

The invariant generation is always called, with or without specification. This is performed by another module parallel to the Proof Obligation generation and we do not tackle the invariant generation in itself.

However, we need to add a special formula INV($) to certain preconditions and postconditions. The method INV() contains the invariants for all of the classes in the program (which have to hold in any visible state, even the invariant of a class A after calling a method in another class B). It takes the current store as argument, which is $. You will find further explanations in [4].

As explained in the latter article, we only add invariants in the precondition if the method is neither a helper, nor a constructor, and we only add them in the postcondition if the method is not a helper.

```
private void generatePrePostCondInvariant(JmlMethodDeclaration method) {
        if (method.isConstructor()) {
            POs.appendToPostCond(invdollar);
        } else if (!method.isHelper()) {
            POs.appendToPreCond(invdollar);
            POs.appendToPostCond(invdollar);
        }
    }
```

2.7. Variable declaration

It is possible in JML to declare variables. There are two types of them:

- old variable
- forall variables

The following method, called just before processing preconditions, iterates over them and dispatches handling to two methods generatePreCondSaveOld() and generatePreCondForAll().

```
private void generatePreCondSpecVarDecls(JmlGeneralSpecCase generalSpecCase) {
  JmlSpecVarDecl[] vars = generalSpecCase.specVarDecls();
  if (vars != null)
    for (int i = 0; i < vars.length; i++) {
      if (vars[i] instanceof JmlLetVarDecl) {
        JmlLetVarDecl jlvd = (JmlLetVarDecl) vars[i];
        JVariableDefinition[] jvd = jlvd.specVariableDeclarators();
        for (int j = 0; j < jvd.length; j++) {
          Term ast = ExpressionTransformer.transformTerm(jvd[i].expr());
          String name = jvd[i].ident();
          generatePreCondSaveOld(name, ast);
        }
      }
      if (vars[i] instanceof JmlForAllVarDecl) {
        JmlForAllVarDecl jfavd = (JmlForAllVarDecl) vars[i];
        JVariableDefinition[] jvd = jfavd.quantifiedVarDecls();
        for (int j = 0; j < jvd.length; j++) {
          String name = jvd[i].ident();
          Type typ = CompRef.getKatjaType(jvd[i].getType());
          generatePreCondForAll(name, typ);
      }
    }
  }
}
```

2.7.1. Old declared variables

In postconditions, the user can use \old() expressions to access prestate values. The Proof Obligation Generator then automatically generates a logical variable to store this value.

It is possible that the user decides to choose the name of this variable by himself and declares it in the specification:

/*@ public normal_behavior
  @ **old int var = expr;**
  @ requires ...
  @ ensures ...var...
  @*/

This will allow him, for instance, to write shorter or clearer code.
In which case any occurrence of var in a postcondition is equivalent to \old(expr). Hence, we use the very same method to conjoin the precondition, but neither need to choose a new name, nor to replace it with the variable (it is already there!).

```
private void generatePreCondSaveOld(String name, Term ast) {
  LogicalVar F = POs.registerVarInSymbolTable(name, ast.type());
  POs.appendToPreCond(new IsTrue(new BinOpTermTerm(
    F,
    JEq.INSTANCE,
    ast, JBoolean.INSTANCE)));
}
```

2.7.2. Forall declared variables

Forall declared variables are straightforward to implement, and that is probably the reason why it is not so simple.

Forall declared variables are not documented a lot in JML, and we assume they have the following meaning.

With the following code.

```
/*@ public normal_behavior
  @ forall int var;
  @ requires ...var...
  @ ensures ...var...
  @*/
```

The contract described by the specification contains a bound variable, and is equivalent to a set of contracts C_i:

```
/*@ public normal_behavior
  @ requires ...i...
  @ ensures ...i ...
  @*/
```

all of which hold between the programmer and the client. Or, more intuitively, that the contract C_i holds whatever the value of i you may choose.

In terms of desugaring, we could (only theoretically!) desugar

```
/*@ public normal_behavior
  @ forall int var;
  @ requires ...var...
  @ ensures ...var...
  @*/
```

to

```
/*@ public normal_behavior
  @ requires ...MIN_INT...
  @ ensures ...MIN_INT ...
  @ also
```

```
@ public normal_behavior
@ requires ...MIN_INT+1...
@ ensures ...MIN_INT+1 ...
@ also
@ ...
@ also
@ public normal_behavior
@ requires ...0...
@ ensures ...0...
@ also
@ public normal_behavior
@ requires ...1...
@ ensures ...1...
@ also
@ ...
@ also
@ public normal_behavior
@ requires ...MAX_INT...
@ ensures ...MAX_INT...
@*/
```

In terms of theorem proving, we have to prove

For all i:
if P_i holds before the method call, then Q_i holds after the method call.

Let us make the following assumption on the theorem prover:
if our theorem prover receives as input a formula to prove which contains a free variable,
F(i)
Then it will consider it true if and only if it holds for all i, i.e. it can make no assumption on i!

For example,
$i=j \Rightarrow i+1=j+1$
is considered true, because logically equivalent to:
$\forall i, j \in N, (i=j \Rightarrow i+1=j+1)$

Note: (for readers knowing the theorem prover currently used, Isabelle): In Isabelle, free variables correspond to arbitrary values, introduced by the sign $\wedge$, and the above assumption is actually rule allI (cf [7], page 91).

Hence, if we only register this declared variable and do not bind it, the theorem prover cannot make any assumption about it and will act as if there were an implicit “forall var” in front of the triple.

```
private void generatePreCondForAll(String name, Type type) {
  LogicalVar F = POs.registerVarInSymbolTable(name, type);
}
```

If this assumption should not be respected by the theorem prover, we can just iterate on logical variables in the symbol table and add manually these universal quantifications to the Hoare triple, just before the formula be proven.

2.8. Logical variables and program variables

2.8.1. Variable handling in JML Vs in first-order logic

During the transformation process, we meet impedance mismatches between JML and first-order logic, because of different handling of the variables.

In JML, variables can be:
- any declared variable
- formal parameters
- fields

In our first-order logic, logical variables refer to:
- the store S (generated by the POG)
- the set of assignable locations M (generated by the POG)
- variables used to save prestate values that we want to access in a postcondition and generated by the POG
- any declared variable (old or forall) (variable already existing in JML)

whereas program variables refer to:
- formal parameters (already existing in JML)

In the version of first-order logic used in Jive, Fields are always considered being "looked up" at corresponding locations in the heap, so that they are not treated as variables in formulas.

2.8.2. The impedance mismatch and how it is solved

If the JML parser differentiates fields from other variables, it unfortunately does not differentiate in its tree:
- declared variables
- from formal parameters

Both are local variable expressions!

Hence, one needs to help the expression transformer to differentiate between then and decide whether we have:
- a formal parameter (transformed to a program variable)
- a JML declared variable (transformed to a logical variable)

To that aim, the Proof Obligation Generator provides a static method isLogical(), which the Expression Transformer can use when in doubt. This functions actually looks up in the symbol table of the proof obligation being built, also checks the exception name if we are building an exceptional postcondition, and checks variables used in (forall ...) expressions (currently not added in the symbol table).

Furthermore, we said formerly that the Expression Transformer has to detect formal parameters. Since we have no local variables to methods here, it is straightforward that any "Local Variable Expression" that is not considered logical by the Proof Obligation Generator is a formal parameter and can be treated as explained in 2.5.5.5

2.8.3. What about name colliding?

Obviously, it is not advisable to the user to use a forall or old variable with the same name as fields, but if this should happen (for example, in case of private fields one does not know), Jive handles it the following way.

If a variable in a specification is bound to a forall or old declaration, it "hides" any field which would have the same name, which is the intuitive behavior one expects and actually, already the behavior of JML. To access such a field, one would have to use this.name so that there is no lack of functionality.

Let us mention that any colliding within a specification is not allowed by JML (it fails with a message), so that no renaming is needed.

2.9. Nested specifications

If we do not have clauses, but nested specification, we use the stack functionality of the Proof Obligations Accumulator. All new proof obligations in the future will actually start with the one we pushed on the stack, until we pop it.
Since we have a stack, this allows any level of nesting. We use a variable offset for other methods to know whether we are in a nested specification or in the root one.

```
private void generatePOsForNestedSpecCases(JmlTypeDeclaration type,
  JmlMethodDeclaration method, JmlGeneralSpecCase specCase) {
  POs.pushPO();
  offset++;
  for (int i = 0; i < specCase.specBody().specCases().length; i++)
    generatePOForSpecCase(type, method, specCase.specBody().specCases()[i]);
  offset--;
  POs.popPO();
}
```

3. Output interface

If you would like to retrieve the triples and logical var tables from the proof obligation generator without knowing the implementation details, this subsection is for you.

3.1. Passing types to the Proof Obligation Generator

There is a unique instance of the Proof Obligation Generator, which you can get with the static method getSingleton();

Then the static method setTopLevelTypes() allows to pass the abstract syntax tree of the JML-annotated program (which is the output of the JML parser...).

3.2. Launching and getting the Output

The Proof Obligation Generator method execute() returns an instance of ProofObligationAccumulator. This object contains all of the generated proof obligations.

The package jive.ContainerInterface.HoareFrontend contains all necessary tools to make use of it.

3.3. Iterator on Proof Obligations

Calling the method iterator() on the object – instance of ProofObligationAccumulator - we got from the generator gives us an iterator on proof obligations. It has the basics features you can expect from a forward iterator, that is:
- a method next() which returns the next Proof Obligation, which is an instance of class ProofObligation
- a method hasNext() which precises if there is any remaining proof obligation.

3.4. A Proof Obligation

All Proof Obligations are instances of ProofObligation. A proof obligation consists of:

- a triple, i.e. a precondition, a method reference and a postcondition. This is an instance of the class Triple.
- a table with all registered logical variables (the store in the prestate, the set of assignable locations, expressions in the prestate, ...)

3.5. A Triple

These triples are the very class we saw in the input interface, if you have read it. We called the constructor of the triple with a precondition, a method reference and a postcondition.

To get these data back, the interface is straightforward:

- getPre() returns the precondition (a Formula)
- getCompRef() returns the method reference
- getPost returns the postcondition (a Formula)

3.6. Iterators on logical variables

For each Proof Obligation, getting all of the registered logical variables is performed with an iterator with an interface very similar to the iterator on proof obligations:

- calling next() returns the next logical variable
- calling hasNext() precises whether there remains any logical variable to be processed.

This iterator, like that on proof obligations, is obtained with the method logicalVarIterator() of the current proof obligation.

3.7. Code sample

The following code prints out all of the generated proof obligations.

```
//The Proof Obligation Generator is the unique instance of its class
ProofObligationGenerator pog = ProofObligationGenerator.getSingleton();

//We first parse the annotated code with the JML Checker
topLevelTypes = Main.currentSession().topLevelTypes();

//We pass them to the Proof Obligation Generator
pog.setTopLevelTypes(topLevelTypes);

//and execute the generation, we get the accumulator as output.
ProofObligationAccumulator poAccumulator = pog.execute();

//Then we can iterate on Proof Obligation
ProofObligationIterator poIterator = poAccumulator.iterator();
while(poIterator.hasNext())
      ProofObligation po = poIterator.next();

      //We can access pre- and postconditions, and the method reference
      System.out.println(po.getTriple().getPre());
      System.out.println(po.getTriple().getCompRef());
      System.out.println(po.getTriple().getPost());

      //And then iterate on logical variables for this Proof Obligation
      LogicalVarIterator lvi = po.logicalVarIterator();
      while(lvi.hasNext())
            LogicalVar lv = lvi.next();
            System.out.println(lvi);
      }

}
```

Note that actually, in a proof obligation, toString() is overridden and prints everything including logical variables, which allows shorter code inside the main loop in this case.

IV. Tests

The testing part is performed with JUnit. In this part, we give some examples of test code together with what we expect.

1. Basic: pure, non_null, helpers and constructors

So that tests be as orthogonal as possible, we first test sugar keywords and whether invariants are correctly handled, without specifications.

```
package jive.PC.FrontEnd.testfiles;
public class NoSpec {
    private int i, j;

     public /*@pure@*/ NoSpec() {}          Pure constructor
```

What is expected: invariant only in postcondition, with a trivial precondition. Additional proof obligation for purity, with fields as i and j as only assignable locations.

```
     public NoSpec(int a) {}                Constructor
```

What is expected: invariant only in postcondition, trivial precondition.

```
     private /*@helper@*/ NoSpec(long a) {}  Helper constructor
```

What is expected: trivial pre- and postconditions.

```
     public void nothing() {}               Method
```

What is expected: invariants in pre- and postconditions.

```
     private /*@helper@*/ void helper() {}   Helper method
```

What is expected: trivial pre- and postconditions.

```
     public /*@pure@*/ void pure() {}       Pure method
```

What is expected: a PO with invariants in pre- and postcondition, an additional PO for purity (nothing is assignable).

```
     public /*@non_null@*/ Object           Non-null result
          nonNullResult() {
               return new Object();
     }
```

What is expected: invariant in precondition, result is non-null (and invariant) in postcondition.

Non-null parameter

```
public void nonNullParameter(/*@non_null@*/ Object a)
{}
```

What is expected: parameter is non-null (and invariant) in postcondition, invariant in post-condition.

```
}
```

2. Privacy checking

We know check that specification binding occurs as we wish (with trivial specifications)

```
package jive.PC.FrontEnd.testfiles;

public class PrivacyChecking {
     /*@private behavior
      @requires true;
      @ensures true;
      @*/
     private void private_private() {
     }
```

What is expected: specification is statically bound (jive.PC.FrontEnd.testfiles.PrivacyChecking@private_private() as method reference)

```
     /*@public behavior
      @requires true;
      @ensures true;
      @*/
     public void public_public() {
     }
```

What is expected: specification is dynamically bound like the method (jive.PC.FrontEnd.testfiles.PrivacyChecking:public_public() as method reference)

```
     /*@private behavior
      @requires true;
      @ensures true;
      @*/
     public void public_private() {
     }
```

What is expected: specification is NOT dynamically bound like the method, but statically bound (jive.PC.FrontEnd.testfiles.PrivacyChecking@public_private() as

method reference). This is the case when a call to the MethodRef method is not enough and we need to generate the reference by ourselves.

```
}
```

3. Behavior specifications

Now, we check the behavior keywords. Again for orthogonality reasons, we allowed all locations to be assigned (This is the default setting if nothing was said about this, but since they are heavyweight specificaitons, skipping the assignable assertion would generate a warning...)

```
package jive.PC.FrontEnd.testfiles;

public class Behaviors {

    public int i, j;

    /*@
     @ public behavior
     @ requires i!=0;
     @ ensures j>1;
     @ assignable \everything;
     @*/
    public void behavior() {
    }
```

What is expected: invariants in both pre- and postconditions, the precondition i!=0, normal and exceptional postconditions as implications (χ=Normal $\Rightarrow$... ,and also if there had been an exceptional postcondition, χ=Exc$\Rightarrow$...).

```
    /*@
    @ public normal_behavior
    @ requires i!=0;
    @ ensures j>1;
    @ assignable \everything;
    @*/
    public void normal_behavior() {

    }
```

What is expected: invariants in pre- and postconditions, the precondition i!=0, χ=Normal as postcondition, normal postconditions with no implication (χ=Normal $\Rightarrow$... would be useless, this is an optimization).

```
    /*@
    @ public exceptional_behavior
    @ requires true;
    @ signals (Exception e) j>1;
    @ signals (RuntimeException e) j>3;
    @ assignable \everything;
```

```
    @*/
    public void exceptional_behavior() {

    }
```

What is expected: invariants in pre- and postcondition, χ=Exc as postcondition, exceptional postconditions conjoined with with no general implication (χ=Exc $\Rightarrow$... would be useless, this is an optimization) but with type implications (typeof(ExcV) <: RuntimeException $\Rightarrow$ j>3 for example).

```
}
```

4. Old expressions

We know test \old expressions, and use of old variable declarations.

```
package jive.PC.FrontEnd.testfiles;

public class OldExpression {
    public int i;
    public B x;
    /*@requires true;
    @ensures \old(i)==i+1;
    @ensures \old(this.i+2)==i+1;
    @ensures \old(x.f)==i+2*\old(x.f);
   */
  public void old() {
  }
```

What is expected: invariants in pre- and postconditions, three normal postconditions conjoined with χ=Normal implication, logical variables !x0, !x1 and !x2 created and registered, to save I, this.i+2 and x.f in the prestate, same logical variable for the two identical expressions x.f, \old replaced with these logical variables.

```
  /*@old int oldi = i;
    @requires true;
    @ensures oldi==i+1;
   */
  public void oldVarDecl() {}
```

What is expected: invariants in pre- and postcondition, normal postconditions conjoined with χ=Normal implication, a logical variable with user-defined name oldi registered, and a precondition to store i into this logical variable in the prestate.

```
}
```

5. Forall expressions

```
package jive.PC.FrontEnd.testfiles;

public class ForAllExpression {
    public int i;

    /*@public behavior
    @forall int a;
    @requires a!=0;
    @ensures \result!=a;
   */
    public int forall() {
        return 0;
    }
```

What is expected: invariants in pre- and postconditions, normal postcondition conjoined with χ=Normal implication, logical variable a registered (with the same name).

```
    public boolean a;
    /*@public behavior
    @forall int i;
    @requires i!=0;
    @ensures \result!=i;
   */
    public int forallFieldCollide() {
        return 0;
    }
}
```

What is expected: The same as above, and variable I inside the specification interpreted as bound logical variable and NOT as field (this is already done at the JML level).

6. Nested specifications

```
package jive.PC.FrontEnd.testfiles;

public class NestedSpec {



    /*@public behavior
    @old int i = 2;
    @requires i!=0;
    @{|
    @       requires true;
    @       assignable \nothing;
    @       ensures \result!=i;
    @ also
```

```
@       old int j = 1;
@       requires i!=0 && (\forall int l;true;true) &&
                          (\forall int l;true;true);
@       assignable \nothing;
@       ensures \result!=i;
@ |}
*/
public int oneLevel() {
    return 0;
}
```

What is expected: Two proof obligations, one with the common i!=0 precondition with the first nested specification, the second with that same precondition and the second nested specification. Variable i is also declared in the header, and is hence used by the two proof obligations as well.

```
/*@public normal_behavior
@forall int i;
@requires i!=0;
@{|
@       requires true;
@       assignable \nothing;
@       ensures \result!=i;
@ also
@       old int j = 1;
@       requires i!=0;
@       assignable \nothing;
@       ensures \result!=i;
@ |}
*/
public int oneLevelNormal() {
    return 0;
}
```

What is expected: Two proof obligations as above, both normal.

```
/*@public exceptional_behavior
@forall int i;
@requires i!=0;
@{|
@       requires true;
@       assignable \nothing;
@ also
@       old int j = 1;
@       requires i!=0;
@       assignable \nothing;
@ |}
*/
public int oneLevelExceptional() {
    return 0;
}
```

What is expected: Two proof obligations as above, both exceptional.

```
/*@public normal_behavior
@forall int i;
@requires i!=0;
@{|
@       requires true;
@       ensures \result!=i;
@ also
@     {|
@     requires true;
@          ensures \result!=i;
@         assignable \nothing;
@       also
@       {|
@          requires true;
@              ensures \result!=i;
@             assignable \nothing;
@         also
@              forall int k;
@         requires i!=0;
@              ensures \result!=i+i;
@             assignable \nothing;
@       |}
@     |}
@ |}
*/
public int moreLevelsNormal() {
    return 0;
}
```

What is expected: Four proof obligations, with respect to the way they are nested (this is to show that nesting specification is possible and work properly)

}

7. A "real" example

7.1. IntMathOps

We chose to take the following one from [2]:

```
package jive.PC.FrontEnd.testfiles;

public class IntMathOps {
    /*@ public normal_behavior
       requires y >= 0;
       assignable \nothing;
       ensures 0 <= \result
           && \result * \result <= y
           && y < ((\result+1) * (\result+1));
      @*/
    public static int isqrt(int y) {
        return (int) Math.sqrt(y);
    }
}
```

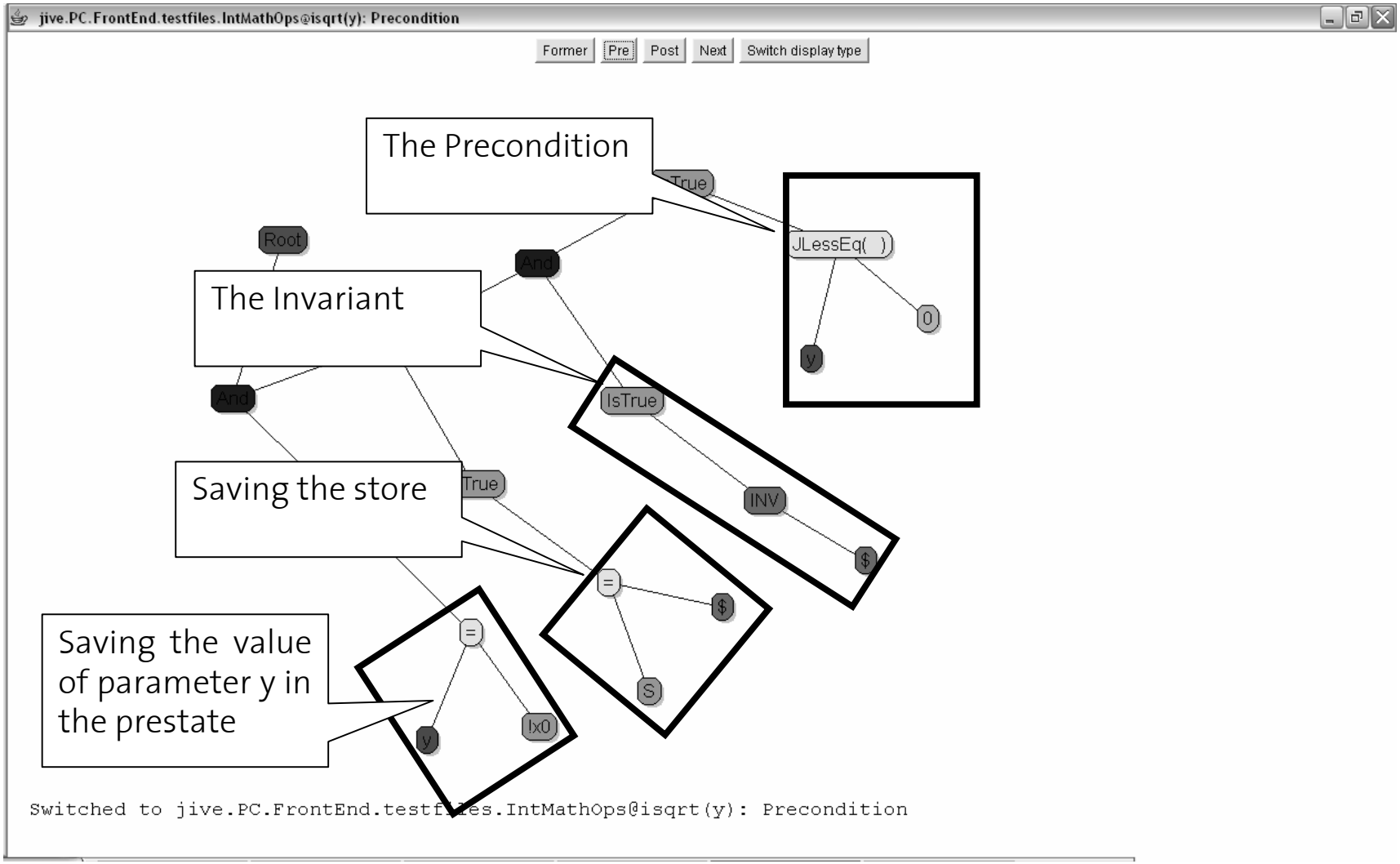
7.2. Visualization

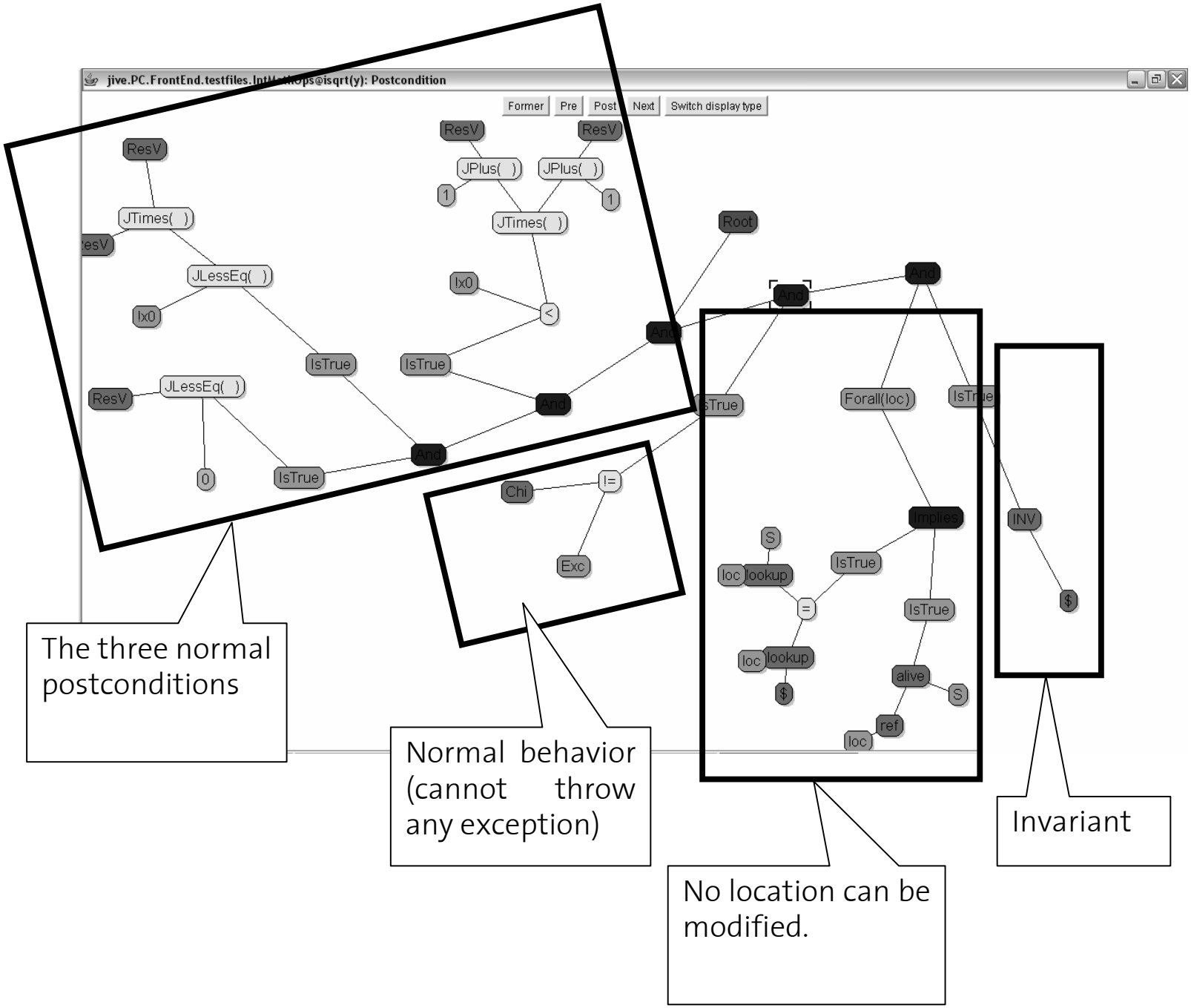
The Proof Obligation Generator comes along with a Proof Obligation Visualizer which offers a graphical view on the tree and the structure of proof obligations.

We present below views of the precondition and postcondition in the Hoare triple.

In this proof obligation, the following logical variables are registered:

- S (the store)
- !xo (the value of y in the prestate)


jive.PC.FrontEnd.testfiles.IntMathOps@isqrt(y): Precondition
Former
Pre
Post
Next
Switch display type
The Precondition
The Invariant
Saving the store
Saving the value of parameter y in the prestate
True
Root
And
JLessEq( )
0
y
And
IsTrue
INV
$
True
=
$
S
=
y
Ix0
Switched to jive.PC.FrontEnd.testfiles.IntMathOps@isqrt(y): Precondition


jive.PC.FrontEnd.testfiles.IntMathOps@isqrt(y): Postcondition
Former
Pre
Post
Next
Switch display type
ResV
JPlus( )
1
JTimes( )
JLessEq( )
lx0
IsTrue
0
Root
And
Chi
!=
Exc
Forall(loc)
Implies
S
loc
lookup
=
alive
ref
INV
$
The three normal postconditions
Normal behavior (cannot throw any exception)
No location can be modified.
Invariant

V. Conclusion

1. What the current implementation covers

The following features are supported:

- appending invariants with respect to constructors and helpers
- adaptative binding with privacy of specifications and methods, i.e. it makes sure that a private specification is bound statically.
- preconditions (requires), even when multiple
- postconditions (normal: ensures, exceptional: signals), even when multiple
- assignable clauses (including saving the store in a logical variable S), handled "the modifies way". An optimization prevents unused generation of S.
- computing the downward closure and "saving" it in a logical variable M
- optimization for \assignable nothing, \assignable everything
- defaults for behavior and lightweight specification when clauses are omitted
- desugaring of normal and exceptional behavior, and optimization of corresponding postconditions (removing the implication)
- pure methods with additional proof obligation
- non-null parameters and results
- \old expressions, \forall and \exists expressions (for the latter, without registration in the logical variable table of proof obligations). They appear in pre- and/or postconditions (\old only in postconditions).
- Old and forall variable declarations (unlike the above ones, they appear before a JML specification header)
- Nested specifications (any level)
- Handling of special keywords, even when no specification is given
- Everything which is supported by the expression transformer that does not require further assistance of the Proof Obligation Generator

2. Further work

There is still an issue when the user uses a \forall (or an \exists) within a pre- or postcondition (not to be mistaken with global forall declarations). In this case, it is not decided yet whether to register the corresponding variable or not. The current implementation does consider a \forall variable logical, but without officially registering them. It is possible that this should be changed in the future, if that should be needed by the team at Kaiserslautern.

In that case, a difficulty is that the same name could be used several times (in two different succeeding – not nested - \forall), and we should rename them so as to avoid collisions in the logical variable registry.

Model fields still need a little patch to be properly handled (not as concrete locations). They are marked with TODOs in the code.

VI. Annex: Katja Interface

Since this interface changes and adapts to the needs, one might find useful, for possible debugging purposes, to find included the interface for Katja terms and formulas which was in use when this paper was written.

1. Terms

```
package jive.ContainerInterface.Term

Term = Constant
     | Variable
     | FunctionApplication
     | UnaryOpTerm
     | BinaryOpTerm
     | ConditionalTerm
     | Field ( String name, ClassOrInterfaceType dtype, Type type )
     | SetTerm (TermList elems, Type type)


TermList * Term

Constant = True ( Type type )
         | False ( Type type )
         | ByteLiteral    ( Byte value, Type type )
         | ShortLiteral   ( Short value, Type type )
         | IntegerLiteral ( Integer value, Type type )
         | LongLiteral    ( Long value, Type type )
         | Null ( Type type )
         | Normal( Type type )
         | Break( Type type )
         | Exc( Type type )
         | Ret( Type type )
         | DeclaredConst ( String theory, String name, Type type )

Variable = ProgVar    ( String name, Type type )
         | LogicalVar ( String name, Type type )
         | Dollar ( Type type )
         | Chi  ( Type type )
         | This ( Type type )
         | ResV ( Type type )
         | ExcV ( Type type )


FunctionApplication = FuncAppl ( DeclaredConst func, TermList args, Type type )
                    | Tuple ( TermList entries, Type type )


UnaryOpTerm (UnaryOp op, Term t, Type type)

BinaryOpTerm = BinOpTermTerm (Term term1, BinaryOp   op, Term term2, Type type)
             | BinOpTermType (Term term1, BinaryOpT  op, Type term2, Type type)
             | BinOpTypeType (Type term1, BinaryOpTT op, Type term2, Type type)

ConditionalTerm ( Term condition, Term thenExpr, Term elseExpr, Type type )


UnaryOp = JNot()         // Java boolean negation !
        | JBitCompl()    // bit complement
        | JUPlus()       // unary plus +
        | JUMinus()      // unary minus -

BinaryOp = JLogicalAnd()     // strict AND &
         | JLogicalOr()      // strict OR |
         | JConditionalAnd() // lazy AND &&
```

```
          | JConditionalOr()  // lazy OR ||

          | JEq          // reference equality ==
          | JNotEq       // reference inequality !=
          | JLess()      // comparison <
          | JLessEq()    // comparison <=
          | Element()    // subset relation \in

          | JLShift()      // left shift <<
          | JRShift()      // right shift >>
          | JURShift()     // unsigned right shift >>>
          | JPlus()         // math operator +
          | JMinus()        // math operator -
          | JTimes()        // math operator *
          | JDiv()          // math operator /
          | JMod()          // math operator %
          | JBitAnd()        // bitwise AND &
          | JBitOr()         // bitwise OR |
          | JBitXOr()        // bitwise XOR ^

BinaryOpT = InstanceOf()  // JML instanceof operator

BinaryOpTT = Subtype()       // JML: <:
           | ProperSubtype() // JML: <<:
           | JEq()        // reference equality ==
           | JNotEq()     // reference inequality !=
```

2. Types

```
Type = Store  ()
     | Status ()
     | ArrayBaseType
     | NullType ()
     | FunctionType ( Type dtype, Type rtype )  // _ -> _
     | TupleType                                // _ *  _
     | SetType ()
     | JMLObjectSet ()
     | JMLValueSet  ()

ArrayBaseType = ReferenceType
              | PrimitiveType
              | ProverType (String theory, String type)

ReferenceType = ClassOrInterfaceType
              | ArrayType ( ArrayBaseType type )

ClassOrInterfaceType = ClassType
                     | InterfaceType ( String name, StringList pkg )

ClassType = ConcreteClassType ( String name, StringList pkg )
          | AbstractClassType ( String name, StringList pkg )

StringList * String

PrimitiveType = JBoolean()
              | JByte()
              | JShort()
              | JInt()
              | JLong()

TupleType * Type
```

3. Formulas

```
package jive.ContainerInterface.Formula

import term.katja

Formula = IsTrue
        | FNot
        | BindingFormula
        | BinaryFormula

FormulaList * Formula

IsTrue( Term t )   // here the transition Term -> Formula takes place
FNot ( Formula formula )

BindingFormula ( BindingOp op,
                 BindingList bindingList,
                 Formula formula )

BindingList * Binding
Binding (LogicalVar lv)

BindingOp = Forall()
          | Exists()

BinaryFormula ( Formula left, FormulaOp op, Formula right )

FormulaOp =  FAnd()        // /\ logical and
           | FOr()         // \/ logical or
           | FImplies()    // logical implication ==>
           | FIff()        // logical equivalence <==>
           | FNotIff()     // logical inequivalence <=!=>
```

VII. Bibliography

[1] G.T. Leavens, Y. Cheon, Design by Contract with JML. Last revised October 2004.

[2] G. Leavens, A. Baker and C. Ruby. Preliminary design of JML: a notation for detailed design. Technical report, Iowa State University, last revised June 2004.

[3] A. D. Raghavan, G.T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.

[4] A. Darvas, P. Mueller, Semantics of JML Specifications in Jive. Technical Report, Software Component Technology Group, ETH Zurich.

[5] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, J. Kiniry. JML Reference Manual. Last revised January 2005.

[6] J. Meyer, A. Poetzsch-Heffter. An Architecture for Interactive Program Provers. Fernuniversität Hagen, Germany.

[7] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. Isabelle/HOL, a proof assistant for first-order logic. April 20th , 2004, Springer-Verlag.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata. Extended Static Checking for Java. Compaq Systems Research Center, Palo Alto. PLDI'02, June 2002, Berlin Germany.

[9] Peter Müller, Arnd Poetzsch-Heffter. Universes: a type system for Alias and Dependency control. Fernuniversität Hagen, Germany.