Improving JML's assignable clause analysis

Cui Ye

TR #06-19

July 2006

Keywords: JML, frame axiom, modifies clause, assignable clause, runtime assertion checking, static analysis, intraprocedural analysis, interprocedural analysis, precondition, assignable field set, flow predicate, precision, safety.

2006 CR Categories: D.2.1 [*Software Engineering*] Requirements Specifications — languages; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, logics of programs, mechanical verification, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — program analysis.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011-1041, USA

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I thank all those who helped me with my research and thesis work. Gary T. Leavens, my major professor as well as a mentor to me, guided my study and research during my years at Iowa State. He led me into the academic world and stimulated my interest in specification languages. He is always patient and ready to help with my questions. Dr. Leavens spent numerous hours with me discussing ideas and technical details, which eventually led to this thesis, and even helped to improve my English. I also thank the other committee members, Samik Basu and Ratnesh Kumar for their help and contributions to this work.

I thank the JML developers, who contributed their hard works to jmlc, especially Clyde Ruby, who patiently explained the existing tool implementations to me, shared his knowledge on jmlc, and discussed design and implementation issues with me.

Also, I want to thank my lab-mates: Kristina Boysen, Neeraj Khanolkar, Steve Shaner, and Faraz Hussain, for sharing comments on my thesis and defense, discussing both research and non-research topics, and making my study at Iowa State such a pleasant experience.

Last but not the least, I want to thank my parents for their constant love and care. Though they are not experts in my area, they are great mentors to me on how to be a good person, how to work hard, and much more. I also want to thank Fengming, who's always been there for me with his love, support, encouragement, and patience.

ABSTRACT

The Java Modeling Language (JML) is a formal behavioral interface specification language (BISL) for Java. Its RAC tool (jmlc) performs runtime assertion checking by embedding assertions that check user specifications into the source code compiled for user program.

Reasoning about state changes, that is, about side effects, is crucial to reasoning about programs. Such reasoning further affects proving program correctness and analyzing interesting properties. In JML, the assignable clause is for the purpose of specifying possible side effects that could happen in a method. This thesis work focuses on making sure that the assignable clause is checked appropriately by jmlc.

To perform the assignable clause checking, we combine runtime and static checking. The runtime checking follows the fashion of current jmlc. It generates assertions that check whether certain side effects are allowed in a method and embeds them into user programs. These assertions are checked at runtime. Meanwhile, the static checking collects useful predicates appearing in the method's source code, and its strategy is to use the available predicate environments to prove target assertion predicates at the points right before embedding the target assertions into user programs. Examples of useful predicates are if-conditions, loop tests, predicates of assertions appearing in a method, etc. If the static checking could prove the target assertion, then there is no need to go on with the rest of runtime checking. Otherwise, the tool continues with inserting the assertion. The runtime checking improves the precision, while the static one improves runtime performance.

CHAPTER 1 Introduction

This chapter presents an overview of the thesis, which starts with relevant background information, followed by the objectives, problem summary, and a brief outline of my solution to the problems. Finally, I summarize my contributions.

1.1 Background

The work in this thesis is based on the current Java Modeling Language and its runtime assertion checking tool support. The Java Modeling Language (JML) is a formal behavioral interface specification language (BISL) for Java. It uses Java expressions to specify the behaviors of program modules, but is not limited to that. JML also has a rich set of specification constructs such as quantifiers, logical connectives, and various expression notations [7, 2]. In JML, pre- and postconditions [4] are most commonly used to specify the behavior of a method; this could also apply the idea of Design by Contract [6, 8]. JML supports inheritance, subtyping and refinement of specifications. It also provides syntactic sugars and different specification styles to ease the way of writing specifications.

The following sections give an introduction to part of JML's syntax and some concepts that are closely related to the work in this thesis, including JML annotations, JML assertions and expressions, some highlights of method specifications, and the **r**untime **a**ssertion **c**hecking (RAC) tool. The RAC tool is also referred to as "jmlc" in this thesis.

1.1.1 JML Annotation

JML annotations are comments for the Java compiler, but get processed as specifications by the JML compiler. JML annotations appear in a Java program as single line comments followed with an at-sign, `//@`, or, C-style comments starting with `/*@` and ending with `@*/`. The text of an annotation is either the content following `//@`, or the text between signs `/*@` and `@*/`. In latter form, at-signs at the beginning of lines are ignored, which is usually used to help the reader see the extent of an annotation [5]. Figure 1.1 is a simple example.

1.1.2 JML Assertions and Expressions

JML assertions and expressions are written in Java's expression syntax, but they must be *pure*. That is, operations that cause side-effects cannot appear in JML assertions and expressions. But Java assertions and expressions allow side-effects. Another difference between JML assertions and Java assertions is that JML assertions can only appear in JML annotations.

In addition, there are several expressive JML-specific constructs that could be used in JML assertions and expression. For example, `\old(`E`)` and `\pre(`E`)` represent the pre-state value of expression E. The term *pre-state value of* E refers to the value at the beginning of a method execution. `\result` stands for the return value of the method. For a complete list of JML's extension to Java expression, please refer to the JML Reference Manual [7].

In JML, one can write in-line JML assertions, which are interwoven with Java code. Figure 1.1 is a code example of JML specification with an in-line assertion (the first line of the method body).

1.1.3 Method Specifications

Since JML provides large, extensive syntax and several features, it is difficult to cover them all here. We are going to focus on those most relevantly to this these work in this section. For a

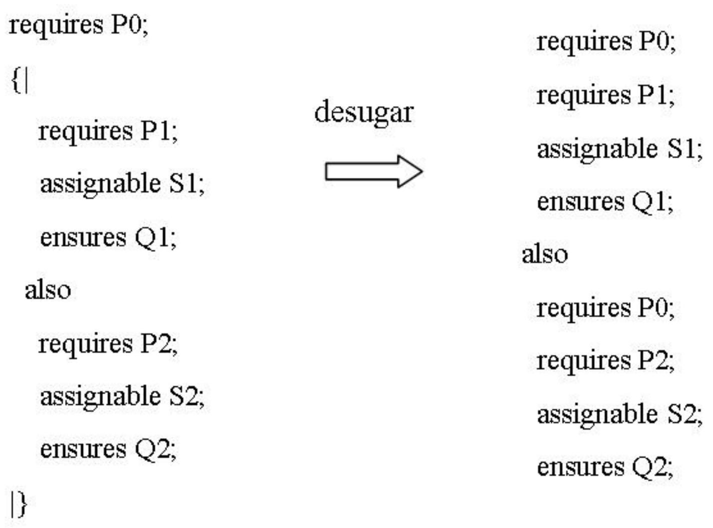
```
/*@ requires x != 0;
  @ assignable y1, y2;
  @*/
  public void m(){
      //@ assert x != 0;
      y1 ++;
      y2 ++;
  }
```

Figure 1.1 Example of JML specification

comprehensive introduction, please refer to the *JML Reference Manual* [7] and other documents [2, 5].

1.1.3.1 Specification Clauses

There are two specification clauses closely related to this thesis work. The first one is the *requires* clause. It is used to specify *preconditions* of a certain method. The other one is the *assignable* clause. My work concentrated on the *assignable* clause. It is used for defining the frame axioms of a method [1]. In JML, the specification of a frame axiom starts with the keyword `assignable`, followed by a list of storage references [7]. Each storage reference listed in the assignable clause represents a data group [11], which is a set of locations. The meaning of the assignable clause is that only the locations in the named data groups are allowed to be assigned in the method body. However, local variables and locations created during the method's execution are not limited by the assignable clause. Please note that in later sections or chapters, these "locations" specified in the assignable clause are also referred to as "fields" or "class fields". They are interchangeable concepts here. Take the example in Figure 1.1, fields `y1` and `y2` are listed in the assignable clause of method `m1`, so they are allowed to be assigned in the body of `m1`. If there is another variable `y3` in the same data group as texttty1, then `y3`



```
requires P0;
{|
   requires P1;
   assignable S1;
   ensures Q1;
  also
   requires P2;
   assignable S2;
   ensures Q2;
|}
```

```
  requires P0;
  requires P1;
  assignable S1;
  ensures Q1;
also
  requires P0;
  requires P2;
  assignable S2;
  ensures Q2;
```

Figure 1.2 Desugaring nested method specification

is also assignable.

1.1.3.2 Heavyweight and Lightweight

JML provides two styles of specification: heavyweight specifications and lightweight specifications. A *heavyweight* specification starts with a behavior keyword, such as `normal_behavior`, and each method specification clause has a well-defined default meaning. A *lightweight* specification does not start with behavior keyword, and only the clauses that the user is interested in will be specified. Those omitted method specification clauses, they default to `\not_specified`.

1.1.3.3 Syntactic Sugar

Nested method specification is allowed in JML, where common pre-state clause such as requires clauses can be factored out [2], [7]. Figure 1.2 shows an example of a nested method specification, and the flat method specification it is desugared into.

1.1.4 Runtime Assertion Checking

Cheon [2] describes how runtime assertion checking (RAC) works in the jmlc tool. Assertions tell what are true at certain points of program [4]. The jmlc tool checks specification assertions during runtime, which may help the programmers who write formal specifications immediately. It executes these assertions to check the validity of an implementation. Meyer and others have pointed out that checking assertions at runtime is a practical and effective means for debugging programs [8, 10]. Besides, assertions not only help in debugging and testing, but also provide the ability to formally prove program correctness [12].

Jmlc takes JML specifications, translates them into statements that will check for assertion violations at runtime, and embeds these assertion checks into the code compiled for user program modules. All these checks are transparent to users, because nothing is changed unless a violation occurs [2].

1.1.5 Precision and Safety

Since this thesis work involves static analysis on assignable clause checking, we need to discuss what precision and safety [9] are, respectively, in our analysis. We say an analysis is more *precise* [9], if the analysis result has a closer meaning to the specification. If the semantics of a method specification means field x is assignable in the method under precondition p, then the analysis checks that only field x could be assigned when p is true at the beginning of method execution; no other fields are allowed to be assigned in this method. This analysis is more precise than the one that allows both fields x and y to be assignable in the method, for example.

A safe but imprecise [9] analysis allows a larger set of implementations than a precise analysis. For the same specification example as above, if the analysis allows both fields x and y to be assignable in the method under precondition p, which is not exactly the same according to the specification, then this is an imprecise but safe analysis. A safe but imprecise analysis is usually used when it is too expensive to give a more precise one.

Specifically, in our case of checking JML assignable clauses, we say it is $unsafe$ [9] if an assignable field is reported as unassignable. However, it is safe (but imprecise) if an unassignable field is interpreted as assignable.

1.2 Objective

The essential overall goal of JML and its tool support is to ease the writing of formal interface specifications and reward these efforts by checking specification assertions during the execution of programs [2, 5]. It also aims to help with debugging, testing, reasoning about the correctness of programs, and providing detailed design documentation.

As an important aspect of program behavior, side effects play a crucial role in reasoning about the state changes in programs, which further affects reasoning about program correctness and interesting properties. In JML, the assignable clause is for the purpose of specifying possible side effects that could happen in a method. Thus it is important to make sure that the assignable clause is checked precisely by JML's tool support, in particular by jmlc.

As introduced before, to verify user programs versus specifications, the current jmlc tool translates specifications into assertions and performs checking at runtime. As a result, a runtime penalty is inevitable. This provides both a challenge and opportunity for static program analysis. By using both static and runtime checking for the assignable clause, we could use the static analysis to reduce the cost of runtime penalty, while guaranteeing precise results by runtime checking. Proving such hybrid checking is the objective of this thesis.

```
/*@ requires x > 0;
  @ assignable y1;
  @
  @ also
  @ requires x < 0;
  @ assignable y2;
  @*/
public void m1(){
    y1 ++;
    y2 ++;
}
```

Figure 1.3 Example of JML specification

1.3 The Problem

To perform hybrid checking of assignable clauses, there are two main problems. One is to develop a runtime assertion checker targeting at assignable clause, which guarantees the bottom line of safe and precise checking results. This checking is modular, which means it only depends on the information from the method it is checking. For example, when it encounters a method call, it will not go into the called method's body. Instead, the checking only uses the called method's specification. The other problem is to design a static checker, which turns off the runtime checking when the program state passes static checking.

When starting this project (April 2006), the tool performed imprecise static checking on assignable clauses. It collected all the locations appearing in the assignable clause(s) in a method specification, and then checked each assignment in the method body to see whether the assigned field was in the location set or not. The problem was that JML allows multiple specification cases in one interface specification. If different field references are listed under different preconditions, should they be all assignable in all the cases? The answer is obvious: no, that would be imprecise.

I use the example in Figure 1.3 to state the problem. The imprecise checking collects `y1` and `y2` into the assignable set. During the checking, we see there are two fields being assigned in `m1()`, and they are `y1` and `y2`. Both of them are in the assignable set, so there is nothing unsafe with respect to assignable clause checking. However, the actual semantics tells us that `y1` is assignable only if precondition `x > 0` is true, and `y2` is assignable only if precondition `x < 0` is true, so this is imprecise. Also, logically, it is obvious that `x > 0` and `x < 0` cannot be true at the same time. Thus `y1` and `y2` cannot be both assignable in `m1()`. So, we know at least one of `y1` and `y2` is not assignable, but this problem is not reported by current tool.

1.4 Approach

To solve the problem of imprecision, we perform precise runtime checking that distinguishes different assignable field sets from different specification cases, which means to bound the assignable field sets to the corresponding preconditions. When it comes to the point of checking whether a field is assignable or not in program, the tool will insert an assertion that verifies the corresponding precondition before the assignment, which will be checked at runtime. For the example in Figure 1.3, assignable field set {`y1`} will be bound to precondition `x > 0`, while assignable field set {`y2`} will be bound to precondition `x < 0`. If `y1` is assigned in `m1()`, then an assertion of `x > 0` will be inserted before the assignment. The same for `y2` and `x < 0`.

While building a precise basis of runtime checking, I use a static analysis to reduce its runtime cost. The easy case is if there is only one specification case, then only the fields listed in the assignable clause are assignable. In this case, we can statically tell there is violation or not by looking up the assigned fields in the specification. For the case of multiple specification cases, I take advantage of the predicates appearing in if statements, while-loops, and Java/JML assert statements. These predicates reveal some information of the states at certain points of a program. Since the preconditions are certain information about the method's pre-state, we

```
/*@ requires x > 0;
  @ assignable y ;
  @ ensures y >= 0;
  @
  @ also
  @ requires x < 0;
  @ assignable \nothing;
  @ ensures y = \old(y);
  @*/
  public void m2(){
      if(x > 10){
          y = x-1; // y is assignable for sure.
      }
      else{
          // not sure y is assignable or not statically :
          y = x+1;
      }
  }
```

Figure 1.4 Example of JML specification

could use the information collected from the predicates in the method to refer the information held in preconditions, if the predicates are also only related to the pre-state. In this thesis, we call these useful pre-state predicates, collected from the program, *flow predicates*.

Take the method and its specification in Figure 1.4 for example. In the if-branch, we know for sure that `x > 10`, which further implies that `x > 0` is true. So it is precise to say that `y` is assignable at this point according to the specification, without having to check the assertion dynamically. However, we do not have the same luck in the else branch. Here we know `x <= 10` must be true from the code, but both `x < 0` and part of `x > 0` fall into the range of `x <= 10`. In this case, we cannot say for sure either `y` is assignable or nothing should be assigned in the else-branch statically. So we need to make the judgement based on the checking result of assertion `x > 0` at this point dynamically.

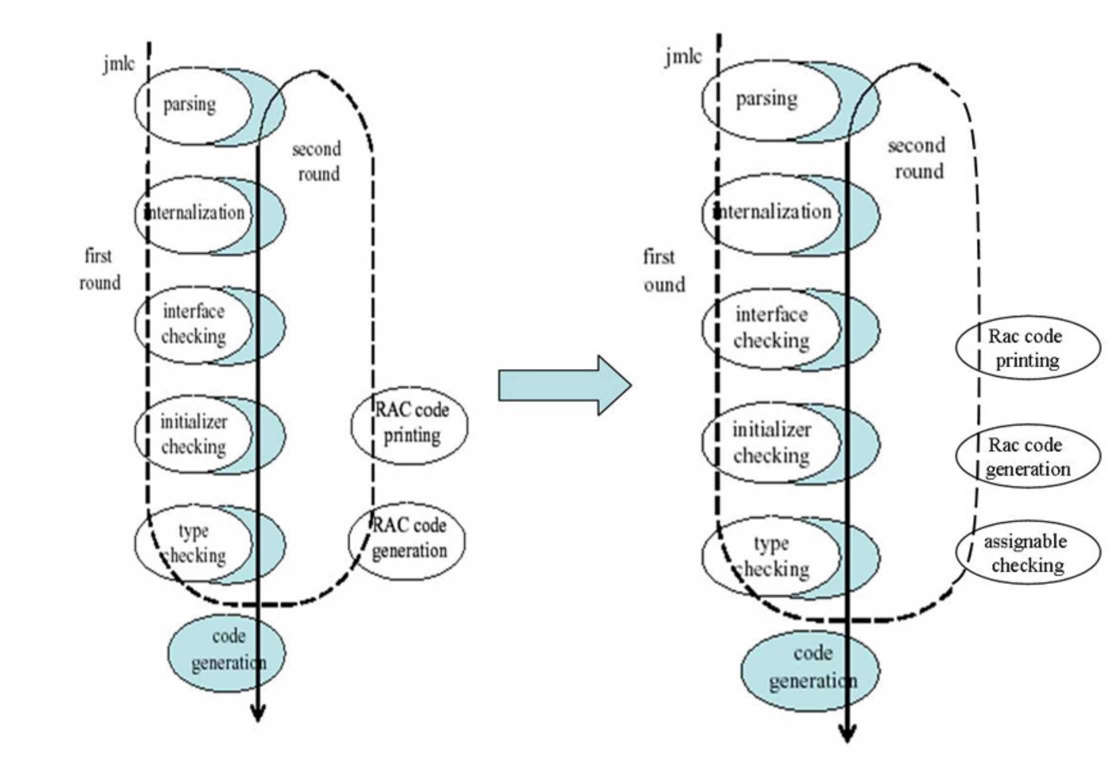


Figure 1.5 The infrastructures of jmlc before and after adding assignable checking

1.5 Implementation

The JML compiler (jmlc), according to Yoonsik Cheon's dissertation work [2], consists of a sequence of compilation passes, which is shown on the left hand side of Figure 1.5. The white ovals represent compilation passes for JML, and the gray ones represent those for MultiJava[1]. MultiJava is the common code base of JML tools [2]. It extends an open-source Java compiler to support open classes and multiple dispatch [3].

My approach for implementing the assignable clause checking is to add a new compilation pass into the sequence, which focuses on the task of checking assignable fields in each method

[1]Several MultiJava-specific compilation passes are omitted in the figure.

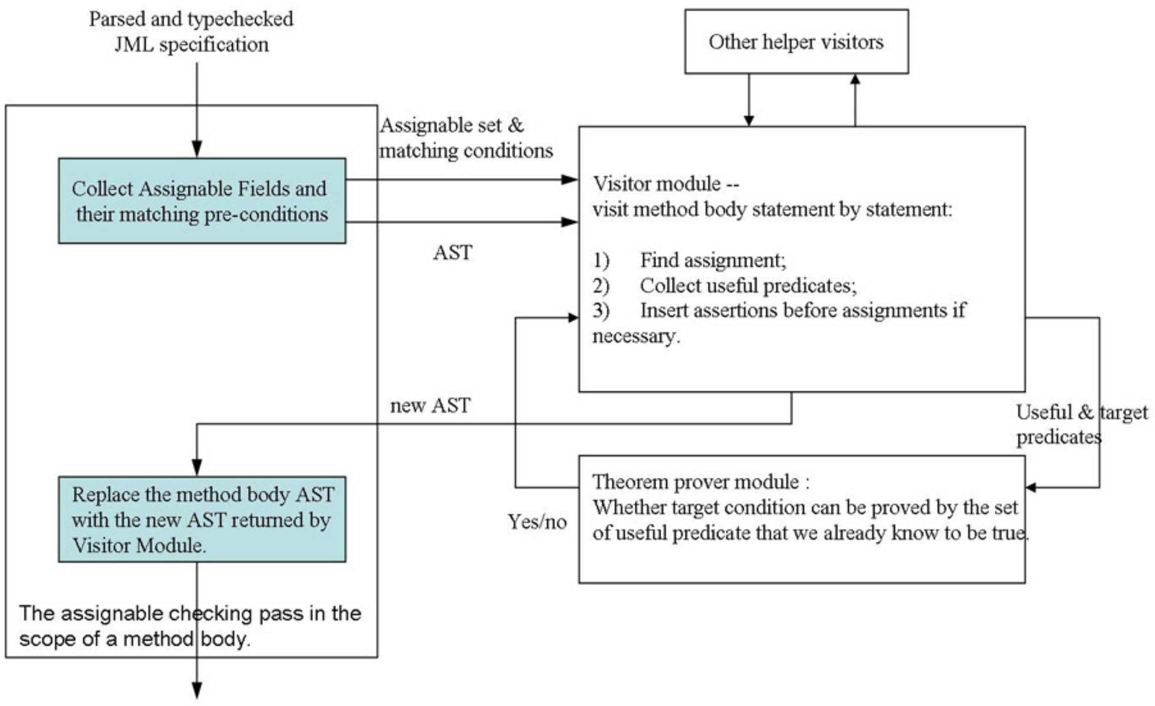

Parsed and typechecked JML specification
Other helper visitors
Assignable set & matching conditions
Collect Assignable Fields and their matching pre-conditions
AST
Visitor module -- visit method body statement by statement:
1) Find assignment;
2) Collect useful predicates;
3) Insert assertions before assignments if necessary.
new AST
Useful & target predicates
Theorem prover module : Whether target condition can be proved by the set of useful predicate that we already know to be true.
Yes/no
Replace the method body AST with the new AST returned by Visitor Module.
The assignable checking pass in the scope of a method body.

Figure 1.6 The structure of assignable checking pass

body. When this is done, the new sequence of compilation passes for JML compiler will look like the part on the right hand side of Figure 1.5.

For the new assignable checking pass, I reuse the information retrieved by previous compilation passes as much as possible, including assignable field specifications, preconditions, and abstract syntax trees of method bodies. The output of the pass would be a modified abstract syntax tree of the method body that is being checked, with the necessary assertions inserted before the corresponding assignments. The big picture of this pass is shown in Figure 1.6. Inside this pass, the main module is a visitor pattern that visits a method body statement by statement, to find assignments. When an assignment is found, the module looks up the field assigned in the collection of assignable fields and matching preconditions. If no matching precondition is found, then it reports an error. Otherwise, it inserts a JML assert statement with the matching precondition as the asserted predicate. Meanwhile, the main module also collects flow predicates[2] in the program as it goes on. An important procedure used by the main module is the static checking procedure. This is performed right before the insertion of an assertion. We are planning to plug in a theorem prover and feed it with the set of flow predicates obtained from the static analysis and the target (pre-)condition that is to be proved. Based on the result returned by the theorem prover, the main module decides whether it will insert the assertion or not.

1.6 Contributions

This thesis makes several contributions.

As discussed in section 1.2, side effects play an important role in program reasoning. In JML, the assignable clause is used to specify possible side effects appearing in the program. Hence, precise checking of assignable clauses will help to improve the validity guarantees provided by the jmlc tool.

Moreover, static checking helps reducing runtime penalty caused by runtime assertion checking, while not giving up on precision. It also provides an interesting feature of jmlc by combining static and dynamic analysis in one specific checking system.

Lastly, it improves jmlc tool support for JML. For example, I introduced the pass of assignable clause checking into jmlc (Figure 1.5). I also fixed the problem that the JML constructs `\old` and `\pre` expressions are not supported in in-line assertions. Both of the constructs represent the pre-state value of a given expression. We will see the details in Chapter 2, when talking about the runtime assertion checking on the assignable clause.

[2]The concept of flow predicate is introduced in the Approach section.

1.7 Outline

The rest of the thesis is organized as follows.

In the main body of this thesis, Chapter 2 and 3, I present all the interesting problems that came up during my work, and explain all the interesting implementation details. Chapter 2 mainly focuses on runtime checking, while Chapter 3 talks about how the static analysis and checking work.

Finally in Chapter 4, I conclude the thesis, and outline possible future research work.

CHAPTER 2 Precise Runtime Assertion Checking on Assignable Clauses

In this chapter, after introducing the main ideas of this thesis work and relevant JML background, I explain my solution to runtime checking. First, I briefly introduce the work that is already done and how it is related to my work. Then I present some interesting detailed problems encountered during my implementation.

Before looking into those details, let us revisit the strategy for performing precise runtime assertion checking on assignable clauses. To distinguish different assignable field sets in different specification cases, we need to bind the assignable field sets to their matching preconditions. To check whether a field is assignable in a statement, we will insert an assertion that verifies the matching precondition before the assignment, which will be checked at runtime. Here, the runtime assertion checking is intraprocedural [9]. That is, each method is a procedure. The runtime checking checks the validity of assignments in one method body and it does not care about method calls.

2.1 Work Already Done

As introduced in Chapter 1, there was an imprecise implementation of assignable checking, which was also supposed to happen at the place where I added the new compilation pass shown in Figure 1.5. To perform a checking on assignable clauses, there is something must be done in both the old and new implementations. First of all, the task of checking assignable clauses should be carried out after parsing and typechecking the source files. This is because we want to make sure that the program passes both the syntax checking and the semantics checking of the compiler before it being verified against its specification. Secondly, this checking should happen in the scope of a method definition, while not on the level of, for example, either class or statement, because the assignable clause is a method specification clause in JML. Further, during the process of parsing and typechecking, there should be some data structures created to record the information in the source files. For example, the abstract syntax trees (ASTs) of the code and the specification, the type of a given expression, and so on. Besides their own checking tasks, these passes are also stepstones to the later compilation passes.

In the existing tool implementations, we could find modules that already perform the tasks described above.

For the first one, in Figure 1.5, there are passes for parsing, typechecking, and others (not related to this thesis) before the pass of assignable clause checking.

For the second one above, in the `org.jmlspecs.checker` package, there is an existing implementation of assignable clause checking, in which there is a specific task in `Main` class – `JmlCheckAssignableTask`, which is responsible for the assignable clause checking on all compilation units from the input source files. There is a method called `checkAssignableClauses()` in class `JmlCompilationUnit`, which is called by `JmlCheckAssignableTask`. Finally the task is forwarded to `checkAssignableFields( JmlDataGroupMemberMap )` of `JmlMethodDeclaration`.

Lastly, in the simple version, we at least need the information of assignable locations listed in the assignable clause, data groups from the class and interface specification, fields being assigned in the method body and so on to perform basic checking. Table 2.1 lists the closely relevant types in package `org.jmlspecs.checker` that are already implemented. The assignable field set is implemented as `JmlAssignableFieldSet`. It simply stores a set of data groups listed in the assignable clause. Thus it has all the fields that are assignable according to the assignable clause specification.

`JExpression` and its subtypes	Represents expressions in MultiJava
`JStatement` and its subtypes	Represents statements in MultiJava
`JBlock`	Represents a code block in MultiJava
`JmlAssignableFieldSet`	The set of assignable fields collected from specs
`JmlDataGroupMemberMap`	Data group information defined in specification
`JmlMethodDeclaration`	A JML method
`JmlExpression` and its subtypes	Expressions in JML, an extension of JExpression
`JmlStatement` and its subtypes	Statements in JML, an extension of JStatement
`JmlPredicate`	Predicate in a JML specification clauses or assertions
`DesugarSpec`	A visitor pattern used to desugar nested specs

Table 2.1 Important types in the existing `org.jmlspecs.checker`

I reuse the scheme of `JmlCheckAssignableTask` and the way it forwards the task from the top of an AST to a node that is concrete enough to perform the task (e.g. method level). We will see how these types are reused in my solution in the following sections.

2.2 The Main Method for Checking Assignable Clause

The `checkAssignableFields(JmlDataGroupMemberMap)` of `JmlMethodDeclaration` is the main procedure for checking a method specification's assignable clause. When it starts, it first desugars the method specification into flat heavyweight style, in case there is nested specification. Next, it calls the procedure `getAssignableMap()` that actually collects the information we want and stores it into a `JmlAssignableMap` object. Details on how to collect the information and the related implementation types are explained in Section 2.3. Then, the procedure creates an instance of a visitor pattern `CheckAssignmentInMethodBody`. The visitor traverses the method body, which is stored as a tree structure, to generate assertions and insert them at necessary places. Details on this visitor and how it generates and inserts assertions are discussed in Section 2.4 and 2.5. When the control flow coming back to the main procedure after visiting the method body, it looks for pre-state expressions used in the process, and generates fresh local variables at the beginning of the method body, which is being checked, to store pre-state values. The last part is shown in Section 2.6.

2.3 Collect Necessary Information

The simple version of assignable clause checking only collects locations specified in assignable clauses. To solve the imprecision problem, we need something more than that – matching preconditions with assignable field sets. In the `org.jmlspecs.checker` package, I introduced a new type called `JmlAssignableMap`, which acts as a map, corresponding the assignable fields mentioned in an assignable clause to their precondition predicates in the same specification case. In the implementation, it maps a `JmlAssignableFieldSet` object to an `ArrayList` of `JmlPredicate` objects.

For a specification like the one on the left hand side of Figure 1.2, it will first be desugared to a form similar to the one on the right hand side, except that for all the method specification clauses that are not defined there, they will be added into the desugared specification with default values. Intuitively, this procedure is performed on the AST of the pre-processed JML method specification, and is forwarded among the tree nodes. Since JML supports inheritance, subtyping and refinement of specifications, the AST of a method specification used here is pre-processed by some passes before the assignable clause checking to contain all the specification information related to this method. Since these passes are not closely related to the work here, we will not go into the details. Figure 2.1 presents the implementation hierarchy of a method specification and the parts collecting information for assignable clause checking. It also shows how the method calls are forwarded in the hierarchy, which is also the way that method calls are forwarded among the AST nodes of a method specification. For example, the task of collecting relevant information first starts from the `JmlMethodDeclaration`. Then it

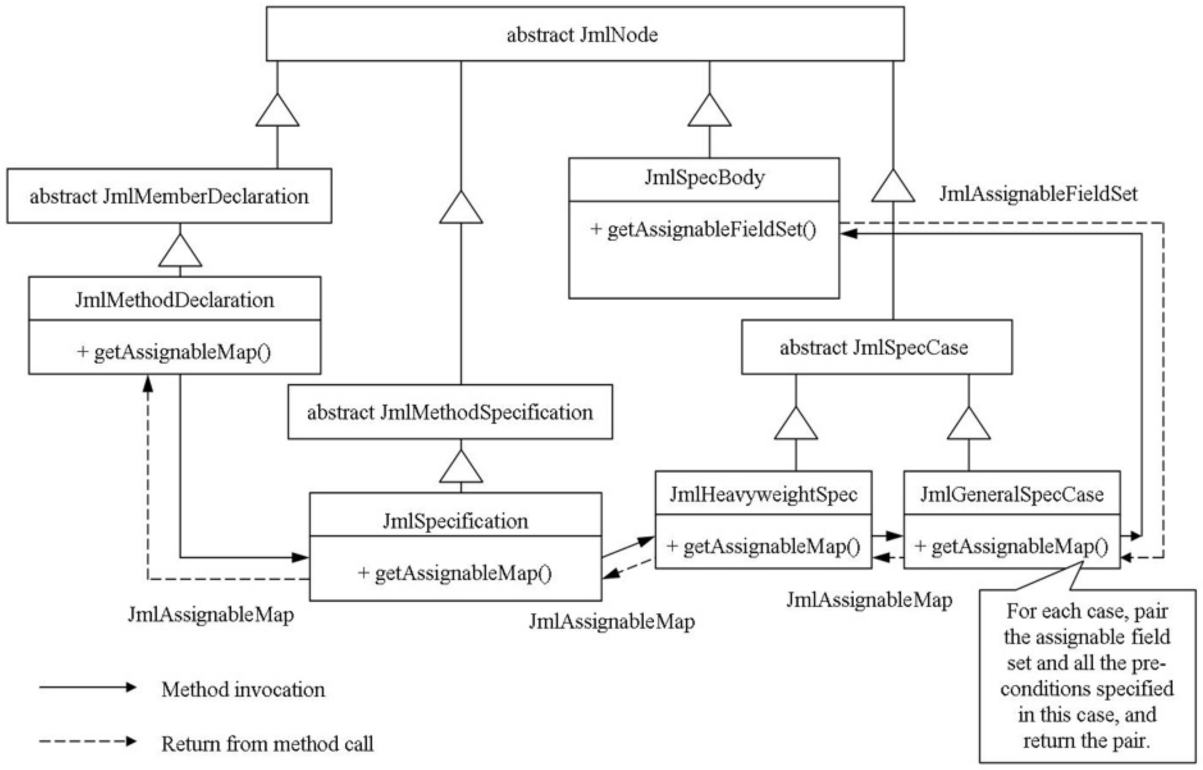


Figure 2.1 Hierarchy of implementation collecting spec info

goes to the top level of the specification's AST (`JmlMethodSpecification`). From there, it is forwarded to each heavyweight specification (`JmlHeavyweightSpecification`) and then each general specification case (`JmlGeneralSpecCase`). Finally, in each case, the task is concrete to each specification clause. This happens in `JmlSpecBody`, which usually has leaf nodes as its children in an AST of a method specification. The leaf nodes are all kinds of method specification clauses.

Again, taking the specification in Figure 1.2 as an example, the contents of the result `JmlAssignableMap` object in `JmlMethodDeclaration` will look like $\{(S1, \{P0, P1\}), (S2, \{P0, P2\})\}$, which means there are two cases in the specification, one with assignable set $S1$ under condition $P0$ && $P1$, the other with assignable set $S2$ under condition $P0$ && $P2$. For the case that there is an assignable clause but no requires clause, the `ArrayList` of `JmlPredicate` objects will be empty, meaning no precondition to check. If nothing should be assignable, the `JmlAssignableFieldSet` will be empty. If there is no assignable clause in the method specification at all, then the member field `notSpecified` of the map object will be set to true. If the assignable clause is not specified, then inside this method, everything will be assignable. However, if the method is specified to be pure, then even in the case that assignable clause is not specified, we interpret it as nothing is assignable both inside the method and in those methods called by this method.

2.4 Visitor org.jmlspecs.checker.CheckAssignmentInMethodBody

Having all the necessary information at hand, now we are able to work on the runtime checking. I designed a visitor on method bodies – `CheckAssignmentInMethodBody`. It is responsible for the major functions that we are expecting here.

2.4.1 Statements and Expressions

The MultiJava compiler on which jmlc is based interprets various types of statements and expressions as subtypes of `JStatement` and `JExpression`. Objects of `JStatement` and `JExpression` are shaped as subtrees in an AST of a method. Thus to traverse an AST, we need two types of visitors, one for (blocks of) statements (`CheckAssignmentInMethodBody`), and one for expressions (`CheckAssignmentInExpressionVisitor`) in a statement. Since we are only concerned about the method body here, the default meaning of AST in the later context refers to the method body, which is further a subtree of the whole compilation unit.

The expression visitor's job is very simple – collect assigned fields and method calls in the given expression. For example, if the expression is a `JAssignmentExpression` and the left

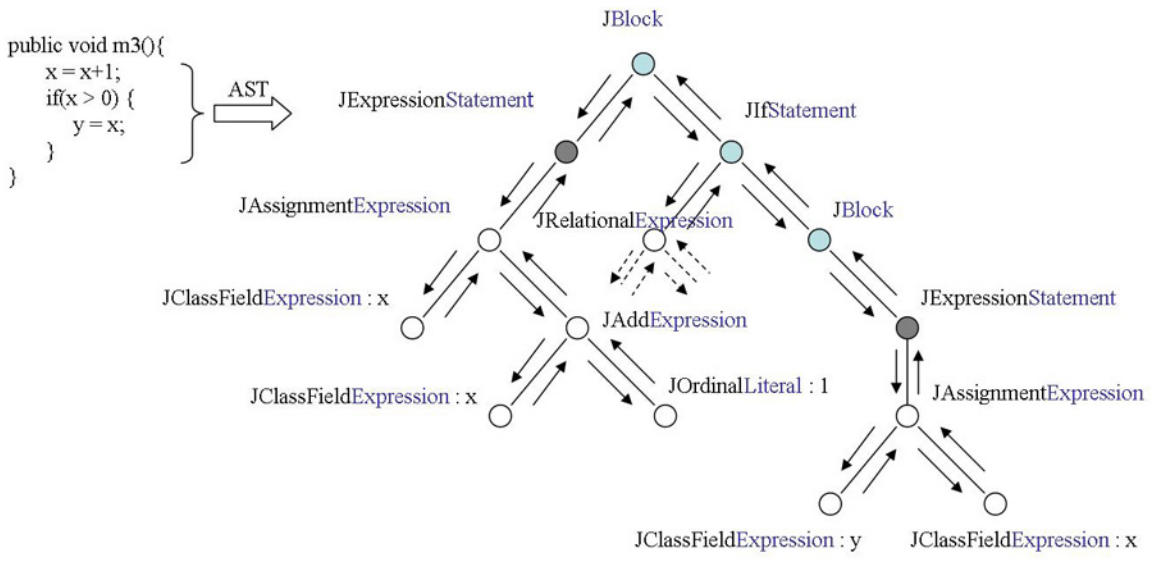


Figure 2.2 AST for an example program

hand side of the assignment is not local, then the visitor collects the left hand side. If there is a method call (`JMethodCallExpression`) in the expression, the visitor collects the object that represents the called method.

The statement visitor is much more complicated. Its main jobs are looking for side-effects, generating and inserting assertions, and collecting flow predicates appearing in the program for static checking. The statement visitor first starts with the whole method body (`JBlock`), the root of the AST, and then traverses that tree, applying the expression visitor when coming to an expression node. When it comes down to a concrete type of statement, for example not a block or compound statements, it instantiates a new `CheckAssignmentInMethodBody` visitor for the statement collecting assigned fields in this single statement. Then it generates the appropriate assertions and inserts them right before this statement.

Figure 2.2 illustrates the AST structure and how a visitor traverses the tree. White nodes are expressions, visited by `CheckAssignmentInExpressionVisitor`. Colored nodes are statements/blocks, visited by `CheckAssignmentInMethodBody`. Among the colored nodes, blue (light grey) ones are those containing a block or a sequence of statements. (Dark) grey ones are single concrete statements. They are also the places where new objects of `CheckAssignmentInMethodBody` are created. The reason for why we use new visitor objects while not the existing one is explained in the next subsection. Arrows in the figure show the order of traversing.

2.4.2 Looking for Side-effects

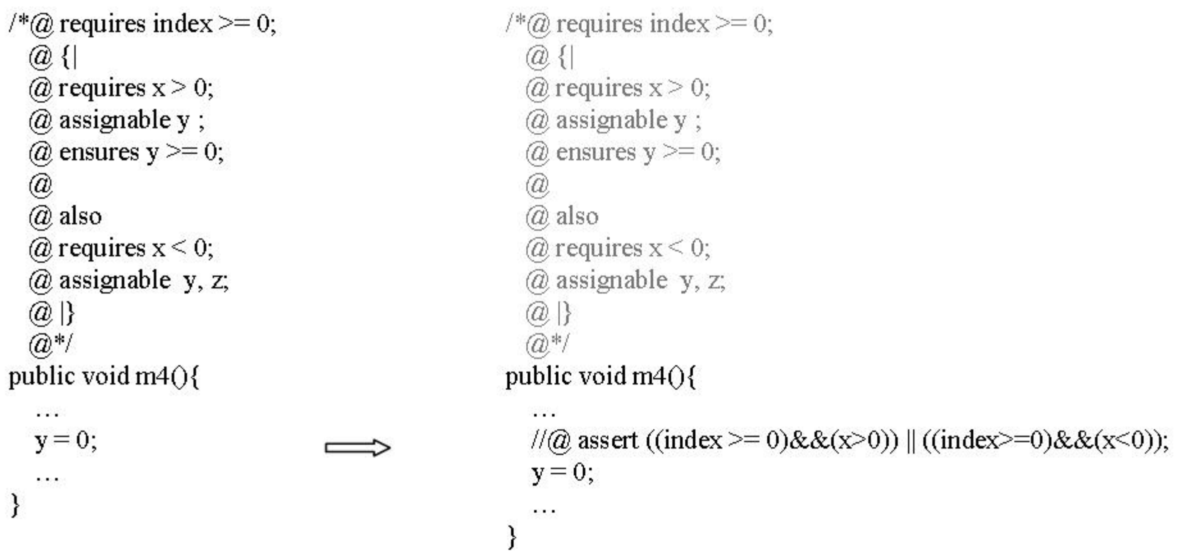
Previously, we have already seen how this part fits into the whole procedure. To be more detailed, `CheckAssignmentInMethodBody` and `CheckAssignmentInExpressionVisitor` both have member fields used to store the fields that are assigned either in a statement or in an expression. Take the example in Figure 2.2. After running an expression visitor on the assignment expression `x = x + 1`, it stores `x` in its member field `assignedFields`, which is a list. When it returns to the visitor on statement `x = x +1;` (`JExpressionStatement`), the statement visitor collects the information from expression visitor's `assignedFields` to its own list, `assignedFields`. If a localized statement visitor (e.g. the one on `JExpressionStatement` object for statement `x = x + 1;`) is returning to its parent level visitor (e.g. the root node *JBlock*), its `assignedFields` will be collected into its parent's `accumAssigned`. The field `accumAssigned` is a list which collects all the fields that are assigned so far in the visible scope (e.g. the whole method body) accumulatively.

Why do we instantiate a new `CheckAssignmentInMethodBody` object at each single statement? If we use only one visitor object from the start on the whole method body, we are able to collect all the assigned fields in the program, but it would be difficult to tell which one happens in which statement exactly. If we collect all the assigned fields via one visitor object, we are still able to perform the runtime checking, for example by inserting an assertion that checks all the assignments at the end of a method body. However, this does not satisfy the requirements well. First of all, we want to localize these assertions to their corresponding assignments. More importantly, localizing the assertions to the right scope will make static checking easier and more efficient later. The new `CheckAssignmentInMethodBody` object instantiated at each single concrete statement has the local information related only to that statement. As a result, we can get local information from the member fields of the local visitor.

2.4.3 Insert Assertions

Now we come to the part that does the real work – generating and inserting runtime assertions. This happens at the level of single concrete statement (dark grey node in Figure 2.2). Let us continue from the previous subsection. When a local `CheckAssignmentInMethodBody` visitor returns to its parent visitor with all the assigned fields it collects, the parent visitor gets those assigned fields and looks up them in the assignable map object one by one. For each assigned field, it looks through all the assignable field sets in the map. If there is a set containing the field, then the matching predicates will be retrieved and connected with conditional-and (`&&`) operator. If there are several sets containing the field, the predicates from different cases will be connected with conditional-or (`||`) operator. The next step is to wrap the result predicate into a `JmlAssertStatement` object, and the work of generating assertion is done for one assigned field.

An example might help to understand the procedure. For the specification in Figure 2.3, the content of the assignable map is {(`{y}`, `{index >= 0, x > 0}`), (`{y, z}`, `{index >= 0, x < 0}`)}. When coming to the assignment `y = 0`, the procedure looks through the sets in the map. It finds `y` in the first set `{y}`, then conditional-and's the predicates in the matching precondition set, which results in a new predicate `(index >= 0) && (x > 0)`. Looking into the



```
/*@ requires index >= 0;
  @ {|
  @ requires x > 0;
  @ assignable y ;
  @ ensures y >= 0;
  @
  @ also
  @ requires x < 0;
  @ assignable  y, z;
  @ |}
  @*/
public void m4(){
   …
   y = 0;
   …
}
```

```
/*@ requires index >= 0;
  @ {|
  @ requires x > 0;
  @ assignable y ;
  @ ensures y >= 0;
  @
  @ also
  @ requires x < 0;
  @ assignable  y, z;
  @ |}
  @*/
public void m4(){
   …
   //@ assert ((index >= 0)&&(x>0)) || ((index>=0)&&(x<0));
   y = 0;
   …
}
```

Figure 2.3 Generating `JmlAssertStatement` for a side-effect

next assignable field set, it finds `y` also contained in set {`y, z`} and then conditional-and's the matching predicates into `(index >= 0) && (x < 0)`. After looking through all the assignable sets, it conditional-or's all the new predicates from previous step, and gets `((index >= 0) && (x > 0)) || ((index >= 0) && (x < 0))` in this example.

Before inserting the assertion, the current child is this single statement with side-effects. Then we compose the new assert statement with the original statement into a sequence of statements (`JCompoundStatement`). To perform the insertion, we just replace the original statement child node with this new `JCompoundStatement` node. Thus the new method body right after inserting the generated `JmlAssertStatement` is on the right hand side in Figure 2.3.

The reason we use `JmlAssertStatement` instead of using a Java assert statement here is because JML allows its own constructs and expressions in the specification. By using

`JmlAssertStatement`, we do not have to worry about these constructs here, but leave this job to a later pass of RAC code generation.

2.5 Pre-state Values

A method's precondition specifies the conditions that the program state should satisfy at the beginning of a method execution. Hence, the values in a precondition all refer to the pre-state values, which are the values at the beginning of a method execution. Since Java allows side-effects, the value of a variable appearing in a precondition might be changed by the method at runtime. In such case, if a new assertion is inserted after possible side-effect(s), then we are not actually checking the right precondition. The solution is conceptually easy – replacing the variables with their old expressions. Remember that old and pre expressions are JML constructs that represent pre-state value of an expression. For example, the predicate in the generated `JmlAssertStatement` in Figure 2.3 is transformed into

```
(\old(index)>=0 && old(x)>0) || (\old(index)>=0 && old(x)<0)
```

This completes the discussion of how to perform runtime assertion checking for assignable clauses. However, before going to the static checking part, there is one more problem left. All the generated JML assertions here are in-line assertions. But when work was began on this thesis, the jmlc did not support `\old` or `\pre` expressions in in-line assertions. The next section presents a solution to this problem.

2.6 Old/Pre Expressions in In-line Assertions

Since `\old(E)`[1] refers to the pre-state value of expression `E`. Pre-state value means the value at the beginning of a method execution. To evaluate `\old(E)`, we just need a way to record the

[1] `\old` and `\pre` are semantically the same. So when referring to `\old`, we also refer to `\pre` here.

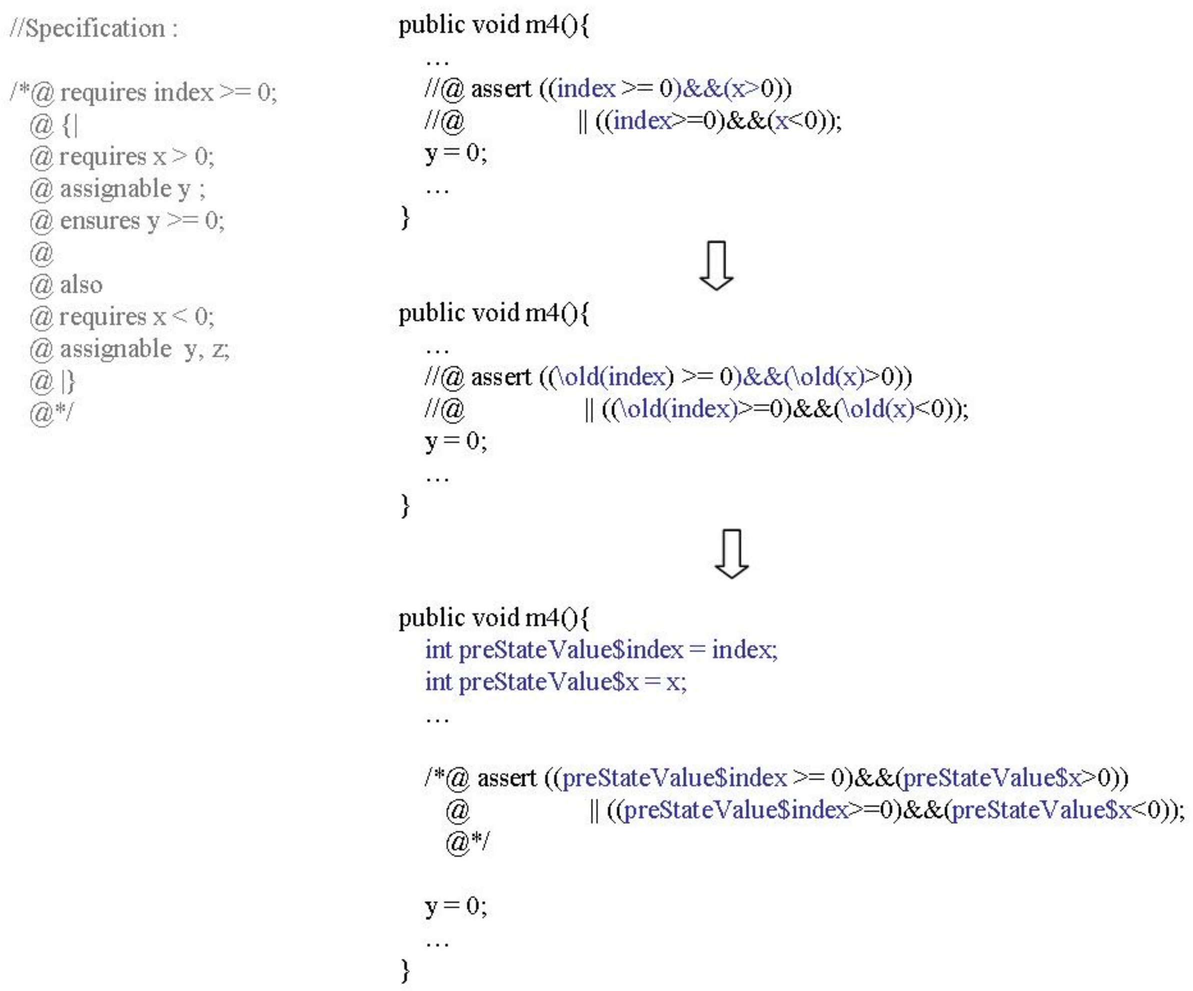
value of E at the beginning of a method and refer to this value wherever the `\old(E)` expression appears.

The way that I solve this problem is searching for all the `\old` expressions in the predicates of all assertions, then generating a fresh local variable to store the value of each expression, and inserting the local variable definitions at the beginning of the method body, finally, replacing the old expressions with their corresponding local variables.

To find out the `\old` expressions in a predicate, we need to traverse the AST for this predicate. Hence, I introduce another visitor `JmlPredicateVisitor`. It traverses the expression tree in the same way as the previous expression visitor. The difference is that the `JmlPredicateVisitor` is only interested in `\old(`E`)` and `\pre(`E`)` that appear in current predicate expression. The `JmlPredicateVisitor` has a field `oldExprs` for storing all the different `\old` expressions it has seen so far, and another field `genVars` for the fresh variables generated for these `\old` expressions. The `\old` expressions and fresh variables are matched one by one.

When the visitor comes into a `JmlOldExpression` or `JmlPreExpression`, it does the following. A boolean flag `isOldChild` is set to true, which means the current child of its parent node is an `\old(`E`)` or `\pre(`E`)` expression. If E is of type `JClassFieldExpression` or `JArrayAccessExpression`, which means it is a leaf node, and the current `oldExprs` does not contain E, then the visitor puts E in `oldExprs`, and creates a new `JLocalVariableExpression` with a new `JVariableDefinition`, using E to initialize the variable, and puts the new local variable object in `genVars`. Otherwise, E is not a leaf node. The visitor keeps traversing the subtree. When it returns, it generates a fresh variable for this expression as above. Once an expression is added to `oldExprs`, a new local variable is generated for it. In this way, they are matched one by one.

After returning from traversing the subtree of each internal node, the visitor checks whether



```
//Specification :

/*@ requires index >= 0;
  @ {|
  @ requires x > 0;
  @ assignable y ;
  @ ensures y >= 0;
  @
  @ also
  @ requires x < 0;
  @ assignable  y, z;
  @ |}
  @*/
```

```
public void m4(){
   …
   //@ assert ((index >= 0)&&(x>0))
   //@                || ((index>=0)&&(x<0));
   y = 0;
   …
}
```

```
public void m4(){
   …
   //@ assert ((\old(index) >= 0)&&(\old(x)>0))
   //@                || ((\old(index)>=0)&&(\old(x)<0));
   y = 0;
   …
}
```

```
public void m4(){
   int preStateValue$index = index;
   int preStateValue$x = x;
   …

   /*@ assert ((preStateValue$index >= 0)&&(preStateValue$x>0))
     @                || ((preStateValue$index>=0)&&(preStateValue$x<0));
     @*/

   y = 0;
   …
}
```

Figure 2.4 Pre-state values in an in-line assertion

current node has any pre-state expression child, which is either a `\old` expression or a `\pre` expression. If yes, then the visitor looks through the `oldExprs` to find this pre-state expression, and uses the matching element in `genVars` to replace this child. If none of the direct children is pre-state expression, then nothing is changed.

The example on the right hand side of Figure 2.3, which is the result of section "Insert Assertions", now is the top one in Figure 2.4. In the middle, there is the result of previous section "Pre-state Values". The bottom one shows the result of this section.

So far, we have matched preconditions with assignable clause specifications, generated JML assertions from these preconditions to check side-effects in method body, and then used pre-state value to substitute the fields in assertion predicates to ensure the correct semantics. The work of precise runtime assertion checking on assignable clause is done. The next chapter talks about the solution to the static checking problem.

CHAPTER 3 Static Checking on Assignable Clauses

The idea of performing static checking on assignable clause comes from the idea of static program analysis and the observation of the relation between predicates in source code and preconditions used for assignable clause checking. Both of the two kinds of predicates are about program state. One tells what are true about the program states at certain points. The other is what to be proved true about the program pre-state at certain points. In this chapter, I first give the main idea of the static checking. Then I talk about what is to be done in the static checking and formalize it. Finally, we will see some implementation details of the static checking part.

3.1 Main Idea of Static Checking

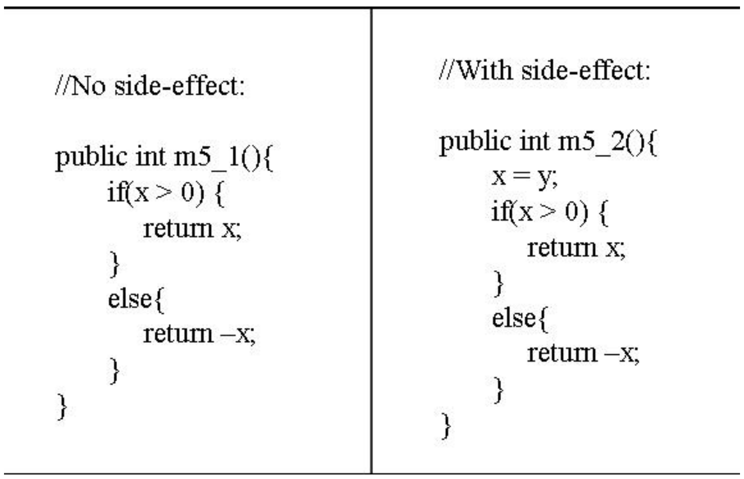
One possible solution to the static analysis is to have a separate pass after the runtime assertion checking. The main job is to remove the JML assert statements, inserted into the code by runtime checking pass, if the assertions are proved to be true in current context statically. However, after observing that this solution requires another traversal of the AST of a method body in the same fashion as the runtime checking, I decided to embed this pass into the previous runtime checking pass. Thus, one time traversing on the method body does the work of both runtime assertion checking and static checking. As a result, we pay only a small cost to combine the interesting new feature of static checking into the dynamic checking.

In last chapter, I explained how the assertions are generated and inserted into the source code AST. Now the main idea of my solution to static checking is that before inserting the assertion into the AST and after generating the assertion to be checked, we send the predicate in the new assertion to another method that has the context knowledge of flow predicates. We ask this method to tell us whether this predicate is true based on the context knowledge it has. If the answer is yes, then we know there is no necessary to perform runtime checking on this one. So in this case we ignore the code inserting part. Otherwise, we leave the precise checking to runtime, and still insert the assertion into the AST as before.

3.2 Formalize Static Analysis

The previous section already gave a brief strategy on how to perform the static checking. Now let us look at it more closely. The first question that arises when thinking in more detail about the implementation is what we know about the context. The knowledge of what to be true in certain context is crucial here. If we do not collect the knowledge safely, then potentially the result of assertion proof could lead to wrong static checking result.

The reason that causes this problem is the fields in the generated assertions only refer to their pre-state values. So if a field is changed in a method, then we cannot use the predicate with this field later in the program to prove the assertion. This is because we cannot use a predicate with updated value of certain field to reason about other predicate(s) with pre-state value of the same field. In a method that has no side-effects, there are fewer troubles. The example code on the left hand side in Figure 3.1 has no side-effect. Thus we know in the if-branch, predicate `x > 0` is true, and in the else-branch, predicate `!(x > 0)` is true. All these variables refer to their pre-state values. Let's say in the if-branch, we want to prove something like `\old(x) > -1`. It is trivial based on the knowledge of `\old(x) > 0`. However, in the method on the right hand side of Figure 3.1, there is an assignment to variable `x`. After that, the variable `x` in the method does not refer to its pre-state value any more. Even later `x > 0` is true in the if-branch,



```
//No side-effect:

public int m5_1(){
      if(x > 0) {
          return x;
      }
      else{
          return –x;
      }
}
```

```
//With side-effect:

public int m5_2(){
      x = y;
      if(x > 0) {
          return x;
      }
      else{
          return –x;
      }
}
```

Figure 3.1 Examples of flow predicates

but we interpret that differently – the updated value of `x` is greater than 0. If we still want to prove `\old(x) > -1`, there is no easy answer in this case, because we only know about the updated `x` not the pre-state `x`.

Now we need some *criteria for safety.*

1. The predicate should only contain variables that are consistent with their pre-state values. That is if there is a variable in the predicate being assigned before in the method, then this predicate should not be included in the context.

2. The predicate should not have side-effects, for the same reason as above.

3. We do not care much about local variables in the assignable checking. Thus we do not want predicate containing local variables, except for those generated by the tool itself in previous runtime checking part.

4. The predicates should have their own visible scope. For example, in method `m5_1()`, predicate `x > 0` is only visible in the if-branch, and predicate `!(x > 0)` is only visible in the else-branch. None of them should be visible outside the if-statement.

The first three conditions could be checked by traversing the predicate. From which we could tell whether it contains side effects, local variables or updated fields that appear in the updated field list. The updated field list is an accumulative list of fields that have been changed throughout the method body.

Based on previous discussion, now I formalize the static checking by defining a set of transition rules shown in Figure 3.2. In these rules, A stands for the context knowledge of what predicates are true at certain point. H is a function that matches a class member field listed in JML assignable clause specification to the predicate/condition generated for this field in the previous runtime assertion checking part. Sign $\models$ means provable. For example, $A \models p$ means predicate p is provable from context knowledge A. On the other hand, $\not\models$ means unprovable. Thus, $A \not\models p$ means predicate p cannot be proved true based on context knowledge A. That means it is not sure whether p is true statically. In each judgement, $\vdash$ is for provability, on the left hand side of $\vdash$ there are the assumptions, and on the right hand side there is the conclusion of the judgement. Typically, each conclusion further consists of two parts. One is a transition from an original statement (or a sequence of statements) to a new statement (or a modified sequence of statements), generally in the form $S \Rightarrow S'$. The second part is the updated context knowledge A.

The rule [assign1] says from assumption A and H, if E has side-effect on a non-local location x, we could come to the conclusion that statement E; will be transformed to statement E;, and the context knowledge A remains the same. This is based on the hypothesis that A proves $H(x)$. It means if E has side-effect on some class field x, and the matching condition that makes the field assignable in the specification is proved based on context knowledge A, then there is no change to the original statement, and the side-effect in E is allowed. Otherwise, if the matching condition cannot be proved, then the analysis inserts a JML assertion in front of E;, which is shown in rule [assign2]. For both of the cases, predicates in A with field x in it [assign1] $\frac{A \vDash H(x)}{A,H \vdash E; \Rightarrow E;, A \backslash \{p \mid p \in A \wedge p \ contains \ x\}}$,

for each non-local location x on which E has a side-effect

[assign2] $\frac{A \nvDash H(x)}{A,H \vdash E; \Rightarrow /{*}@assert \ H(x);@{*}/ \ E;, A \backslash \{p \mid p \in A \wedge p \ contains \ x\}}$,

for each non-local location x on which E has a side-effect

[if] $\frac{\{E\} \cup A,H \vdash S_1 \Rightarrow S_1', A_1 \ ; \ \{!E\} \cup A,H \vdash S_2 \Rightarrow S_2', A_2}{A,H \vdash \ if \ (E) \ S_1 \ else \ S_2 \Rightarrow if \ (E) \ S_1' \ else \ S_2', A}$, if E satisfies the criteria

[seq] $\frac{A,H \vdash S_1 \Rightarrow S_1', A_1 \ ; \ A_1,H \vdash S_2 \Rightarrow S_2', A_2}{A,H \vdash S_1 S_2 \Rightarrow S_1' S_2', A_2}$

[while] $\frac{\{E\} \cup A,H \vdash S \Rightarrow S', A'}{A,H \vdash while(E) \ S \Rightarrow while(E) \ S', A \cup \{!E\}}$, if E satisfies the criteria

[assert] $A, H \ \vdash \ assert \ p; \Rightarrow \ assert \ p;, A \cup \{p\}$, if p satisfies the criteria

[jmlassert] $A, H \ \vdash \ //@assert \ p; \Rightarrow \ //@assert \ p;, \ A \cup \{p\}$, if p satisfies the criteria

[general] $A, H \ \vdash \ S \Rightarrow S, A$, if no other rules apply

Figure 3.2 Transition rules for static analysis on assignable checking

have to be removed from the knowledge A. Take [while] as another example. If the loop test E satisfies the flow-predicate criteria discussed before, then based on assumption A and H, the original while-loop $while(E)\ S$ is transformed to $while(E)\ S'$ and context A is updates with predicate $!E$, because in the control flow $!E$ must be true after the loop. S' is a result of the induction, which takes the loop body and checks it inductively. Another thing to notice is that inside the loop body (in the induction part), predicate E is added to A, because the loop test must be true inside the loop body. The other rules are interpreted in the same fashion as the example discussed above.

3.3 Implementation of Static Checking

As introduced in the main idea of static checking, the implementation is embedded into the visitor `CheckAssignmentInMethodBody` of runtime checking implementation.

To add a new feature, the first thing is to have appropriate fields in the visitor holding the information we need to perform the static checking. According to the formal rules and the criteria, we need A, H and an AST to perform the analysis. It is trivial that H is generated from `assignableMap`. The procedure of finding the matching conditions of a given field is described in the Insert Assertions section of Chapter 2. And we already have a visitor (`CheckAssignmentInMethodBody`) on the AST of a method body, which is one of the major reasons that why we want to embed the static checking into an existing visitor. The important thing left is A. To record set A, I added a field `predKnowledge`, which is a `LinkedList` in class `CheckAssignmentInMethodBody`. It is used to store the list of predicates that are true at certain program point. Now, when the visitor instance comes to a predicate in if, while, or assert statement during traversing the AST of a method body, it is able to collect the predicate into the list `predKnowledge`. However, before adding the predicate into the list, there is one more thing to do – checking whether the predicate satisfies the criteria discussed in previous section or not. If there is only one specification case in the method specification, then we initialize `predKnowledge` with the precondition.

In the implementation of runtime assertion checking, I already explained how to check for side-effects in the method body. Other than that, there is another field `accumAssigned` in `CheckAssignmentInMethodBody` that contains all the class fields whose states have been changed so far at certain program point. Thus by implementing another expression visitor to visit the predicate, we are able to tell whether this given predicate has variable with updated state or has side-effect itself. If a variable in the predicate has occurrence in `accumAssigned`, then its state has been changed in the method. If there is a sub-expression in the predicate that has a side-effect, then the predicate has a side-effect. Also, during traversal of the predicate, if the visitor meets a node of type local variable, then the visitor ignores the predicate. Predicates that do not have the above three problems will be added to the `predKnowledge` of current `CheckAssignmentInMethodBody` object.

The last problem in collecting context knowledge is how to make the predicates visible in correct scopes. Remember in the section of looking for side-effects in Chapter 2, I mentioned that a new `CheckAssignmentInMethodBody` object is instantiated at each single concrete statement to collect the local information related only to that statement. I also said it would help with the static checking. Here is an example on how it works (Figure 3.3). Take a look at method `m5_1()`. When the outermost visitor comes to the if-condition, it creates an expression visitor. The expression visitor visits the condition to check whether it satisfies the first three criteria or not. If no, then nothing is changed. The outermost visitor keeps traversing the then-branch and else-branch. If the condition does satisfy the criteria, then the outer-most visitor keeps the condition temporarily, because the condition is not true on the if-statement level, but inside the then-branch level. When the method body visitor comes down to the statement in then-branch, it initializes a new `CheckAssignmentInMethodBody` object for the statement with

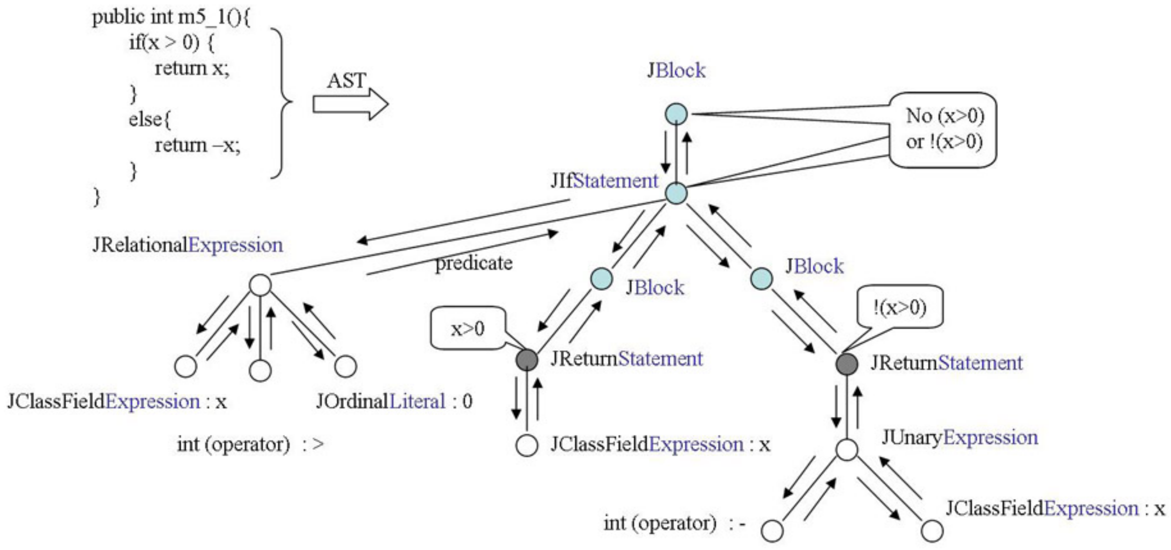


Figure 3.3 An example on scopes of flow predicates

the if-condition added into the new object's `predKnowledge` (as well as other visible predicates, if any). The same happens to the else-branch. The negation of the if-condition will be added into the list of predicates that is used to initialize the `CheckAssignmentInMethodBody` object for statements in the else-branch. Since the condition (or its negation) is only added into the new visitor object for the subtree of then-branch (or else-branch), but not the higher level visitor object. And the new object's life ends after finishing traversing the subtree. So the condition (or its negation) is only visible in its corresponding subtree, which guarantees that it is not visible outside the appropriate scope. In the other case, if there is any predicate to be added to the higher level `CheckAssignmentInMethodBody` object, i.e. after an assert statement or a while-loop, the higher level object will add the new predicates from its child statement to its own `predKnowledge` list, when the visitor on the child statement returns to the higher level visitor.

So far, the problem of context (flow predicates) collecting is solved. As defined in the transition rules, we make the visitor collects the context while traversing the AST. Also we check the predicates to make sure that we only collect the ones that satisfy the criteria. The final step is how to use the context of flow predicates to prove the conditions that we want to check.

3.4 The Hook Method

The last thing needed here is something that could prove the condition based on the context information we collect. A theorem prover would meet this need. What I did in this part is writing a method to hook up the information we got with a theorem prover, although we haven't finally chosen the one that would fit our project the best. But generally, we provide two inputs. One is the target predicate that needs to be proved. The other is the context information that we collect in `predKnowledge`. The result returned from the theorem prover should be either yes or no. Yes means the target predicate can be proved based on our static knowledge set. Thus we do not need further runtime checking any more in this case. If no, it means there is not enough information to prove the target predicate right now, so we continue with inserting the assertion and performing the checking dynamically.

Since we haven't chosen the theorem prover that would fit in the best, what the hook method does currently is translating the target predicate and the context predicates into Strings, and looking through the knowledge set to see if there is any predicate that is exactly the same as the target predicate literally. If yes, then there will be no runtime checking. Otherwise, the tool continues with the work in the runtime checking part.

3.5 Method Calls

For the case that there are method calls in a method body, since the called method is executed in the context of the caller, it is straightforward that the callee needs to satisfy related specifications of the caller. For the assignable clause, we simply require that the callee does not allow any field not specified in the caller's assignable clause to be assigned in callee's method body. But the fields that the callee allows to be assigned could be a subset of the fields that the caller allows. The tool takes an union of the assignable field sets from different specification cases for both the caller and the callee. Then it checks whether the callee's union set is a subset of the caller's union. If there is a field specified in the callee's but not the caller's, then the tool reports an error. This at least guarantees that all the potentially assignable fields in the callee method are allowed in the caller context. Actually, this works in a precise fashion when there is only one specification case for both of the methods. However, when there are multiple specification cases, this checking could give a larger assignable set than the precise result. Thus it is safe, but imprecise.

To make the checking more precise, we still need to work on separating different specification cases as we did in the intraprocedural [9] analysis. A possible solution to improving the precision of interprocedural [9] checking is using runtime assertion checking, which we already used to guarantee the precision of intraprocedural assignable clause checking. Before each method call, we search through all the assignable fields specified by the called method's specification. For each assignable field, if it is also allowed in the caller, then its corresponding precondition should be true right before the method call. If it is not assignable in the caller's context, then its corresponding precondition should not be true. Otherwise, it means the called method allows some field that is not assignable in the caller's context to be assignable in the callee. We could insert assertions checking these information.

```
public class MethodCallExample{
    /*@spec_public@*/ private int x, y, z;
    /*@ requires P1;
      @ assignable x;
      @ also
      @ requires P2;
      @ assignable y;
      @*/
    public void callerM(){
        calledM();
    }
    /*@ requires Q1;
      @ assignable x;
      @ also
      @ requires Q2;
      @ assignable z;
      @*/
    public void calledM(){
        ...
    }
}
```

In the above example class `MethodCallExample`, based on previous discussion, the method call in `callerM()` is legal if the side-effects in `calledM()` not only satisfy `calledM()`'s specifications, but also do not violate specifications of `callerM()`. That is, according to both the specifications, when `calledM()` is called by `callerM()` under precondition `P1`, only `x` is assignable in `calledM()`, while under precondition `P2`, `calledM()` is actually not callable. Though `z` is assignable under precondition `Q2` according to `calledM()`'s specification, it can never happen in this example.

The logic to perform the checking is described as follows.

```
if (callerM() executed under P1){
    if calledM() executed under Q1
        then checking is done
    else reports error : calledM() cannot be called by callerM()
}else reports error : calledM() cannot be called by callerM()
```

This is our strategy of how to perform precise interprocedural assignable clause checking.

CHAPTER 4 Conclusion

4.1 Future Work

To improve the strength of static checking, a theorem prover is needed to perform a more complicated logic proof. The current simple prover will be substituted if time permits.

Also there are some other improvements that could be done to make the process more efficient. For example, we could keep a list of class member fields that have already been checked, no matter statically or dynamically. If there is a reoccurrence of certain field in the list, we do not need to do an extra check again. Because the field has been checked before this point, and if the control flow is able to come to the later point, we know it must pass the checking at some earlier point.

Another thing to think about is that the predicates with updated variable value might not be totally unuseful. For example, the target predicate is `x > 0`, and the current context condition is `x > 5`, but the value of x is changed once before by expression `x ++`. Actually it equivalently means `x1 = x + 1` and `x1 > 5`. Now we could use `x1 = x + 1` and `x1 > 5` to prove `x > 0`. However, this requires much more work in collecting context information and translating them into more useful formulas. This also arises the question – how much effort the static work is worth to make its benefit greater than its cost.

JML is a big project, which makes it difficult to understand all the implementation details. The question on how to document such a big project to help followers get involved easier and faster is also an interesting topic.

4.2 Summary

The work presented in this thesis was motivated by the lack of precise checking on assignable clauses, considering the role that side-effects play in JML and the RAC tool. We want to solve this problem because side effects change program states, which further affects the reasoning about program correctness and some interesting properties. My work solves the imprecision problem on assignable clause checking. The solution guarantees the precision by using runtime assertion checking, while improves runtime performance by using static checking. It helps improving the validity guarantees of jmlc.

BIBLIOGRAPHY

[1] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[2] Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. *Ph.D. dissertation, Department of Computer Science, Iowa State University, TR 05-13*, 4 2003.

[3] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA*, pages 130–145, 2000.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

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

[6] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005.

[7] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok,

and Joseph Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, July 2004.

[8] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, 2 edition, 1997.

[9] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 2 edition, 2005.

[10] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, 21(1):19–31, 1995.

[11] K. Rustan and M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, pages 144–153, 1998.

[12] Jürgen F. H. Winkler and Stefan Kauer. Proving assertions is also useful. *SIGPLAN Notices*, 32(3):38–41, 1997.