Universidade de Pernambuco
Escola Politécnica de Pernambuco
Departamento de Sistemas e Computação

Pós-graduação em Engenharia da Computação

Implementing JML Contracts with AspectJ

por

Henrique Emanuel Mostaert Rebêlo

Dissertação de mestrado

Recife, maio de 2008

UNIVERSIDADE DE PERNAMBUCO
DEPARTAMENTO DE SISTEMAS e COMPUTAÇÃO

Henrique Emanuel Mostaert Rebêlo

Implementing JML Contracts with AspectJ

Este trabalho foi apresentado à Pós-graduação em Engenharia da Computação do Departamento de Sistemas e Computação da Universidade de Pernambuco como requisito para a aprovação da dissertação de mestrado.

ORIENTADORES:
Prof. Ricardo Massa Ferreira Lima
Prof. Márcio Lopes Cornélio

R289i Rebêlo, Henrique Emanuel Mostaert
   Implementing JML Contracts with AspectJ /
Henrique Emanuel Mostaert Rebêlo. - Recife : O Autor,
2008.
   xii, 120 f. : il., fig., tab.
   Dissertação (mestrado) - Universidade de
Pernambuco. DSC. Engenharia da Computação, 2008.

   Inclui bibliografia e apêndice.

   1. Engenharia de software. 2. Linguagens de
programação (Orientada a aspectos) - Aplicação. 4.
Experimentos (Engenharia de software) - Avaliação.
5. Título.

CDU - 681.31.34 UPE

Acknowledgments

I wish to thank my supervisor, Professor Ricardo Massa, for his great guidance, careful supervision, encouragement, and friendship. I will take into account the time that he spent educating me since undergraduate, and for all the lessons and valuable discussions that I will take for the rest of my life.

I wish to thank my co-supervisor, Professor Márcio Cornélio, for his guidance, valuable discussions, refined perception, and friendship. He also spent a lot of time educating me. I am so thankful to him. In fact, he worked as a supervisor.

I am grateful to Professor Paulo Borba and Sérgio Soares for examining this dissertation and making several suggestions to improve it.

Special thanks to Professor Gary Leavens, Professor Patrice Chalin, and Perry James for their several and helpful discussions about JML and its semantics.

I wish to thank Leopoldo Ferreira, an undergraduate student, for his help on early stages of this work.

I would also like to thank Fernando Calheiros from Meantime Mobile Creations (an enterprise by CESAR), in particular of the PIDAM project. He shared a substantial amount of domain knowledge related to Java ME with AspectJ (including the specific AspectJ runtime environment to run Java ME applications with AspectJ that he sent to me). I am most grateful to him.

I wish to thank my MSc colleague César Lins for his friendship, encouragement, and motivation during the development of this work. He discussed several aspects of my work that helped to improved it.

I am grateful to Professor Luís Carlos Menezes for several useful discussions that improved the development of this work.

I wish to thank the Department of Computing and Systems, its professors and the technical staff, for supporting my research. In particular, I wish to thank Ana Georgina, for her great dedication, patience and competence. In fact, she worked as a nanny for the Masters students.

Special thanks to my grandfather Walter Mostaert, he was supported me in all the ways he could and I needed. In fact, he is like a father to me.

I am deeply grateful to my girlfriend Natália Prado for her encouragement, patience, and love during this work.

My gratitude to my mother-in-law Socorro Prado and father-in-law Josoé Matias for receiving me in their home with love and respect for almost all weekends.

Most importantly, I thank God for giving me strength to keep going even with all difficulties posed on my life.

The work described in this dissertation was financially supported by CAPES (Brazilian Federal Agency for Postgraduate Education).

Aos meus pais.

Abstract

The Java Modeling Language (JML) is a formal behavioral interface specification language (BISL) designed for Java. It was developed with the aim of improving the functional software correctness of Java applications. JML has a rich set of features for specifying Java applications, including abstract specifications, method and type specifications, and multiple inheritance specifications. The current JML compiler (jmlc) does not work properly when applied to Java dialects such as Java ME. The instrumented object program generated by the original JML compiler uses the Java reflection mechanism and data structures not supported by Java ME platform. In order to overcome this limitation, our new JML compiler — ajmlc (AspectJ JML Compiler) — uses AspectJ to instrument Java code with JML predicates. A set of translation rules are defined from JML predicates into AspectJ program code. Such rules avoid AspectJ constructs not supported by Java ME. The result is a code compliant with both Java SE and Java ME applications. The translation rules handle a number of JML specifications, such as pre-, postconditions, and invariants. The work includes proofs of concept to compare the size of the final code generated by our compiler with the code size produced by the jmlc compiler. The results indicate that the overhead in code size produced by our compiler is very small when using the abc AspectJ weaver. Such results are essential when Java ME applications are considered. Finally, the proofs of concept validate our compiler when applied to Java SE as well as Java ME applications.

Keywords: design by contract, JML language, JML compiler, aspect-oriented programming, AspectJ language, AspectJ weaving

Resumo

Java Modeling Language (JML) é uma linguagem formal para a especificação comportamental de programas Java. Ela foi desenvolvida com o objetivo de melhorar a corretude funcional de aplicações Java. Para esta finalidade, JML possui um conjunto significativo de características para especificar aplicações Java, incluindo especificações abstratas, especificações de métodos e tipos, além de especificações para herança múltipla. O compilador atual de JML (jmlc) não funciona adequadamente quando aplicado a dialetos Java como Java ME. O programa objeto instrumentado gerado pelo compilador original de JML faz uso de reflexão Java e estruturas de dados não compatíveis com a plataforma Java ME. A fim de superar esta limitação, nosso novo compilador JML, conhecido como ajmlc (AspectJ JML Compiler) faz uso de AspectJ para instrumentar o código Java com predicados JML. Foi definido um conjunto de regras de tradução de predicados JML para código (construções) em AspectJ. Tais regras evitam construções de AspectJ não suportadas pela plataforma Java ME. O resultado é um código compatível com ambas as plataformas, Java SE e Java ME. As regras de tradução lidam com vários tipos de especificações JML, como pré-, pós-condições e invariantes. O trabalho inclui provas de conceito para comparar o tamanho do código final gerado pelo nosso compilador com o tamanho do código produzido pelo compilador JML. Os resultados indicam que o *overhead* no tamanho do código produzido pelo nosso compilador é muito menor quando utilizando o abc *weaver* de AspectJ. Tais resultados são essenciais quando aplicações Java ME são consideradas. Finalmente, as provas de conceito validam a utilização do nosso compilador para aplicações Java SE assim como aplicações em Java ME.

Palavras-chave: projeto por contrato, linguagem JML, compilador JML, programação orientada a aspectos, linguagem AspectJ, recomposição de AspectJ

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software engineering is the process (application) of a systematic, disciplined approach to develop (implement) a software that works as it supposed to do [69]. Software engineering seeks the production of quality software. This way the correctness is a prime software quality — correctness is the ability of software products to perform their tasks as defined by their specification [53]. The commitment to develop a correct software is necessary in order to increase its reliability and reuse. However, after the software is delivered the maintenance is usually performed to fix problems (bugs). This kind of maintenance is known as *corrective maintenance* [69]. Thus, maintenance activities usually have the highest costs [18]. According to Meyer [53], maintenance is estimated as 70% of a software cost as a whole.

Design by Contract (DBC), originally conceived by Meyer [51], is a technique for developing and improving functional software correctness. This technique is based on "contracts" — a formal agreement between a client and its suppliers. To call a method, a client class must satisfy the conditions imposed by a supplier class. If these conditions are satisfied, the supplier class must guarantee certain properties, which constitute the supplier class obligations. On the other hand, if conditions are not satisfied, that is, there is a contract violation, then a runtime error occurs. Thus, the DBC technique is a means to reduce the problems related to software engineering in developing reliable software. As a consequence, the DBC technique helps to reduce corrective maintenance effort.

Several specification languages have been designed to annotate source code written in programming languages [21, 7, 39]. Most of these languages adopt the Design by Contract (DBC) [51] technique. This is the case of the Java Modeling Language (JML) [39].

1.1 A bird's-eye view of JML

The Java programming language does not have built-in support for Design by Contract (DBC). The Java Modeling Language (JML) [39] is a behavioral interface specification language (BISL) for Java — JML adopts the practicality of DBC language to specify the behavior of Java classes and interfaces.

JML introduces a number of constructs for declaratively specify behavior. JML annotations specify the expected behavior implemented by Java code. In other words,

```
public class JMLExample {
  //@ requires b > 0;
  public int div(int a, int b) {
    return a/b;
  }
}
```

Figure 1.1: Example of JML specification.

annotations specify what should happen at runtime. For example the behavior of a method describes what should happen when the method is called. To this end, JML specifications are composed of pre-, postconditions, and invariant predicates based on Hoare-style [31, 32]. JML specifications are annotated in Java code in the form of comments. Figure 1.1 presents code example of a JML specification with a precondition clause, requiring `b > 0`. This precondition states that the input parameter `b` must be grater than zero.

The benefits of adding JML specifications (annotations) to your Java source code include the following [74]:

- more precise description of what the code should do;
- efficient discovery and correction of bugs;
- early discovery of incorrect client usage of classes;
- reduced chance of introducing bugs as the application evolves;
- precise documentation that is always in accordance with application code.

There are a number of tools that give support for JML specifications, including a JML type checker, and a JML assertion checking compiler known as jmlc [14]. The JML compiler is responsible for translating JML-annotated Java source code and generating instrumented bytecode with automatic runtime checks. Code compiled with the JML compiler will check at runtime the assertions that describe a software contract, raising a JML exception when a condition (e.g, precondition) does not hold.

1.2 Motivation

As mentioned above, software engineering seeks the production of quality software; thus by writing formal specifications for program modules (e.g., classes and interfaces), we can improve the desired quality of software. In this way, Java Modeling Language (JML) is a formal specification language that provides means to increase such desired quality for Java applications.

1.2.1 Limitations of JML when enforcing contracts

Java ME [63] is intended for devices with limited resources such as handheld mobile devices. Many existing Java Standard Edition (Java SE) applications can be used in

Java ME applications, but this code usually is not scaled down to fit limited hardware. Additionally, the Java ME API is a subset of the Java SE API geared toward handheld devices.

Similarly to Java SE applications, Java ME applications could be annotated with JML. But, we have identified the following limitations of the JML compiler when applied to Java ME applications:

- it adopts Java reflection [54] to implement specification inheritance and separate compilation. However, Java ME does not support reflection;
- it employs data structures, such as *HashSet* and *Map* [54], both from the `java.util` package, which are not supported by Java ME;
- the final bytecode size is quite bigger than a pure Java ME application.

Besides these limitations imposed to other Java dialects such as Java ME, the infrastructure of the JML compiler (jmlc) translates contracts into runtime checking code in a *non-modularized way*. That is, the generation code of the JML compiler provides an intermediate code completely tangled (intermixed/clutted) with the runtime checks. This leads to problematic modifications, extensions, and optimizations in the standard solution, for example, to adapt the JML compiler to deal with Java ME applications and extend it to implement new constructs (features).

1.2.2 Aspect-oriented programming to implement JML contracts

The implementation of runtime contract enforcement (e.g., preconditions, postconditions, and invariants), as done in JML, can be categorized as a crosscutting concern [19, 35, 37, 50, 49] — Contracts are systematically spread throughout the code, and their evaluation also crosscuts various modules and tangles (mixes) with the application code. It is important to notice that this happens in an intermediate level of the JML compiler (code generation). Regarding this scenario in JML, we need a strategy to answer four questions:

1. how to implement contracts conveniently?
2. how to provide a better code instrumentation?
3. how to check contracts during runtime?
4. how to modify properly the code generation of the JML compiler to support runtime checking of contracts in Java ME applications?

In order to answer the first three questions, this dissertation uses aspect-oriented programming (AOP) to separate the contract enforcement concern (the generated runtime checks provided by the JML compiler) from application code. In particular, AspectJ [36], a general-purpose aspect-oriented extension to Java, is used to provide an aspect-oriented implementation of the JML contracts. The use of AspectJ aspects has several advantages: (1) allow better modularity (clear separation of concern related to contract enforcement); (2) ease of modification/extension to treat other features; (3) better understandability (since the runtime checks are separated from the source code). Moreover, by using AspectJ, the problem of code instrumentation is automatic solved in a standard technology (AspectJ weaving). Code instrumentation involves many issues such as the order of execution, dynamic dispatching, object initialization, inheritance, and so forth. In this way, we leave the responsibility of instrumentation for AspectJ (which knows how to do it better). By using AspectJ, the code instrumentation process occurs at bytecode level (avoid polluting the source code and facilitate modifications and possible extensions). Finally, enforcing JML contracts with AspectJ provides all necessary information for blame assigment and error reporting (as done in the standard JML). This capability is very useful during testing to detect failures and help to locating faults (debugging).

Concerning the last question, by carefully using AspectJ constructs and features, we can generate code compliant to both Java SE as well as Java ME applications. As a consequence, we can verify JML contracts in Java ME applications.

1.3 Objectives

The main goal of this dissertation is to implement a new JML compiler that generates bytecode compliant with Java SE as well as Java ME applications. In this way, we can annotate Java ME applications with JML. Our compiler uses Aspect Oriented Programming to implement JML contracts. The proposed compiler produces AspectJ program code. We carefully use AspectJ constructs that are compliant with both Java SE and Java ME platforms. By using aspect-oriented programming (AOP) we take advantage of the weaver technology. Since we work in the Aspect level, gains obtained through weaver optimizations are automatically transferred to our compiler.

Our compiler deals with the following JML levels [1] [43, Section 2.9]:

Level 0 requires, ensures, signals, assignable, invariant, not_specified, also, old, instance, result, nothing, everything, spec_public, spec_protected, behavior, normal_behavior, exceptional_behavior;

Level 1 pure, static;

Level 2 only_assigned.

Another aim is to develop a proof of concept to investigate that our proposed compiler really generates code that is compatible with Java SE and Java ME. In addition, we have to take into account the final bytecode size we obtain after the AspectJ weaving process so that applications can be embedded in devices such as a handheld.

To the best of our knowledge, ajmlc is the first JML compiler with support for Java SE and Java ME applications that adopts aspect-oriented programming to implement

[1]JML was designed to be used with a variety of different tools, but the evolution of the JML language means that some features are not completely documented or implemented. In this way, some tools that work with JML have features that are not supported by other tools. Consequently, the research groups (`http://www.eecs.ucf.edu/~leavens/JML/`) working on JML decided to define several language levels. The aiming to define these language levels is to make it easier to learn and use it with various tools that work with JML

JML contracts. The Jose tool [24] adopts a different specification language to write contracts in Java. Similar to ajmlc, Jose uses AspectJ to instrument Java contracts, but Jose is not compliant with Java ME applications. Moreover, the semantics for contracts in Jose is different from that of JML. The language JCML [59] is a subset of the JML language targeting Java Card applications. Differently from our compiler, the JCML compiler does not use aspect-oriented programming.

1.4 Contributions of the Dissertation

The main contribution of this dissertation is synthesized bellow:

- novel JML compiler compliant with Java ME and Java SE applications;
- usage of aspect-oriented programming to implement JML contracts;
- proof of concept to investigate our proposed approach with Java SE and Java ME applications and to contrast with the original JML compiler proposed by Cheon [14].

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides the background related to this dissertation. In Chapter 3, we present the new JML compiler, which we call ajmlc (AspectJ JML Compiler), including the translation rules from JML predicates into AspectJ. In Chapter 4, we investigate the overhead in code size produced by our compiler through a proof of concept. We compare our work with others present in the literature in Chapter 5. Finally, in Chapter 6, we present concluding remarks and point out future works.

Chapter 2

Concepts

This chapter presents the main concepts used in this dissertation. We briefly present JML, AspectJ, and Java ME platform.

2.1 An Overview of JML

This section gives an overview of the JML language (Java Modeling Language) [43, 39], introducing its major features such as method and type specifications.

2.1.1 Behavioral Interface Specification

JML [43, 39], which stands for "Java Modeling Language", is a *behavioral interface specification language* (BISL) tailored to Java [27]. JML combines the design by contract (DBC) technique [51, 53] of Eiffel [52] with the model-based specification approach typified by VDM [34], Z [70], and Larch/C++ [40]. It also adds some elements from the refinement calculus [6]. As in Eiffel, JML uses Java's expression syntax in assertions, which makes the language easy to learn and use when compared with other specification languages such as Z [70].

As a BISL, JML is used to specify Java modules (classes and interfaces). It concentrates on two aspects of a Java module during the specification process:

- syntactic interface — consists of the names and static information (e.g., method names, modifiers, arguments, return type) found in Java declarations;
- functional behavior — describes how the module works when used.

Thus, BISLs are languages that describe interface details for clients. For example, Larch [29] describes how to use a module in a C++ program, just as JML specifies how to use a module in a Java program.

2.1.2 Annotations

JML specifications are written in special *annotation comments*, which start with an `@` sign, that is, comments in the form: `//@` <JML specification> or `/*@` <JML specification> `@*/`. These annotations work as simple comments for a Java compiler, whereas they are interpreted as specifications by the JML compiler [14].

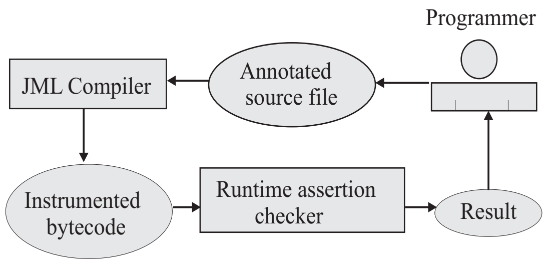


Figure 2.1: An overview of the JML environment.

It is important to note that the at-sign (`@`) must be right next to the start of comment characters. A comment starting with `// @` will be ignored by JML. In other words, such a comment is not processed as a specification by the JML compiler. This happens because JML tools do not currently warn the programmer about comments that use such mistaken annotation markers.

Figure 2.1 depicts an overview of the JML environment. A programmer includes annotations in the Java source file in the form of comments. Then, the JML compiler translates the annotated Java source file into instrumented bytecode that check whether the Java program respects the specification.

2.1.3 Assertions and Expressions

Assertions and expressions of specifications in JML are written using Java's expression syntax. However, they must be *pure*. This means that side-effects cannot appear in JML assertions or expressions [41]. But Java assertions and expressions do allow side-effects. Regarding the prevention of side-effects, the following Java operators are not allowed within JML specifications:

- assignment — assignment operators (such as `=`, `+=`, `-=`) are not allowed;
- increment and decrement operators — all forms of increment and decrement operators (`++` and `--`) are not allowed.

In addition, only pure methods can be used in JML expressions and assertions — a method is pure if it does not have any side-effects on the program state. In other words, the method does not modify the state (e.g., by assigning any fields of objects). The pureness of expressions is specified in JML by using the `assignable` clause [58]. Only the fields listed in an `assignable` clause can be modified by a method. Two JML keywords can be used with the `assignable` clause: `\nothing` and `\everything`. We can indicate that a method does not have any side effects by writing `assignable nothing`. On the other hand, we can say that a method can modify anything by writing `assignable \everything`. Thus, Figure 2.2 illustrates a JML specification that the method `getWeight` cannot modify any fields of objects that are visible outside the method and that exist before the method have started its execution. However, the method may still modify its local variables. Another way to indicate that a method has only pure expressions is by using the JML modifier `pure` when declaring a method.

```
public /*@ pure @*/ int getWeight(){ return weight; }
```

```
//@ assignable \nothing;
public int getWeight(){
  return weight;
}
```

Figure 2.2: JML specification with an assignable clause.

This method declaration denotes a method with no side-effects; it is not allowed to modify the program state. The use of the JML modifier `pure` is equivalent to the `assignable \nothing` clause.

Besides `assignable`, JML provides a rich set of constructs, some of which make extensions to the Java's expression syntax to provide more expressive power in JML specification; they can be used in JML assertions and expressions. For example, `\old`(E) represents the pre-state value of expression E. An expression with a pre-state value refers to the value before method execution. The `\result` construct specifies the return value of a method. Note that in JML assertions, such constructs start with a backslash (`\`) in order to avoid interfering with identifiers present in a user program. JML also provides the use of logical connectives such as *conjuction* (`&&`), *disjunction* (`||`), *negation* (`!`), *forward* (`==>`) and *reverse implications* (`<==`), *equivalence* (`<==>`), and *inequivalence* (`<=!=>`). Regarding quantifiers, JML supports several kinds such as *universal quantifier* (`\forall`), *existential quantifier* (`\exists`), and *generalized quantifiers* (`\sum`, `\product`, `\min`, and `\max`). The quantifiers `\sum`, `\product`, `\min`, and `\max` are generalized quantifiers that return respectively the sum, product, minimum, maximum of the values present in JML expressions. For example, an expression (`\sum int x; 1 <= x && x <= 5; x`) denotes the sum of values from 1 to 5.

Another feature provided by JML is that one can use *informal descriptions* when specifying a Java module. Informal descriptions are useful for producing an informal documentation of the Java code. JML also allows informal descriptions when specifying a Java module.

```
(* some text describing a boolean-valued predicate *)
```

The interpretation for this informal description is entirely determined by the reader. They have boolean type. Their meaning is either `true` or `false`. Since informal descriptions are not-executable, they may be treated differently by different tools [11] in different situations.

In-line assertions

JML provides the use of a specific kind of assertion known as *in-line* assertions (also called *intracondition*). These assertions can be specified in the method body. In other words, they are interwoven with Java code. Figure 2.3 shows an example of a JML specification with in-line assertion (`assert y1 != 0 && y2 != 0`). Here, when the execution of method `m` reaches the assertion, the expression (`y1 != 0 && y2 != 0`) must be satisfied. Otherwise a `JMLAssertError` is raised signaling the assertion violation. JML provides several kinds of in-line assertions, such as `assert` statements, `assume` statements, `henceby` statements, `unreachable` statements, `set` statements, and

```
public class InLineAssertion {
  public void m() {
    //@ assert y1 != 0 && y2 != 0;
    y1 ++;
    y2 ++;
  }
}
```

Figure 2.3: JML specification with in-line assertion.

`loopinvariant` and `variant` statements. Additional information about the kinds of in-line assertions and their implementation can be found in Cheon's work [14, Section 4.7].

Semantic differences between JML and Java expressions

There are a number of differences between JML and Java expressions [14, Section 3.1]. We can point out two of them:

- abnormal termination — the evaluation of an expression in Java may complete *abnormally* by throwing an exception. This abnormal termination always has an associated reason (i.e., `throw`). However, in JML, the semantics consists in replacing an expression that throws an exception by a value (e.g., `true` or `false`);

- order of evaluation — differently from JML, the Java operators are order-sensitive. For instance, consider expressions presented in Table 2.1, where `x` is an array type. In Java, if the subexpression (`true`) comes first, the result will always be `true` and the second subexpression is never evaluated because of the short-circuit evaluation; otherwise, if the subexpression (`x.length > 0`) comes first, we have two possible results: (1) if `x` is not null, the result is `true`; (2) if `x` turns out to be null, then `NullPointerException` is throw. This means that, Java evaluates the right-hand operand only if the left-hand operand is not conclusive. On the other hand, if the whole expression completes normally, the order of evaluation is not relevant for the final result. In contrast, with JML expressions, the result is always `true`. This happens because JML expressions obey the standard rules of logic even in the presence of *undefinedness* [14, Section 3.2] [16].

	`true || x.length > 0`	`x.length > 0 || true`
Java	Always `true`	if `x` is not null: `true` Otherwise: `NullPointerException`
JML	Always `true`	if `x` is null: `true` Otherwise: `true`

Table 2.1: Example of semantic difference between Java and JML [68].

Null is not the default

Null pointer exceptions are among the most common faults raised by components written in object-oriented (OO) languages such as Java [27]. For example, if x is null then x.f and x.m() both result in a `NullPointerException` [43, Section 2.8]. Such a problem causes undefinedness [14, Section 3.2] [16] in expression evaluation. One can prevent this problem by declaring every reference type field as `nullable`. However, any declaration (except for local variables) of a reference type is implicitly declared non-null (has an implicit `non_null annotation`), unless *explicitly* declared `nullable`. A declaration can be explicitly declared nullable using the `nullable` annotation. Moreover, a class or interface that implicitly contains nullable declarations is specified by using the `nullable_by_default` class annotation.

```
public class Person {
  public String name;
  public Address address;
}
```

```
public class Person {
  public String name;
  public Address address;

  /*@ invariant name != null
    @ && address != null @*/;

  // ...
}
```

Figure 2.4: Desugaring non-null annotations for field types in JML specification.

JML treats the implicitly declaration `non_null` as an invariant (see Section 2.1.5). In Figure 2.4 the reference type fields (`name` and `address`) on the left-hand side are desugared by the JML compiler into explicitly non-null `invariant` annotations on the right-hand side. Thus, after the constructor execution, if any reference type declaration is null, a `JMLInvariantError` is raised (see Section 2.1.4).

Early versions of the JML compiler were originally implemented with implicity null default semantics. Nevertheless, Chalin [13] stated that programmers want more than 50% of declarations of reference types to be non-null. Thus, with the old JML semantics, programmers must add `non_null` annotations to the majority of declarations. As a result, programmers usually forget to add the `non_null` annotation to some fields, leading clients to call methods with null arguments resulting in `NullPointerException`. Therefore, based on this study, Chalin [13] proposed to modify the JML semantics by allowing declarations to be non-null by default. Thus, the JML tools [11] were enhanced with this new semantics. Like JML, other languages [52, 8] have type systems that allow the addition of types that are not-null by default.

2.1.4 Method Specifications

In JML, method specifications contain pre-, postcondition predicates based on Hoare-style [31, 32], but with many extensions (see Section 2.1.3). It also has a number of improvements, including heavyweight and lightweight specifications, normal and exceptional postconditions, and frame conditions. In the following, these and other features are briefly described.

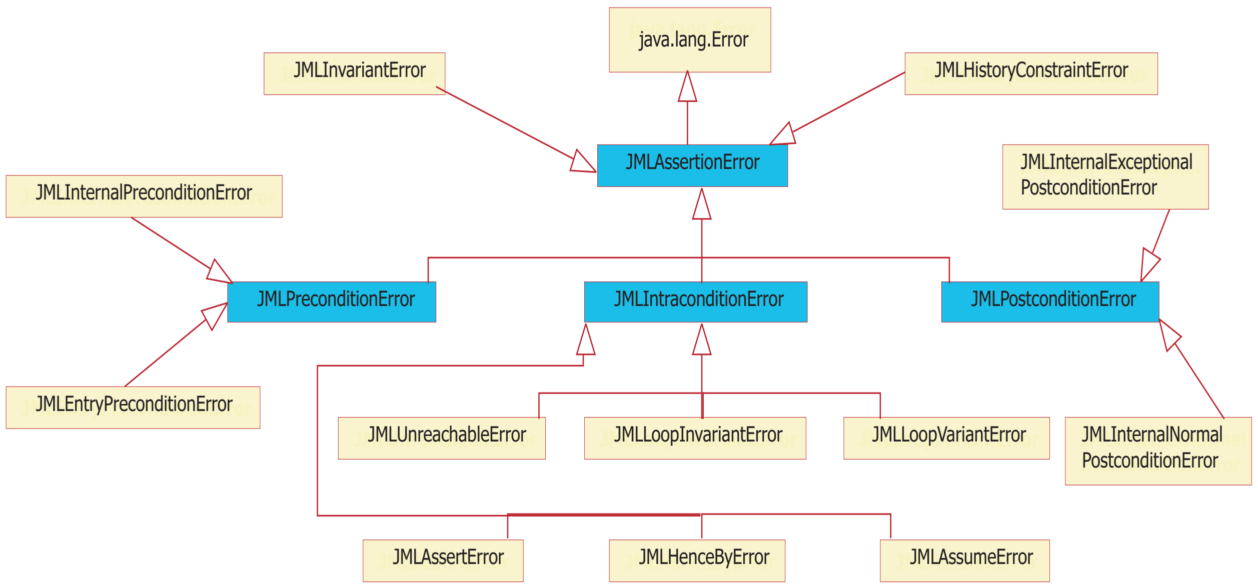


Figure 2.5: JML class hierarchy of assertion violation errors.

States of method specifications

JML constructs for method specifications are divided into three groups of states [14, Section 4.1]:

- pre-state specifications — these specifications must be evaluated in the pre-state, immediately before the execution of the method body. Example of these specifications are: `requires` clauses, `old` variables and `old` expressions. It is important to note that expressions such as `old` appear in post-state specifications, but they must be evaluated in the pre-state (before method execution);

- post-state specifications — these specifications must be evaluated in post-state, immediately after the execution of the method body. Clauses such as `ensures`, and `signals` are example of post-state specifications;

- internal-state specifications — these specifications must be evaluated in internal states, during the evaluation of the method body. In-line assertions (see Section 2.1.3 [In-line assertions]) are examples of internal specification state.

Reporting assertion violations

When any assertion violation occurs, JML must inform the user about it. After instrumentation by the JML compiler (jmlc), the JML runtime assertion checker (jmlrac) [11] is responsible for the runtime check of assertions. There are several ways to report assertion violations, such as throwing exceptions, printing error messages, halting the program, and logging. The JML compiler instruments its assertions to throw exceptions. Once instrumented, the runtime assertion checker (jmlrac) is the tool that throws the exceptions during runtime. Moreover, an assertion violation should be reported in such a way that one can identify its cause. When any violation occurs, the JML runtime assertion checker also exposes static information about the problem. For example, the location of the violation, which is useful to determine its cause.

`requires`	preconditions
`ensures`	normal postconditions
`signals`	exceptional postconditions
`assignable`	frame conditions

Table 2.2: JML clauses for method specifications.

```
//@ requires a >= 0 && b >= 0;
public int sum(int a, int b) {
  return a + b;
}
```

Figure 2.6: Example of JML precondition specification.

In JML, an assertion violation is a situation from which programs cannot recover. Hence, assertion violations are defined as error classes. Figure 2.5 shows the assertion violation errors hierarchy of JML. The abstract class `JMLAssertionError` is a subclass of class `java.lang.Error` [64], which is the top-level superclass of all assertion violation errors. This hierarchy contains error classes for different kinds of assertion violations (e.g., precondition violations). For example, the class `JMLIntraconditionError` is the superclass for reporting assertion violations of in-line assertions such as `assert` (`JMLAssertError`), and `assume` (`JMLAssumeError`) statements (see Section 2.1.3 [In-line assertions] for in-line assertions). A postcondition violation, `JMLPostconditionError`, is further distinguished into two types: a normal postcondition violation (`JMLInternal-NormalPostconditionError`), and an exceptional postcondition violation (`JMLInternal-ExceptionalPostconditionError`). The former refers to a violation of properties (predicates) after the method execution (normal termination). The latter refers to a violation of properties when the method throws an exception (abnormal termination). The classes `JMLInvariantError` and `JMLHistoryConstraintError` are responsible for reporting assertion violations of type invariants and history constraints, respectively (see Section 2.1.5 for type specifications).

Specification clauses

JML provides a number of specification clauses that can be used to specify the behavior of methods (see Table 2.2): the `requires` clauses specifies *preconditions*; the `assignable` clauses [58] specifies *frame conditions* (see Section 2.1.3); the `ensures` clauses specifies *normal postconditions*, whereas `signals` clauses specifies *exceptional postconditions*.

Preconditions are predicates that must hold before method execution. Figure 2.6 shows an example of a precondition of the method `sum`. According to this specification, the arguments `a` and `b` of the method `sum` must be non-negatives; otherwise a `JMLInternalPreconditionError` is raised signaling an assertion violation error. The jmlc translation rules for preconditions are presented in [14, Sections 3.2.4 and 4.4].

Normal postconditions are predicates that must hold after method execution without throwing any exception. Figure 2.7 shows an example of a normal postcondition of the the method `sum`. According to this specification, the result of the method (indicated by

```
//@ ensures \resut == a + b;
public int sum(int a, int b) {
  return a + b;
}
```

Figure 2.7: Example of JML normal postcondition specification.

```
//@ signals (Exception) a >= 0 && b > = 0;
public int sum(int a, int b) {
  // Throws a java.lang.Exception
  return a + b;
}
```

Figure 2.8: Example of JML exceptional postcondition specification.

the `\resut` construct) must be equal to the sum of the arguments `a` and `b`; otherwise a `JMLInternalNormalPostcondition` is raised, signaling the assertion violation error. The jmlc translation rules for normal postconditions are presented in [14, Sections 3.2.4 and 4.5.1].

Exceptional postconditions are predicates that must hold when the method terminates by throwing an exception. Each exceptional postcondition can consist of several signals clauses. In this way, each `signals` clause must hold when the specified method terminates abnormally by throwing an exception of a type specified in the `signals` clause. Figure 2.8 shows an example of an exceptional postcondition for the method `sum`. According to this specification, when the method `sum` throws an exception, the arguments `a` and `b` must be non-negatives; otherwise the original exception is intercepted and a `JMLInternalExcetionalPostconditionError` is raised signaling the assertion violation error. The jmlc translation rules for exceptional postconditions are presented in [14, Sections 3.2.4 and 4.5.2].

Heavyweight and Lightweight specifications

In JML, one can use two different styles on method specifications: heavyweight specifications, and lightweight specifications [43]. A heavyweight specification can start with the keyword `behavior`, which indicates a "complete" specification that includes both normal and abnormal situations. Figure 2.9 shows an example of a `behavior` specification, including normal (lines `2` to `4`), and abnormal situations (line `5`).

The `normal_behavior` keyword is used only to define specifications that include only normal situations, that is, no method can terminate abnormally by throwing an exception — the `signals` clause cannot appear in these specifications cases. Figure 2.10 (lines `1` to `3`) shows an example of normal behavior specification. Unlike the `normal_behavior`, the `exceptional_behavior` is employed to specify abnormal situations using the `signals` clause. Such specifications cannot terminate normally. This behavior is emulated by introducing an implicit `ensures false`. Hence, if no exception is thrown during the method execution, then the method terminates by throwing a `JMLInternalPostconditionError`.

Figure 2.11 shows an example of an `exceptional_behavior` specification. When an

```
1  /*@ behaviour
2    @ requires ( a >= 0 ) && ( b >= 0 );
3    @ assignable \nothing;
4    @ ensures \result == a + b;
5    @ signals (Exception) a > 200 && b > 100;
6    @*/
7  public int soma( int a, int b ){
8    return a + b;
9  }
```

Figure 2.9: Example of behavior specification.

```
1  /*@ normal_behaviour
2    @ ensures \result == a + b;
3    @*/
4  public int soma( int a, int b ){
5    return a + b;
6  }
```

Figure 2.10: Example of normal_behavior specification.

exception is thrown, both input parameters (`a` and `b`) must be grater than zero (lines `1` to `3`).

A heavyweight specification has a well-defined interpretation for each clause in the specification. When one omits a particular clause, JML assumes that it is interpreted as `true`. On the other hand, a lightweight specification does not start with a `behavior` keyword. Thus, only the clauses of interest are specified. In other words, it defines an "incomplete" specification. For example, a method can specify its normal behavior (by defining only precondition or normal postcondition clauses), or its exceptional behavior (by defining only exceptional postcondition clauses). In lightweight specifications, when a particular clause is omitted, the `\not_specified` interpretation is assumed. The JML semantics states that the keyword `\not_specified` is used to denote that a particular omitted clause has no condition. Regarding implementation details [14], the JML compiler interprets omitted clauses in lightweight specifications similar to ones in heavyweight specifications which means `true`.

Old expressions

In JML, the keyword `old` is employed to refer to pre-state expressions and variables in post-state expressions (e.g., normal postconditions). An old expression, (`\old(`E`)`), denotes the value of the expression E in the pre-state, whereas an old variable, (`\old(`v`)`), denotes the value of the variable v in the pre-state.

Static blocks

Constrained methods are checked when they are called, but if the call is made inside a static block the properties of the methods (e.g., preconditions) are not verified. This

```
/*@ exceptional_behaviour
  @ signals (Exception) a > 0 && b > 0;
  @*/
public int soma( int a, int b ){
  return a + b;
}
```

Figure 2.11: Example of exceptional_behavior specification.

happens because the current JML compiler does not generate instrumented bytecode able to verify constrained methods into static blocks.

```
//@ requires x > 0;
public static void m(int x) { /* ... */ }

static{
  m(-3); // this call leads to no assertion violation
}
```

Figure 2.12: Example of non checked constrained method when it is called.

Figure 2.12 shows an example in which there is a method declaration, and a method call. The constrained method `m` is defined with the precondition `x > 0`, and the call `m(-3)` violates the constrained method is made inside a static block. As a result, no assertion violation is raised. This lack of homogeneity can be problematic and might lead to undesired behavior.

2.1.5 Type Specifications

In JML, *type specifications* are used to refer to both class and interface specifications. In addition, there are other type specifications in Java member declarations that can have JML specifications such as invariants, and history constraints. These type specifications are briefly described in what follows.

Invariants

Invariants are predicates that must hold in all visible (reachable) states[1] [57, Definition 2.1]. An invariant is annotated in a type declaration by using the keyword `invariant`, also known as *type invariant*. Figure 2.13 shows an example of invariant. In this example, the invariant (`weight >= 0.0`) must be preserved in all visible states. In this way, if a call to the methods `setWeight or getWeight` violates the invariant, a `JMLInvariantError` is raised.

JML uses the keyword `helper` to provide more flexibility for invariants. The helper modifier [43, Section 7.1.1.4] can only be used on a private method or private constructor. By using this modifier, such methods and constructors are also called *helper methods*

[1]Visible states are all the possible states in the program execution.

```
public class Person {
  public double weight;
  //@ invariant weight >= 0.0;

  public void setWeight(double weight) {
    this.weight = weight;
  }

  public double getWeight() {
    return this.weight;
  }
}
```

Figure 2.13: JML invariant specification.

and *helper constructors*. In this way, helper methods and constructors are "free" from the obligation of preserving type assertions (e.g., invariants). Suppose that the helper method `m` is part of the class `Person` (see Figure 2.13).

```
private /*@ helper @*/ void m (){ this.weight = -10; }
```

Hence, `m` needs not satisfying the invariant (`weight >= 0.0`), since it is declared as helper. The example presented in Figure 2.13 refers only to instance invariants. In JML, the invariants are distinguished into static and instance invariants. The former refers only to static fields and methods. On the other hand, the latter can refer to both static and instance fields and methods.

The following states of the program execution must be preserved by an object in order to keep an instance invariant integrity:

- after execution of all non-helper constructors;
- before execution of a non-helper finalizer method;
- before and after execution of all non-helper non-static non-finalizer methods.

On the other hand, instance invariants need not be preserved in the following states: (1) after execution of constructors declared with the `helper` modifier; (2) before and after execution of methods declared with the `helper` modifier.

The following states must be preserved by a type[2] in order to keep the static invariant integrity:

- before and after executions all non-helper constructors;
- before and after executions of all non-helper static methods;
- before and after executions of all non-helper non-static methods.

[2]A static invariant that refers to a static state must be preserved by a type T, whereas an instance invariant that refers to an instance state must be preserved by an object O that has type T.

```
 1  public class TemporarilyInvariantBreak {
 2    public int f;
 3    //@ invariant f >= 0;
 4
 5    public void m() {
 6      this.f = -10; // temporarily invariant break!
 7      System.out.println("internal-state");
 8      System.out.println("f:"+f);
 9      this.f = 0; // re-establishment of the invariant
10    }
11  }
```

Figure 2.14: Example of temporary invariant violation in a JML specification.

An invariant may be explicitly declared to be a static one or an instance one by using the modifiers `static` or `instance` during the declaration of the invariant. If an invariant is declared in a class without the modifier, it is an instance invariant by default, whereas if an invariant is declared in an interface without the modifier, it is an static invariant by default.

Concerning invariant violations (for static and instance invariants), in JML, there is only one way to temporarily break an invariant and consequently throws no assertion violation error. This temporarily invariant break is established in an *internal-state* during evaluation of a method's body (see Section 2.1.4). However, before the method returns to the caller, the invariant must be reestablished, that is, a method during the execution of its body can break an invariant many times, but before the end of its execution all invariants must be reestablished, otherwise a `JMLInvariantError` is raised indicating the invariant assertion violation. The example in Figure 2.14 demonstrates the temporarily violation of an invariant assertion even in presence of *callbacks* (recursive assertion checking). In this example the class `TemporarilyInvariantBreak` has an invariant condition (line `3`) that refers to the field `f` (line `2`). For the invariant break purpose, the class `TemporaryInvariantBreak` has a method `m` (lines `5` to `10`) that temporarily breaks the invariant declared in line `3`. The invariant break occurs in line `6` of the method `m`. Once broken, one can use the values of the internal-state (line `8` is printed the value of field f). However, before the method terminates its execution, the invariant is established again (line `9`) in order to prevent invariant assertion violation.

Regarding invariants and method termination, methods and constructors must preserve invariants in the case of normal and abnormal terminations. For example, even if a method terminates abnormally by throwing an exception, the invariant check must be performed.

An invariant can be declared with any Java's access modifiers (`private`, `protected`, and `public`). As in class members, if an invariant is declared in a class with no one of the Java access modifiers, this means that it has a package visibility (default access). An invariant declared in an interface with no modifiers has public visibility. For instance, by writing an invariant declaration with public visibility.

```
//@ public invariant (/*invariant condition*/);
```

The jmlc compiler translation rules for invariant instrumentation are presented in [14, Section 5.2].

History constraints

History constraints, known as *constraints* for short, are used to specify the way that objects can change their values from one state to another (i.e., pre-state to a post-state) during the program execution. JML took constraints from Liskov and Wing's work [46]. In JML, a constraint is written in a constraint clause with the keyword `constraint`. Moreover, `constraint` clauses are usually written using old expressions to relate the previous state (pre-state) with the resultant state (post-state), because they are *two-state predicate*. On the other hand, invariants (discussed in a previous section) cannot use old expressions, because they are *one-state predicates*. The constraints are applied to methods, and they can be thought as an implicitly clause conjoined to method's post-condition and they must hold in the post-state of every (non-helper) method execution.

```
public class ConstraintSample {
  public int index;
  //@ constraint index == \old(index + 1);

  public void incIndex(){
    index++;
  }

  // ...other methods
}
```

Figure 2.15: Example of constraint in JML specification.

Class `ConstraintSample` in Figure 2.15 includes an example of constraint in JML. The example defines the public field `index` and the method `incIndex`. The constraint (`index == \old(index + 1)`) states that after execution of the method `incIndex` (in a post-state), the specified constraint must be checked. If the constraint does not hold, a `JMLHistoryConstraintError` is raised indicating the constraint assertion violation. A scope problem is observed in the constraint defined in Figure 2.15. Here, the constraint is applied to all methods in class `ConstraintSample`. In this way, we can have calls to other methods in class `ConstraintSample` which do not modify the field `index`, thus leading to a constraint assertion violation (`JMLHistoryConstraintError`). To deal with such a scope problem, JML also allows one to specify constraints that are applicable only to a specific set of methods [43, p.55] [14, p.22].

```
//@ constraint index == \old(index + 1) for incIndex();
```

This rewritten `constraint` clause shows how to use a constraint to constrain only the method `incIndex`. The `for` clause is used to define a set of methods, $(m_1,...,m_n)$ where each m_i with its signature[3] $m(T_1x_1,...,T_nx_n)$, is constrained by the `constraint` clause. On the other hand, if the `for` clause is omitted, it becomes an universal constraint that

[3]The methods includes their types in order to distinguish between overloaded methods.

constrains all methods. The constraint specification in Figure 2.15 is an example of universal constraint.

As with invariants, JML makes a distinction between instance constraints and static constraints. An instance constraint must hold only for instance methods, whereas a static constraint must hold both for instance and static methods. It is important to note that: (1) instance constraints do not apply to constructors and finalizers because there is no well-defined pre-state for constructors and no post-state for finalizers. Moreover, helper methods, like invariants, are free from the obligation of preserving history constraints; and (2) unlike instance constraints, static constraints must be satisfied by all constructors (after constructor execution).

Regarding constraints and method termination, just like invariants, constrained methods must preserve constraints in the case of normal and abnormal terminations. In relation to access modifiers, like invariants, constraints can also be declared with any Java's access modifiers (`private`, `protected`, `and public`). The rules of visibility posed on invariants are the same on constraints.

The jmlc compiler translation rules for history constraint instrumentation are presented in [14, Section 5.3].

Specification for interfaces

As previously mentioned, JML is used to specify Java's modules such as classes and interfaces. Thus, interfaces may have specifications as well. JML specifications present into interfaces provides two semantics differences between JML and Java:

- stateful interfaces — in Java, interfaces are *stateless* in the sense that there are no time-varying fields in interfaces. In JML, however, interfaces become *stateful* as one can declare instance fields into interfaces by using *model fields* (one feature of the model programs [43]). Accordingly to JML, locations should be allocated somewhere for storing state information attributed to the interfaces;

- multiple inheritance — Java allows only single inheritance of code whereas JML supports multiple inheritance of specifications. That is, an interface in JML can have its own (model) fields and (model) methods, and this brings all of the problems associated with multiple inheritance, such as name conflicts [43 cheon reference].

Figure 2.16 shows an example of a interface that contains JML specifications, that is, the interface `ICalc` declares two methods `add` and `sub`. These methods have one `requires` clause and one `ensures` clause each one. A class that implements the interface `ICalc` also inherits its specifications (see Section 2.1.6 for inheritance of specifications).

JML specifications are translated into assertion methods by the JML compiler (jmlc), resulting in pre- and postcondition methods. However, interfaces cannot have assertion methods inserted into it, because in Java an interface must be abstract [28]. Therefore, for interfaces, the JML compiler generates a separate assertion class, known as *surrogate class*, that includes all assertion methods translated from the specifications of the interface.

Figure 2.17 shows an example in a class diagram of a `Surrogate` class used to contain the assertion methods generated by the JML compiler. In this example, `I` is

```
public interface ICalc {

  //@ requires x > 0 && y > 0;
  //@ ensures \result == x + y;z
  public double add(double x, double y);

  //@ requires x > 0 && y > 0;
  //@ ensures \result == x - y;
  public double sub(double x, double y);
}
```

Figure 2.16: Example of interface with JML specifications.

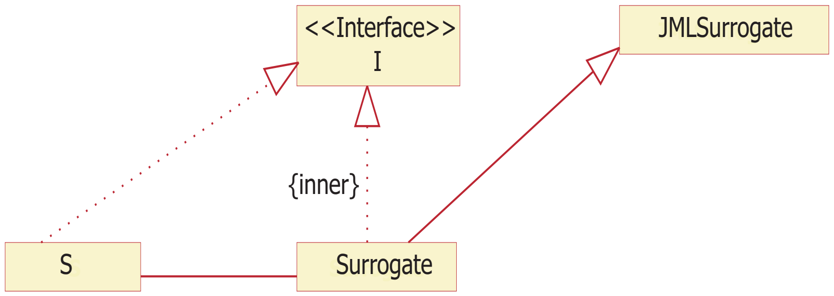


Figure 2.17: Interface and its surrogate class.

an interface (that has JML specifications) with an implementing class `S`. The surrogate class `Surrogate` is a subclass from the class `JMLSurrogate`, which is a JML class that defines common properties to all surrogate classes. This way, each object of the class `S` has an unique surrogate object of the interface `I`. The surrogate object is used by the object of the class `S` to make calls to assertion methods defined in the interface's surrogate class. For more details about the instrumentation of interfaces by the JML compiler, refer to [14, Section 6.5].

Discussion

There are two limitations to the current approach of checking type invariants. The first is related to class initialization. JML semantics states that static invariants should be established after initialization of a class, and they should be preserved by all non-helper constructors and static methods [43]. Nevertheless, when constrained methods are called inside a static block, no checks related to invariants are performed. Figure 2.18 shows an example where the method `m` is constrained with the invariant (`x > 0`) and a call (`m()`) that violates the invariant is made inside a static block. As a result, no assertion violation is raised. Cheon's compiler [14] does not generate instrumented bytecode properly to deal with this limitation. The second limitation occurs when the invariants (static or instance) are declared in an interface. The class that implements such an interface has the obligation to check both static and instance invariants of the interface (accordingly to JML semantics [43]). However, the jmlc [14] compiler generates instrumented code that ignores these invariants (implementation mistaken).

```
public static x;
//@ static invariant x > 0;

public static void m() { x = −3; }

static{
  m(); // this call leads to no assertion violation
}
```

Figure 2.18: Example of non checked type invariant when it is called.

2.1.6 Inheritance of Specifications

In JML, there are several ways to inherit specifications: subclassing, interface extension and implementation, and refinement [39]. A subtype inherits not only fields and methods from its supertypes, but also specifications such as pre- and postconditions, and invariants. To provide the effect of specification inheritance, JML employs the keyword `also`, which denotes a combination (join) of specification cases, which consist of clauses including pre-, postconditions and so forth.

Regarding preconditions and postconditions, Leavens [38, Definition 1] adopts the notation $T \rhd (pre, post)$ to denote an instance method specification written in type T. According to his definition, if $T \rhd (pre, post)$ and $T' \rhd (pre', post')$ are specifications of an instance method m, and U is a subtype of both T and T', then the join of $(pre, post)$ and $(pre', post')$ for U, written $(pre, post) \sqcup^U (pre', post')$, is the specification $U \rhd (p, q)$, where the precondition p and postcondition q are given by Formulas (2.1) and (2.2), respectively [38].

$$pre \ \texttt{||} \ pre' \tag{2.1}$$

$$(\texttt{\textbackslash old}(pre) \texttt{ ==> } post) \ \texttt{\&\&} \ (\texttt{\textbackslash old}(pre') \texttt{ ==> } post') \tag{2.2}$$

This definition states that the preconditions are combined by a disjunction of *pre* and *pre'*. Postconditions are combined by a conjunction of implications such that when one of the preconditions holds in the pre-state, then the corresponding postcondition must hold.

To define invariant, history constraint, and method specification inheritance in a type T [38, Definition 2], Leavens introduces the notation $added\_inv^T$, $added\_hc^T$, and $added\_spec^T_m$, respectively. The inheritance of *pre* and *post* specifications of method m declared in type T is represented as $added\_spec^T_m = (added\_pre^T_m, added\_post^T_m)$. The mechanism to define a specification inheritance can be explained by constructing an extended specification. For invariants and history constraints we have the following: the extended invariant (ext_inv^T) and history constraint (ext_hc^T) of T is the conjunction of all added invariants and history constraints in T and its proper supertypes (indicated by $supers(T)$), see Formulas (2.3) and (2.4).

$$ext\_inv^T = \bigwedge\{added\_inv^U | U \in supers(T)\} \tag{2.3}$$

$$ext\_hc^T = \bigwedge\{added\_hc^U | U \in supers(T)\} \tag{2.4}$$

In the case of method specifications, for all methods m introduced in the proper supertypes of T, the extended specification of m is the join of all added specifications for m in T and all its proper supertypes (see Formula (2.5)).

$$ext\_spec^{T}_{m} = \bigsqcup^{T} \{added\_spec^{U}_{m} | U \in supers(T)\} \tag{2.5}$$

2.1.7 Privacy of Specifications

As in Java, JML provides rules of visibility for JML annotations. It imposes extra rules to the usual Java visibility rules. One rule states that an annotation cannot refer to names (e.g., fields) that are more hidden than the annotation visibility. Suppose x is a name declared in package P with type T ($P.T$), by applying this JML visibility rule, we have the following restrictions:

- An expression in a public method specification can refer to x only if x is declared as public;
- An expression in a protected method specification can refer to x only if x is declared with public or protected visibility, and x must also respect Java's visibility rules — if x has protected visibility, then the reference must occur from within P or outside P only if the reference occurs in a subclass of T;
- An expression in a method with default (package) visibility can refer to x only if x is declared with public, protected or default visibility, and x must also respect Java's visibility rules — if x has default visibility, then the reference must occur from within P;
- An expression in a private method specification can refer to x only if x is declared with public, protected, default or private visibility, and according to Java's visibility rules, if x is private, the reference must occur within $P.T$.

Figure 2.19 shows an example with the use of various privacy level names that are legal and illegal. In this example, the lines `8, 9, 10, 14, 15, 20` present illegal invariant declarations in contrast to legal ones in lines `7, 12, 13, 17, 18, 19, 22, 23, 24, 25`. Both situations are according to JML visibility rules described above. We used invariants to illustrate the privacy levels, but a similar scenario could be illustrated for method specifications, history constraints, and so on.

Note that the visibility access modifiers used in JML only occur in heavyweight method specifications. In a lightweight method specification, the privacy level is the same of the method. For example, a public method with a lightweight method specification is considered to have a public visibility annotation.

Still regarding privacy of specifications, JML also offers the keywords `spec_public` and `spec_protected`. Both are used to provide means to make a declaration that has a narrower visibility become wider (public or protected), and thus, respecting the rules of JML visibility. For instance, the declaration

```
private /*@ spec_public@ */String name;
```

```
public class PrivacyLegalAndIllegalUsing {
  public int pub;
  protected int prot;
  int def; // default (package) visibility
  private int priv;

  //@ public invariant pub > 0; // legal
  //@ public invariant prot > 0; // illegal!
  //@ public invariant def > 0; // illegal!
  //@ public invariant priv < 0; // illegal!

  //@ protected invariant pub > 1; // legal
  //@ protected invariant prot > 1; // legal
  //@ protected invariant def > 1; // illegal!
  //@ protected invariant priv < 1; // illegal!

  //@ invariant pub > 1; // legal
  //@ invariant prot > 1; // legal
  //@ invariant def > 1; // legal
  //@ invariant priv < 1; // illegal!

  //@ private invariant pub > 1; // legal
  //@ private invariant prot > 1; // legal
  //@ private invariant def > 1; // legal
  //@ private invariant priv < 1; // legal
}
```

Figure 2.19: Example of using legal and illegal privacy levels in JML specifications.

introduces a string field `name` that Java considers private but JML considers public. In this way, this declaration can be used, for example, in a public method specification.

The JML visibility rule described above concerns only Java fields — class or interface fields. This rule is described by Leavens and Müller [42, Rule 1]. Besides the visibility rule applicable to Java fields, JML also provides visibility rules to Java method specification cases. These rules are also described by Leavens and Müller [42, Rule 2 and Rule 3]. For example, regarding behavioral subtyping, Rule 3 states that "every overriding method $S.m$ inherits those specification cases of each overridden supertype method $T.m$ that are visible to type S. The implementation of $S.m$ must satisfy the specification cases for m given in S as well as all inherited specification cases". For instance, the method `move` of class `ScreenPoint` (right side) in Figure 2.20 [42] cannot see the private specification case of the overridden `move` method defined in class `Point` (left side). Therefore, the specification case of method `move` in class `ScreenPoint` has not to obey the private specification case of the overridden `move` method, because it is not visible (according to [42, Rule 3]).

The two discussed rules (Rule 1 and Rule 3) and some other covered in [42] explore some parts of JML semantics. However, rules have been not implemented yet. For example, the JML compiler does not enforce the visibility rules which refer to method

```
public class Point {
  protected int _x, _y;
  private int _oldX, _oldY;

  /*@ protected normal_behavior
    @  requires _x + dx >=0
    @        && _y + dy >= 0;
    @  assignable _x, _y;
    @  ensures _x == \old(_x + dx)
    @        && _y == \old(_y + dy);
    @ also
    @ private normal_behavior
    @  requires _x + dx > = 0
    @        && _y + dy >= 0;
    @  assignable _x, _y;
    @  ensures _oldX == \old(_x)
    @       && _oldY == \old(_y);
    @*/
  public void move(int dx, int dy){
    _oldX = _x;
    _oldY = _y;
    _x += dx;
    _y += dy;
  }

  // ...
}
```

```
public class ScreenPoint extends Point {

  /*@ also
    @ protected normal_behavior
    @  requires _x + dx < 0;
    @  assignable _x, _y;
    @  ensures _x == 0;
    @ also
    @ protected normal_behavior
    @  requires _y + dy < 0;
    @  assignable _x, _y;
    @  ensures _y == 0; @*/
  public void move(int dx, int dy){
    if (_x + dx >= 0) _x += dx;
    else              _x = 0;
    if (_y + dy >= 0) _y += dy;
    else              _y = 0;
  }

     // ...
   }
```

Figure 2.20: JML specification for Cartesian points [42].

specification cases [42, Rule 2 and Rule 3], and according to Leavens, this issue is important future work.

2.1.8 Tool support

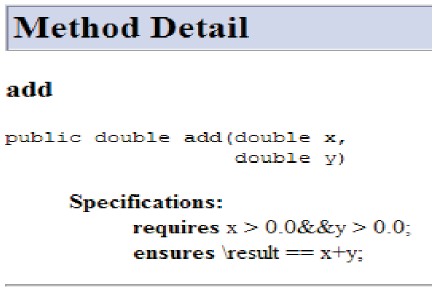
This section describes some tools that are currently available for JML. Most of these tools are part of the standard distribution of JML [11] — also known as JML tools suite, which is freely available from the sourceforge web page for JML[4].

Jmldoc: The documentation generation tool

The JML documentation generation tool — also known as jmldoc — generates HTML web pages from JML specifications. It is a modified version of the javadoc tool [26], which recognizes JML specifications. The web pages generated by the jmldoc are similar to the ones generated by the javadoc tool. The only difference is that the jmldoc tool also includes JML specifications in the documentation. The documentation provided by the jmldoc is helpful to present the obligations that programmers must follow to fulfill the contract. The jmldoc was developed by David Cok and by Raghavan [65] as a part of the JML tools suite [11].

An example of a jmldoc output is shown in Figure 2.21. It depicts the output for the method `add` of the interface `ICalc` (see Figure 2.16).

[4]The sourceforge page for JML is: http://sourceforge.net/projects/jmlspecs


Method Detail

add

```
public double add(double x,
                  double y)
```

Specifications:
requires x > 0.0&&y > 0.0;
ensures \result == x+y;

Figure 2.21: Example of jmldoc in JML.

Jml: The type checker

The JML type checker was developed at the Iowa State University. It is responsible for verifying the syntax and semantics of the JML annotations. After the typechecking phase, an abstract syntax tree (AST) is generated. This AST is used by the JML compiler to generate the assertion checking code.

Jmlrac: The runtime assertion checker

The JML runtime assertion checker — also known as jmlrac — is an extension of the standard Java compiler (javac), including JML runtime libraries to check JML assertions.

Jmlunit: The unit testing tool

The JML unit testing tool — also known as jmlunit — generates code for testing Java classes and interfaces. It is used to automate unit testing of Java code. The JML unit testing tool (jmlunit) uses the JML runtime assertion checker (jmlrac) to provide checking methods for the test oracle — JML specifications can be viewed as a test oracle [62]. The method checking code catches assertion violation errors from method calls to report whether the test data violates any assertion (e.g., precondition) of the method. In order to see examples and more details about jmlunit's implementation, please refer to [15].

Esc/Java Tool

Esc/Java2 [25] performs compile time verifications to check some common errors in Java code, such as dereferencing null, casting to incompatible types, or indexing an array out of its bounds. By Esc/Java2 one can check the consistency between the Java code and the given JML annotations. It responds with a list of possible errors after the program verification. The Esc/Java2 tool uses the theorem prover Simplify [22], which translates a given JML annotated program into logical formulas [44].

Other tools

Similar to Esc/Java2 [25], the Loop tool [73] performs a static verification of the Java programs annotated with JML specifications. The Loop tool translates Java classes

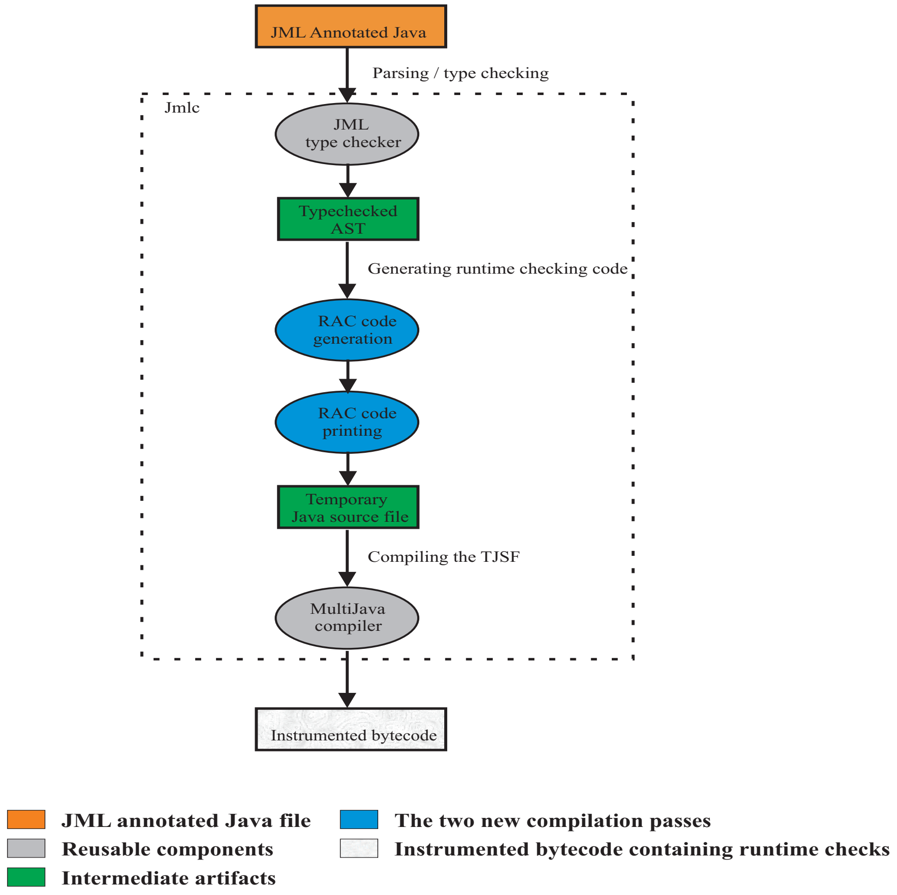


Figure 2.22: The structure of the JML compiler (jmlc).

into high order logic for two theorem provers: PVS [60] and Isabelle [61]. The Jack tool provides an environment for Java and Java Card program verification using JML annotations. As with Loop tool, the Jack tool translates the annotated Java class with JML into high order logic for different theorem provers such as PVS [60]. The Krakatoa [48] tool uses JML as specification language and produces proof obligations for the theorem prover Coq [72].

2.1.9 The JML compiler

The JML compiler (jmlc) [14] was developed at the Iowa State University. It is a runtime assertion checking compiler, which converts JML annotations into automatic runtime checks. The instrumented Java bytecode produced is verified during runtime and notifies when any assertion violation occurs.

Design

The JML compiler is built on top of the MultiJava compiler [17], which is an extension of the Java compiler. It reuses the front-end — the JML type checker tool — of the existing JML tools [11] to verify the syntax and semantics of the JML annotations and produces a typechecked abstract syntax tree (AST). The compiler introduces two new compilation passes: the “runtime assertion checker (RAC) code generation”; and the

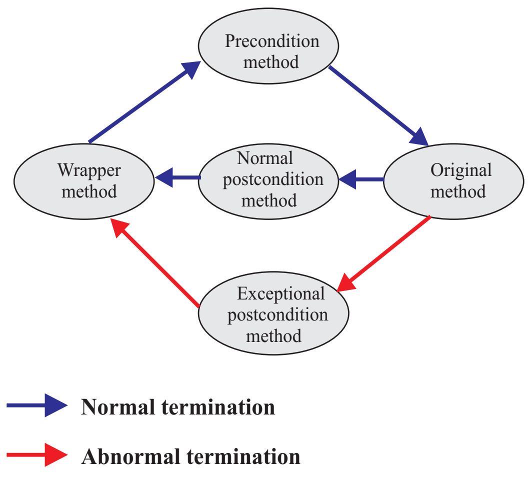


Figure 2.23: Wrapper approach strategy.

"runtime assertion checker (RAC) code printing". The *RAC code generation* creates the assertion checker code from the AST. It modifies the abstract syntax tree to add nodes for the generated checking code. The *RAC code printing* writes the new abstract syntax tree to a temporary file.

For each Java method three *assertion methods* are generated into a temporary Java source file (TJSF): one for precondition checking; and two for postcondition checking (for normal and exceptional termination). They are invoked before method's execution (precondition checking), after method's execution (normal postcondition checking) and when an exception is thrown by the called method (exceptional termination checking). Concerning invariants, the JML compiler also generates (when necessary) *assertion methods* for both kinds of invariants (for instance and static invariants) and writes them likewise the other *assertion methods* into the TJSF. Finally, instrumented bytecode is produced by compiling the TJSF through the MultiJava compiler (see Figure 2.22). The instrumented bytecode produced contains *assertion methods* code embedded to check JML contracts at runtime.

Wrapper approach

The *wrapper approach* [14, 4.1.3] is a strategy used by the JML compiler to implement the assertion checking. Each method is redeclared as private with a new name. Then, a method known as *wrapper method* is generated with the name of the original method. Its surrounds the original method (now with a new name) with the assertion methods. Hence, client method calls the wrapper method, which is responsible for calling the original method with appropriate assertion checks (e.g., precondition checking). The JML compiler is responsible for controlling the order of execution of assertion methods.

Figure 2.23 depicts the wrapper approach strategy. If a client calls the original method, the call goes to the wrapper method. In this way, the precondition assertion method is the first assertion method called, and then only if the precondition is satisfied, it calls the original method. After calling the original method, if it terminates normally, the normal postcondition assertion method is called; otherwise, the exceptional postcondition assertion method will be called.

2.2 An Overview of AspectJ

In this section we present AspectJ [36], a general-purpose aspect-oriented extension to Java. It provides support for implementing in a modular way several crosscutting concerns [37], such as persistence, distribution, and design by contract concerns. By using AspectJ one can affect the execution of a Java program. In the following, we present the main AspectJ features and constructs.

2.2.1 Join Points and Pointcuts

Join points are well-defined points in the execution flow of a program. Join points can be applied to methods and constructors. A *pointcut* is a set of join points. An *advice* is a code associated to a pointcut. Whenever the program execution reaches one of the join points described in the pointcut, the advice associated with the pointcut is executed. This is useful to indicate where and when intercept the execution of the program to execute tasks not directly related with the main concerns of the original program. This permits the addition of aspects to existing software, or the design of software with a clear separation of concerns.

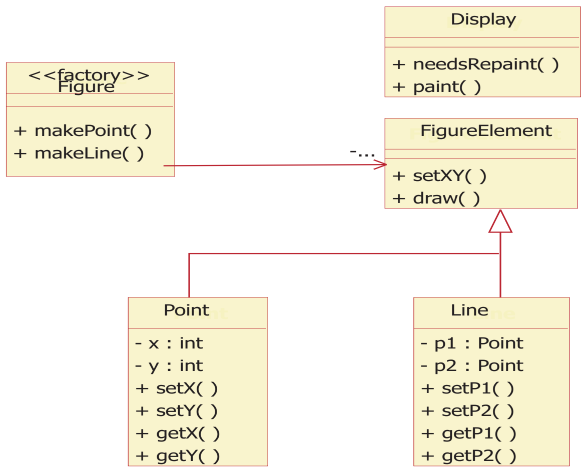


Figure 2.24: Figure editor example [36].

Table 2.3 presents some examples of pointcuts to capture points in the execution flow of the figure editor example [36] from Figure 2.24.

It is possible to use wildcards (∗) in the pointcut definitions. For example:

```
call(void Figure.make*(..))
```

This code intercept calls to any method with prefix name "`make`" defined in the class `Figure`, such as the factory methods `makePoint` and `makeLine`.

```
call(public * Figure.* (..))
```

This code identifies calls to any public method defined in the class `Figure`.

`execution(void Point.setX(int))`	When the `setX` method from class Point with an `int` parameter executes
`call(voidPoint.setX(int))`	When the `setX` method is called
`this(Point)`	Matches join points when the currently executing object is an instance of `Point`
`target(FigureElement)`	Matches join points when the target object is an instance of `FigurePoint`
`args(int)`	When the executing or called method has an `int` parameter
`within(Display)`	Matches executing code defined in the type `Display`
`cflow(call(voidDisplay.paint()))`	Matches executing code in the control flow of a call to a `Displays` no-argument paint method

Table 2.3: Pointcut examples.

`before`	Executes immediately before the join point execution
`after returning`	Executes after the join point returns normally
`after throwing`	Executes after the join point returns abnormally by throwing an exception
`after`	Executes after the execution of the join point even its returns normally or throwing an exception
`around`	Executes when the join point is reached; it has total control over the execution of the join point

Table 2.4: The kinds of AspectJ advices.

2.2.2 Advice

An *advice* is a method-like mechanism used to define the additional code that should be executed at join points. The kinds of AspectJ advices are presented by Table 2.4.

The Figure 2.25 illustrates an AspectJ advice definition. The advice in Figure 2.25

```
after(Point p): call(void p.setX(int)) || call (void p.setY(int)){
  System.out.println(Point p changed!);
}
```

Figure 2.25: Example of AspectJ advice definition.

prints a message after every call to `setX` or `setY` methods on a Point object. Note that by using parameters we can expose part of the execution context at join points. Thus, values exposed by an advice (parameter `p`) can be used in its body.

In order to change the normal execution of a join point, we use the around advice, which has total control of the affected join point. The example in Figure 2.26 demonstrate how to change the parameter of method `setY`. The `around` advice in Figure 2.26 exposes the `int` parameter, called `y`, which is an argument of the `setY` method. In this way, we check if this argument is within the range $(0 - 300)$. If it is, the execution proceeds normally (transparently); otherwise, we change the real execution of the method

```
void around(int y): execution(void Point.setY(int)) && args(y) {
  if (y>=0 && y<=300){
    proceed(y);
  }
  else{
    proceed(30); // note that we changed the real input parameter
  }              // of the method setY(int)
}
```

Figure 2.26: AspectJ code for changing the parameter of method `setY`.

by passing the `y` argument with value `30`. Note that the AspectJ `proceed` method is responsible for calling the original affected method.

2.2.3 Static Crosscutting

All AspectJ constructs described so far use *dynamic crosscutting mechanism*, which changes the way a program executes. In addition, AspectJ allows one to change the static structure of a class. This mechanism is known as *static crosscutting mechanism.*

A static crosscutting mechanism, also known as *inter-type declarations*, we can:

- add new methods to an existing class;
- add new fields to an existing class;
- extend an existing class with another;
- implement an interface in an existing class;
- convert checked exceptions into unchecked exceptions.

An example of an inter-type member declaration is shown in what follows:

```
public String Point.getXY(){return "("+this.getX()+","+this.getY()+")";}
```

This AspectJ code declares a method into class `Point` (the `Point.` prefix tells AspectJ where to insert the method definition). Such a method returns a `String` containing both coordinates (`x` and `y`) of the class `Point`. Moreover, like any other method, the inserted method `getXY` can be called anywhere by an object `Point`.

```
public static final int Point.MAX_Y= 300;
```

We can also use the inter-type declaration to declare fields. As an example, the AspectJ code inserts a static int field `MAX_Y` into class `Point`.

Suppose now that we want to make the class `Point` to implement the interface `Cloneable`. This is done by the following AspectJ code:

```
declare parents: Point implements Cloneable;
```

The AspectJ construct `declare parents` makes the class `Point` to implement the interface `Cloneable`. Thus, this is another example where AspectJ changes a static structure of a class.

2.2.4 Aspects

The main construct of AspectJ is known as *aspect*. Each aspect, similar to a class, can define fields, methods, constructors, initializers, hierarchies, and so forth. Moreover, by using aspects we can define crosscuting members such as pointcuts, advices, and inter-type declarations for implementing in a modular way several crosscutting concerns [37].

By default in an AspectJ program, each aspect has only one instance that affects (crosscuts) its entire program. Nevertheless, there are situations when one desire to associate an aspect with an individual object or control flow. In order to cover these situations in more detail, refer to [37].

Privileged aspects

The Java visibility rules are also applied to aspects. However, there are situations where an aspect needs to access private or protected members of types. Hence, to allow this kind of access, the AspectJ aspects must be declared privileged. In order to illustrate this, suppose the fields `x` and `y` of the class `Point` are both private. The privileged

```
privileged public aspect PrivateAccess {
  after (Point p, int x, int y):
  execution(void setXY(int,int)) && args(x,y) && this(p){
    if ((p.x+ x) > 300 || (p.y+y) > 300){
      throw new PointOutOfBoundsException();
    }
  }
}
```

Figure 2.27: Example of AspectJ aspect privileged.

aspect defined in Figure 2.27 can access the fields `x` and `y`, even though they are private members of the `Point` class. The used `after` advice checks if such points have legal ranges; otherwise, a `PointOutOfBoundsException` is thrown to signaling the violation of the permitted range.

2.2.5 Design by Contract Concern

In an interview [35], Kiczales, one of the major mentors of AOP and AspectJ, cites DBC (Design by Contract) as an example of a crosscutting concern: "[...] *there are many other concerns that, in a specific system, have crosscutting structure. Aspects can be used to maintain internal consistency among several methods of a class. They are well suited to enforcing a Design by Contract style of programming.*". Thus, by using AspectJ aspects provide simple mechanisms for checking pre- and post-conditions, as well as invariants for constraint enforcement. This kind of capability is very useful when using Design by Contract [53] as a programming style. It is important to notice that enforcing Design by Contract style with AspectJ is one important issue addressed by this dissertation. As an example of DBC [36, Section 3.2], consider: The code in Figure 2.28 defines two pointcuts that refers to method calls that change the fields `x` and `y`. The Figure 2.29 defines two `before` advices for the precondition checking

```
pointcut setXs(int x): call(void FigureElement.setXY(x, int)) ||
                       call(void Point.setX(x));
pointcut setYs(int y): call(void FigureElement.setXY(y, int)) ||
                       call(void Point.setY(y));
```

Figure 2.28: AspectJ code for defining two pointcuts.

(contract checking) for operations that change `x` and `y` coordinates. The AspectJ code

```
before(int x): setXs(x) {
  if ( x < MIN_X || x > MAX_X ) {
    throw new IllegalArgumentException(x is out of bounds.);
  }
}
before(int y): setYs(y) {
  if ( y < MIN_Y || y > MAX_Y ) {
    throw new IllegalArgumentException(y is out of bounds.);
  }
}
```

Figure 2.29: AspectJ `before` advices for precondition checking.

in Figure 2.29 checks if the points are inside the range (`MIM` or `MAX`). If it does not, an `IllegalArgumentException` is thrown signaling the violation.

2.2.6 AspectJ compilers

This section briefly presents the two well-known AspectJ compilers used by the AspectJ community (academic and industry).

ajc compiler

The ajc compiler is the standard (official) compiler of the AspectJ language [36]. In fact, ajc is a pre-compiler written in Java. It compiles aspects and classes together and produces Java source code. Such a Java code is usually referred as a “weave”, then ajc invokes javac to actually compile the “weave” into bytecode.

The ajc compiler is also integrated in the Eclipse environment [2] through the AJDT plugin [1] (AspectJ Development Tools, as an extension of the JDT, Java Development tools).

Hilsdale and Hugunin [30] provide more details about the ajc compiler.

abc compiler

The AspectBench Compiler (abc) is an academic compiler [5] that implements the full AspectJ language [36]. The compiler was conceived as a workbench to facilitate easy experimentation with new language features and implementation techniques. In particular, new features for AspectJ have been proposed that require extensions in many dimensions: syntax, type checking and code generation, as well as data flow and control flow analyses. Experiments conducted in this dissertation demonstrated that abc produce a code of better quality if compared with the ajc compiler.

For more details about the abc compiler please refer to [5].

2.3 An Overview of Java ME Platform

This section gives an overview of the Java ME (Java Micro Edition) [56, 63], introducing its features such as configurations and profiles. However, for this dissertation purpose, we focus on the differences between Java SE and Java ME platforms.

2.3.1 Java Editions

Because Java encompasses such a wide range of applications running on so many diverse environments, the platform was divided into three sections in the late 1990s:

- **Standard Edition** (Java SE) — designed to run on desktop and workstations computers;
- **Enterprise Edition** (Java EE) — designed to run at server-based applications (With built-in support for Servlets, JSP, and XML);
- **Micro Edition** (Java ME) — designed for devices with limited memory, display and processing power.

The relationship between the three can be thought of as a superset-subset one. In this way, Java ME (J2ME) is a subset of Java SE, while itself is a subset of Java EE (see 2.30).

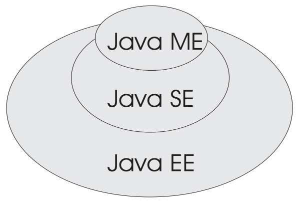


Figure 2.30: Java editions and their relationships.

2.3.2 Device Configurations and Device Profiles

A configuration defines the basic Java ME runtime environment such as the virtual machine and the set classes used to run on devices applications. There are two configurations for Java ME: (1) Connected Limited Device Configuration (CLDC); and (2) Connected Device Configuration (CDC). The CLDC contains a strict subset of the Java-class libraries, and is the minimal amount needed for a Java virtual machine to operate. CLDC is basically used to classify a number of devices into a fixed configuration. A configuration provides the most basic set of libraries and virtual-machine features that must be present in each implementation of a Java ME environment. The CLDC is designed for 16-bit or 32-bit small computing devices with limited memory space. It uses the KVM virtual machine [4] implementation, which is a small version of the classic JVM (Java Virtual Machine). A CLDC includes devices such as pagers, cell phones, dedicated terminals, and handhelds.

On the other hand, the The Connected Device Configuration (CDC) is a subset of Java SE, containing almost all the libraries that are not GUI related. It is richer than CLDC. CDC devices use a 32-bit architecture and have at 2MB of memory available. It implements a complete version of the JVM. CDC supports devices such as home appliances, point-of-sale terminals, and smart phones.

A profile adds domain-specific classes to a particular Java ME configuration. Examples are user interface classes, persistence mechanisms, and so on. Profiles and configurations work together in order to provide a complete Java runtime environment. Currently, there are seven profiles available: Foundation Profile; Game Profile; Mobile Information Device Profile; PDA Profile; Personal Profile; Personal Basis Profile; and RMI Profile.

MIDP (Mobile Information Device Profile)

The Mobile Information Device Profile (MIDP) is one of the seven profiles mentioned above. It is used with the CLDC configuration and was designed to run on mobile devices such as palm tops. MIDP is used with wireless Java applications.

A Java ME application is usually composed of several MIDlets. A MIDlet is a unit based on MIDP. For example, application such as calculator, agenda, etc commonly found in mobile phones are composed of several MIDlets. Programming a MIDlet is similar as creating a Java SE application in that you define a class and related methods. The entry point in Java code for a MIDlet is a class that extends the class MIDlet. The lifecycle of the MIDlet is managed through three of methods that are called by the Java Application Management System (AMS): `startApp` - called on application activation; `pauseApp` - called on application deactivation (suspend); and `destroyApp` - called on application termination. Figure 2.31 shows an example of a simple MIDlet application.

```
public class MidletSimples extends MIDlet {
  //The MIDlet constructor
  public MidletSimples( ){}
  // A method used to starts a MIDlet application
  public void startApp( ){}
  // A method used to pause a MIDlet application
  public void pauseApp( ){}
  // A method used to destroy a MIDlet application
  public void destroyApp( boolean unconditional ){}
}
```

Figure 2.31: Example of simple MIDlet application.

The method `startApp` is called by the device application manager when the MIDlet is started and contains statements that are executed each time the application begins its execution. The method `pauseApp` is called before the device application manager temporarily stops the MIDlet. The device application manager restarts the MIDlet by recalling the method `startApp`. Finally, the method `destroyApp` is called to the termination of the MIDlet by the device application manager.

Other profiles

For more details about the other six profiles please refer to [56, 63].

2.3.3 The K Virtual Machine

The KVM [4] has been developed as part of larger effort to provide a modular, scalable architecture for the development and deployment of portable, dynamically downloadable, and secure applications in embedded devices. The "K" in KVM stands for kilobyte, signifying that the KVM runs in kilobytes of memory as opposed to megabytes. It supports a subset of the features of the "classic" JVM. For instance, a KVM does not support reflection and object finalization or serialization. The CLDC specifies use of the KVM.

2.3.4 Compatibility between Java SE and Java ME applications

Java ME is a small subset of Java SE. Many components were removed to keep the Java ME platform small and efficient An example is the Abstract Window Toolkit (AWT) — many mobile devices do not have the screen capabilities to provide advanced user interface components. However, if the Java SE application addresses only classes available within the Java ME, the program will run on both platforms. Java ME compiler includes a phase known as *preverification* [56, 63]. It is used to check if the generated bytecode is compatible with Java ME environment.

On the other hand, the majority code written for Java ME applications requires specific interface and event handling code, not supported by Java SE. It limits what types of programs will be appropriate for both platforms.

2.3.5 Java SE Inherited Classes

Since the KVM is a subset of the classic JVM, only few classes of the Java SE platform are supported by Java ME. In order to see the classes that are common for both please refer to Appendix A.

2.4 Chapter Summary

In this chapter, we presented the main concepts addressed in this dissertation. We provided an overview about JML (Java Modeling Language), AspectJ (an aspect oriented extension to Java), and Java ME platform.

Regarding JML and AspectJ, we discussed their main concepts and features. We pointed out the main JML specification clauses (e.g., requires clause used for preconditions). We also described the main features of AspectJ such as dynamic and static crosscutting mechanism used to affect the executions of Java programs as well as its static structure.

Finally, we described some important aspects of the Java ME platform. For this dissertation purpose, we focused on the main difference between Java SE and Java ME platforms.

Chapter 3

ajmlc: A JML Compiler Targeting AspectJ code

In this chapter we present the translation rules from JML predicates into AspectJ aspects [36]. An informal description of the translation mechanism has already been considered in [67, 66].

3.1 The implementation strategy

Similarly to Cheon [14], we reuse the front-end of the JML compiler, known as JML Type Checker [11]. Then, we modify the code generation part of the original JML compiler[1] to introduce other two new compilation passes: the *Aspect RAC* code generation; and the *Aspect RAC* code printing. The former produces assertion checker code from the typechecked AST, whereas the latter writes the assertion checker code to a temporary Aspect source file. We traverse the typechecked AST generating *Aspect Assertion Methods* (AAM) for each Java method in a temporary Aspect source file: one for precondition checking, and another for both kinds of postconditions in JML (normal and exceptional). Eventually (when necessary) we also generate AAM for both kinds of invariants in JML (instance and static). These AAM are compiled through the AspectJ compiler (ajc or abc [5]), which weaves the AAM with the Java code (see Figure 3.1, which depicts the compilation passes of the ajmlc). The result, unlike jmlc, is an instrumented bytecode compliant to both Java SE and Java ME applications.

3.2 Translation rules

In this section we present a set of translation rules from annotated Java types into AspectJ aspects. The translation function,
$MAP_{JmlToAspectJAspect}$: $Annotated\_Java\_Type \rightarrow AspectJ\_Aspect$ takes a JML annotated Java type as argument and generates an AspectJ aspect. This generated aspect contains several AspectJ constructs such as advices [36], which represents the JML annotations (e.g., preconditions). JML predicates are composed of Java and JML expressions

[1] Part of the code of the original JML compiler that we used to implement the ajmlc was based on the JML 5.5 version available to download at `http://sourceforge.net/projects/jmlspecs`.

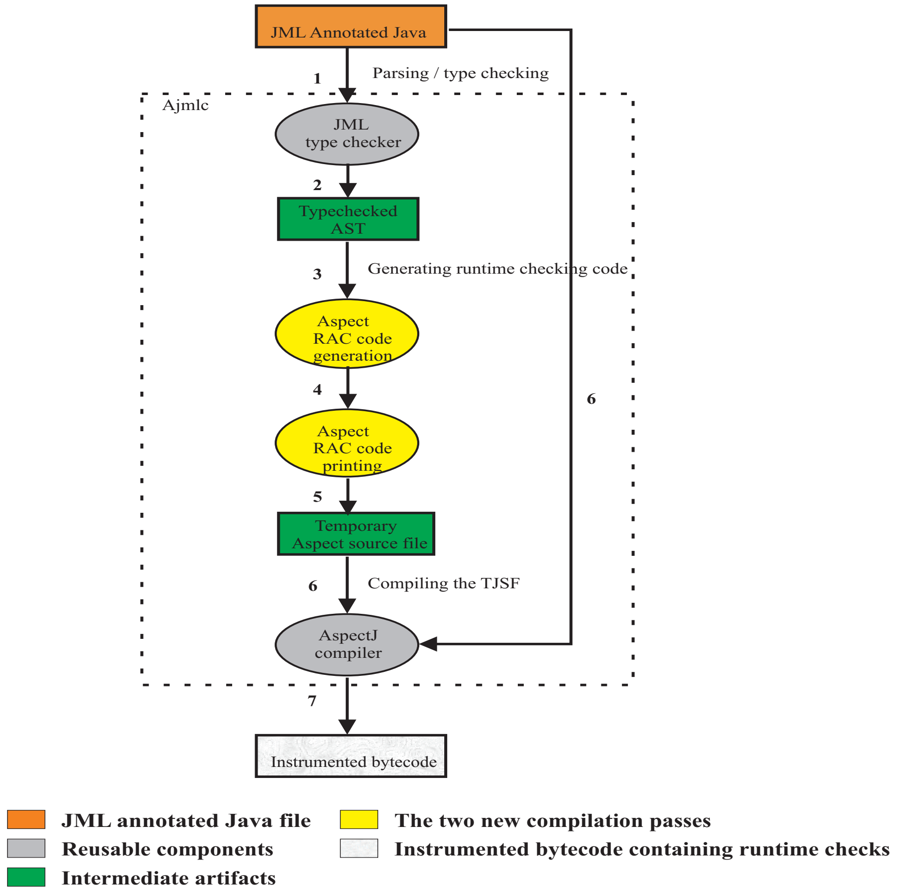


Figure 3.1: The structure of our JML compiler (ajmlc).

(Appendix B presents the subset of JML supported by ajmlc).

For the translation rules, the following elements will also be employed:

- The argument *Annotated_Java_Type* is a tuple composed by the following elements:
  - — $TMod$ — is a sequence of type modifiers, $\langle TMod_1, ..., TMod_x\rangle$;
  - — TN — is the name of the *Annotated_Java_Type* to be compiled;
  - — $Meth$ — is a sequence of methods, $\langle Meth_1, ..., Meth_m\rangle$;
  - — SCN — is the name of the superclass, in which the *Annotated_Java_Type* is subtype;
  - — SIN — is a set of superinterface names, $\{SIN_1,...,SIN_n\}$, of the *Annotated_Java_Type*;
  - — $Inst\_Inv$ — is a set of instance invariants, $\{Inst\_Inv_1,...,Inst\_Inv_r\}$, into the *Annotated_Java_Type*;
  - — $Stat\_Invs$ — is a set of static invariants, $\{Stat\_Inv_1,...,Stat\_Inv_s\}$, into the *Annotated_Java_Type*;
- Each method in $Meth$ is a tuple composed by the following elements:

```
$MAP_{PreconditionToAspectJCode}[\![TN, TMod, SCN, SIN, MN, MMod, FP, PTN, PIN, RT, SC]\!] \triangleq$
  if SC $\neq \langle\epsilon\rangle$
  then if 'static' $\in$ MMod
       then $\langle\langle$staticBeforeAdviceForPreconditionChecking$\rangle\rangle$
            $\langle\langle$staticPreconditionChekingMethod$\rangle\rangle$
       else if 'interface' $\in$ TMod
            then $\langle\langle$preconditionChekingMethod$\rangle\rangle$
            else $\langle\langle$beforeAdviceForPreconditionChecking$\rangle\rangle$
                 $\langle\langle$preconditionChekingMethod$\rangle\rangle$
```

Figure 3.2: General structure of the code that checks preconditions of the method *MN*.

— $MMod$ — is a sequence of method modifiers, $\langle MMod_1, ..., MMod_y\rangle$;

— MN — is the method name;

— FP — is a sequence of formal parameters (pairs of types and identifiers), $\langle T_1, Id_1, ..., T_z, Id_z\rangle$;

— PTN — is a sequence of parameter type names, $\langle T_1, ..., T_z\rangle$;

— PIN — is a sequence of parameter identifier names, $\langle Id_1, ..., Id_z\rangle$;

— RT — is the method return type;

— Ex — is a set of exceptions, $\{E_1,...,E_s\}$;

— SC — is a sequence of local JML specification cases, $\langle SC_1, ..., SC_k\rangle$. Such local specification cases are represented as tuples, $\langle P_i, Q_i, R_i\rangle_{i=1,...,k}$, composed by preconditions P, normal postconditions Q, and exceptional postconditions R.

- Each sequence and each set could also be empty:

  — $\langle\epsilon\rangle$ — denotes an empty sequence;

  — $\{\}$ — denotes an empty set.

It is important to notice that, in JML, when a method has several specifications cases that are not inherited (are local to the method), we also join these specification cases using the JML `also` construct. The semantics of joining these specifications is described in Section 2.1.6.

3.2.1 Precondition translation

Preconditions are predicates that must hold before a method execution. Otherwise an exception must be thrown, indicating the precondition violation. According to the JML semantics, all local preconditions of a method (specification cases) lead to a disjunction, so that we obtain a single effective precondition predicate, $P \equiv P_1 \vee ... \vee P_k$.

The auxiliary function $MAP_{PreconditionToAspectJCode}$ in Figure 3.2 (Appendix D.1.2 presents the definition of this auxiliary function) represents the general structure of the

```
⟨⟨beforeAdviceForPreconditionChecking⟩⟩≡
   let ⟨$T_1$,$Id_1$...$T_z$,$Id_z$⟩ = FP in
   let ⟨$T_1$,...$T_z$⟩ = PTN in
   let ⟨$Id_1$...$Id_z$,⟩ = PIN in
   before (TN current, $T_1$ $Id_1$,...,$T_z$ $Id_z$):
     execution (RT TN.MN ($T_1$,...,$T_z$)) &&
     within (TN) &&
     this (current) &&
     args ($Id_1$,...,$Id_n$){
        if (!current.check$MN$TN ($Id_1$,...,$Id_z$)){
           throw new JMLInternalPreconditionError();
        }
     }
```

Figure 3.3: AspectJ `before` advice to check preconditions of the non-static method *MN*.

AspectJ code that checks a method preconditions. An AspectJ `before` advice (see Figure 3.3) is used to insert an extra behavior before some specified points in the Java program. The affected points are defined by means of the AspectJ designator `execution`. It specifies which method executions will be affected, in this case, executions of a method defined by the *MN*. In addition we employ the AspectJ designators `this` and `within`. The former match the current execution object or its subclasses (we do not need to use the AspectJ wildcard `+` since subtypes that match are already captured without using `+`) and takes the form `this(Type or ObjectIdentifier)`; as we need to use the context of the execution object (e.g., methods or fields), we use the form that uses `ObjectIdentifier` (as in translation rules using the advice parameter identifier *current*). The latter constrains the execution points of methods of a specified type *TN*, which avoids executions in its subclasses (only method executions in the lexical scope in the type *TN* are allowed to be advised). In short, the behavior added by the `before` advice is to check the effective precondition and throws `JMLInternalPreconditionError` if it is violated.

AspectJ `before` advice used to check preconditions calls a precondition checking method inserted (see Figure 3.4) by using AspectJ static crosscutting mechanism 2.2.3(also known *inter-type declaration*). As shown in Figure 3.4, we have to analyze two scenarios before inserting a precondition checking method: (1) with precondition inheritance, and (2) with no precondition inheritance. The first scenario besides the effective precondition, takes into account inherited preconditions (e.g., superclass, superinterfaces). Thus, the first scenario evaluates each local precondition P_i and all inherited ones in a disjunction form. Note that we have explicit calls to *MN*'s supertypes. On the other hand, the second scenario deals only with local preconditions, which checks only each local precondition P_i in a disjunction form.

Specific scenarios

Consider a scenario that the specification case of the method *MN* is: $\langle Q_i, R_i \rangle_,$ where the precondition was omitted. The auxiliary function $MAP_{PreconditionToAspectJCode}$ generates two possible default values to treat the omitted precondition: (1) if the omitted precon-

```
⟨⟨preconditionChekingMethod⟩⟩≡
    let ⟨$T_1$,$Id_1$...$T_z$,$Id_z$⟩ = FP in
    if (SCN ≠ ‘java.lang.Object’) ∨ (SIN ≠ {})
    then let ⟨$SIN_1$,...$SIN_n$⟩ = PTN in
         public boolean TN.check$MN$TN ($T_1$ $Id_1$,...,$T_z$ $Id_z$){
           return ⟨⟨preconditionsToCheck⟩⟩ ||
             checkPre$MN$SCN ($Id_1$,...,$Id_z$) ||
             checkPre$MN$SIN1($Id_1$,...,$Id_z$),...,||
             checkPre$MN$SINn($Id_1$,...,$Id_z$);
         }
    else
       public boolean TN.check$MN$TN ($T_1$ $Id_1$,...,$T_z$ $Id_z$){
         return ⟨⟨preconditionsToCheck⟩⟩;
       }

⟨⟨preconditionsToCheck⟩⟩≡
   if SC = ⟨ε⟩
   then
      false
   else let ⟨P,Q,R⟩ = SC in
        if P ≠ ⟨ε⟩
        then let ⟨$P_1$,...$P_k$⟩ = P in
        $P_1$ || ... || $P_k$
        else
           true
```

Figure 3.4: Code for inserting a precondition method for *MN* (considering inheritance).

dition is part of a lightweight specification case, the default value is `\not_specified`; and (2) if the omitted precondition is part of a heavyweight specification case, the default value is `true`. Indeed, similar to the original JML compiler (jmlc), ajmlc treats `\not_specified` values as `true` values. Therefore, even if we have an omitted precondition in the method's specification case, our compiler generates AspectJ code (precondition checking method) to check the default value of the precondition under the rules of the default specification cases. This apparently unnecessary work, in fact is important to check inherited specifications. Consider now that the specification case of the method *MN* is $\langle\epsilon\rangle$ (empty). Our translation rules generate code (precondition checking method) only for two situations: (1) if the method *MN* is overridden, a default value `false` and code to call inherited specifications is generated; and (2) if the method *MN* is not overridden, a default value `false` is generated for the precondition. These scenarios are covered in the compilation process described above. In order to see examples of these two specific cases please, refer to Appendix D.

Static method precondition translation

The code in Figure 3.5 represents the general structure of the AspectJ `before` advice used to check preconditions of the static method *MN*. Note the absence of the advice parameter `current` and the designator `this`, Because static methods do not have the this object associated with them, thus the `this` pointcut will not match the execution of such a method. Another reason is that we do not want to expose the current executing

```
⟨⟨staticBeforeAdviceForPreconditionChecking⟩⟩≡
   let ⟨T_1,Id_1...T_z,Id_z⟩ = FP in
   let ⟨T_1,...T_z⟩ = PTN in
   let ⟨Id_1...Id_z,⟩ = PIN in
   before (T_1 Id_1,...,T_z Id_z):
     execution (RT TN.MN (T_1,...,T_z)) &&
     args (Id_1,...,Id_n){
        if (!TN.check$MN$TN (Id_1,...,Id_z)){
           throw new JMLInternalPreconditionError();
        }
     }
```

Figure 3.5: AspectJ `before` advice to check preconditions of the static method *MN*.

```
⟨⟨staticPreconditionChekingMethod⟩⟩≡
   let ⟨T_1,Id_1...T_z,Id_z⟩ = FP in
   public static boolean TN.check$MN$TN (T_1 Id_1,...,T_z Id_z){
     return P_1 || ... || P_k;
   }
```

Figure 3.6: Code for inserting a static precondition method for *MN*.

object properties; only static information is needed. In addition, we do not used the AspectJ designator `within` because since such a method is static, will match only the lexical scope of the type that declares it. In other words, the `before` advice used to check the precondition of the method *MN* does not apply to *TN* subclasses. In this way, we only consider the local preconditions (see the static precondition checking method structure in Figure 3.6), since static method specifications are not inherited in JML [39, 43].

It is important to notice that if the precondition is omitted or if the specification case is empty, no AspectJ code is generated to check the effective precondition of the static method *MN*. This is for optimization purposes and because static methods cannot inherit specifications [39, 43]. Thus, we need not a precondition checking method to check the `true` value with no inherited preconditions.

3.2.2 Postcondition translation

Postconditions are properties in JML that must hold after method executions. There are two kinds of postconditions in JML: normal and exceptional postcondition. Normal postconditions are predicates that must hold when the method terminates with no exception thrown, otherwise an exception must be thrown to signal the violation of the normal postcondition. Normal postconditions evaluate the predicate (`\old(`P_i`)` $\Rightarrow$ Q_i), where Q_i is a normal postcondition that must hold after method execution, and P_i is its corresponding precondition that must hold before the evaluation of Q_i. Note that P_i is evaluated in the pre-state, which means before method execution. In order to represent this behavior, the JML old expression is used (`\old(`P_i`)`). According to the JML semantics, unlike preconditions, all local normal postconditions of a method

```
$MAP_{PostconditionToAspectJCode}[\![TN, MN, MMod, FP, PTN, PIN, RT, Ex, SC]\!] \triangleq$
  if $SC \neq \langle\epsilon\rangle$
  then let $\langle P, Q, R\rangle$ = $SC$ in
       if $(Q \neq \langle\epsilon\rangle) \vee (R \neq \langle\epsilon\rangle)$
       then let $\langle Q_1, \ldots Q_k\rangle$ = $Q$ in
            let $\langle R_1, \ldots R_k\rangle$ = $R$ in
            if ‘static’ $\in MMod$
            then $\langle\langle staticAroudAdviceForPostconditionChecking\rangle\rangle$
            else $\langle\langle aroudAdviceForPostconditionChecking\rangle\rangle$
```

Figure 3.7: General structure of the code that checks postconditions of the method *MN*.

are “conjoined” into a single effective normal postcondition predicate, ($\texttt{\textbackslash old}(P) \Rightarrow Q$) $\equiv$ ($\texttt{\textbackslash old}(P_1) \Rightarrow Q_1$) $\wedge$ ... $\wedge$ ($\texttt{\textbackslash old}(P_k) \Rightarrow Q_k$). Exceptional postconditions, on the other hand, are predicates that must hold when the method terminates by throwing an exception. They evaluate the predicate ($\texttt{\textbackslash old}(P_i) \Rightarrow R_i$), where R_i is an exceptional postcondition that must hold when the method terminates abnormally, and P_i is its corresponding precondition, that must hold before the evaluation of R_i. As with a normal postcondition, P_i is evaluated in the pre-state according to the semantics of old in JML. Each exceptional postcondition R_i may consist of several signals clauses of the form (X_{11} e_{11}) ... (X_{kl} e_{kl}) R_i, where X_{ij} is an exception type, and e_{ij} is an optional variable that refers to the current exception thrown. Each R_i must hold when the method terminates abnormally and the exception thrown must be a type of X_{ij}. According to the JML semantics, all local exceptional postconditions of a method are “conjoined” into a single effective exceptional postcondition predicate, ($\texttt{\textbackslash old}(P) \Rightarrow R$) $\equiv$ ($\texttt{\textbackslash old}(P_1) \Rightarrow R_1$) $\wedge$ ... $\wedge$ ($\texttt{\textbackslash old}(P_k) \Rightarrow R_k$).

Handling old expressions

JML old expressions can appear in postconditions (normal or exceptional postconditions). In this way, if the method *MN* has a set of old expressions (local or super expressions) $\{\texttt{v}_1,...,\texttt{v}_f\}$, the ajmlc generates a new local variable $\texttt{old\$v}_i$ into the `around` advice generated (as shown in Figure 3.7 with the code chunk $\langle\langle saveAllOldValues\rangle\rangle$ represented by the Figure 3.10). To represent old expressions in preconditions, normal postconditions as well as in exceptional postconditions, we adopt the notations $P_i[\![v_i{:=}old\$v_i]\!]$, $Q_i[\![v_i{:=}old\$v_i]\!]$, and $R_i[\![v_i{:=}old\$v_i]\!]$. Here, every occurrence of $\texttt{v}_i$ in P_i, Q_i, and R_i is replaced with $\texttt{old\$v}_i$.

Handling result expressions

In JML, the expression `\result` refers to the result of the original method — for methods with non-void return type. Hence, each occurrence of the keyword `\result` in normal postcondition clauses is replaced with the variable `rac$result`, which represents the return of the `around` advice.

The auxiliary function $MAP_{PostconditionToAspectJCode}$ in Figure 3.7 (Appendix D.1.4 presents the definition of this auxiliary function) represents the general structure of the code that checks the postconditions of the method *MN*. The code chunk

```
⟨⟨aroundAdviceForPostconditionChecking⟩⟩≡
   let ⟨Ex_1,...Ex_s⟩ = Ex in
   let ⟨T_1,Id_1,...T_z,Id_z⟩ = FP in
   let ⟨T_1,...T_z⟩ = PTN in
   let ⟨Id_1...Id_z,⟩ = PIN in
   RT around (TN current, T_1 Id_1,...,T_z Id_z) throws Ex_1,...Ex_s :
     execution (RT TN.MN(T_1,...,T_z)) &&
     this (current) &&
     args (Id_1,...,Id_z) {
        RT rac$result; // represents the return of the method
        try{
          ⟨⟨saveAllOldValues⟩⟩
          try{
            // executing the original method
            rac$result = proceed(current, Id_1,...,Id_z);
            ⟨⟨checkNormalPostcondition⟩⟩
          } catch (Throwable rac$e){
              ⟨⟨rethrowJMLException⟩⟩
              ⟨⟨checkExceptionalPostcondition⟩⟩
          }
        } catch (Throwable rac$cause){
            throw new JMLEvaluationError(rac$cause);
        }
     }
```

Figure 3.8: AspectJ `around` advice to check postcondition of the non-static method *MN*.

⟨⟨*aroundAdviceForPostconditionChecking*⟩⟩ in Figure 3.8 presents the AspectJ `around` advice used to check the two kinds of JML postcondition clauses. The AspectJ `proceed` method represents the call to the original method from the advice. It separates the pre-state from the post-state. Because of that, it is suitable for handling old expressions, since it has total control before and after the constrained methods execution — this is the main reason for using the `around` advice to check postconditions. In this way, the code chunck ⟨⟨*saveAllOldValues*⟩⟩ in Figure 3.10 saves all old values used in postconditions.

As previously mentioned, the code in Figure 3.10 generates a new local variable `old$v`$_i$ into the `around` advice. However, the variable is created before the call to the `proceed` method, which saves the values in a pre-state environment. Moreover, the `around` advice inserts an extra behavior after the execution of method `m` to check its normal and exceptional postconditions. The two kinds of postconditions are checked in two parts inside a `try/catch` block within the around advice. Within the `try` block and after the call to the `proceed` method, the code ⟨⟨*checkNormalPostcondition*⟩⟩ (see Figure 3.11) checks normal postconditions. The code in Figure 3.11 represents the evaluation of each local normal postcondition (`\old(`P_i`)` $\Rightarrow$ Q_i) and conjoins the results. The Java expression (`!`P_i `||` Q_i) represents the translation of the JML implication (`\old(`P_i`)` $\Rightarrow$ Q_i) used in postconditions. Moreover, as mentioned above, the notation $[\![v_i{:=}old\$v_i]\!]$ is responsible for retrieving and properly using the old values that can occur in P_i and Q_i. On the other hand, within the `catch` block, when the called method returns abnormally by throwing an exception, two code chunks are generated: (1) ⟨⟨*rethrowJMLException*⟩⟩; and (2) ⟨⟨*checkExceptionalPostcondition*⟩⟩. The code in Figure 3.12 checks whether the exception thrown is an instance of `JMLInternalNormalPostconditionError`. If it is,

```
⟨⟨staticAroundAdviceForPostconditionChecking⟩⟩≡
   let ⟨Ex_1,...Ex_s⟩ = Ex in
   let ⟨T_1,Id_1,...T_z,Id_z⟩ = FP in
   let ⟨T_1,...T_z⟩ = PTN in
   let ⟨Id_1...Id_z,⟩ = PIN in
   RT around (T_1 Id_1,...,T_z Id_z) throws Ex_1,...Ex_s :
     execution (static RT TN.MN(T_1,...,T_z)) &&
     args (Id_1,...,Id_z) {
        RT rac$result; // represents the return of the method
        try{
          ⟨⟨saveAllOldValues⟩⟩
          try{
            // executing the original method
            rac$result = proceed(current, Id_1,...,Id_z);
            ⟨⟨checkNormalPostcondition⟩⟩
          } catch (Throwable rac$e){
              ⟨⟨rethrowJMLException⟩⟩
              ⟨⟨checkExceptionalPostcondition⟩⟩
          }
        } catch (Throwable rac$cause){
            throw new JMLEvaluationError(rac$cause);
        }
     }
```

Figure 3.9: AspectJ `around` advice to check postcondition of the static method *MN*.

```
⟨⟨saveAllOldValues⟩⟩≡
   old$v_1 := v_1;
      ...
   old$v_f := v_f;
```

Figure 3.10: Code for saving JML old values.

then the `JMLInternalNormalPostconditionError` is rethrown (because the exception is a part of a JML violation and must be kept).

If the exception thrown is not a `JMLInternalNormalPostconditionError`, then the code chunk ⟨⟨*checkExceptionalPostcondition*⟩⟩ in Figure 3.13 evaluates each local $\boldsymbol{R}_i$ and conjoins the results. Note that for each $\boldsymbol{R}_i$, its corresponding exception type X_{ij} must match the current exception thrown. The variable `rac$v` contains the result of the conjunction of all exceptional postconditions. If it is `false`, a `JMLInternalExceptionalPost-condition` must be thrown; otherwise the current exception must be rethrown.

Concerning postcondition inheritance (discussed in Section 2.1.6), the absence of the AspectJ designator `within` (present in `before` advice to handle preconditions) makes the advice affect executions of *MN* in subtypes of *TN*. Consequently, all inherited postconditions (normal and exceptional postconditions) will be automatically checked in a conjoined way, respecting Leavens' definition [38, Definition 1].

It is important to notice that the two kinds of postconditions use the same `around` advice and our approach generates the assertion checking code only if necessary. For example, if a method has omitted postconditions clauses (normal or exceptional) or an

```
⟨⟨checkNormalPostcondition⟩⟩≡
  if(!((!(P_1[[v_i:=old$v_i]] || Q_1[[v_i:=old$v_i]]) &&
 ... && (!(P_k[[v_i:=old$v_i]] || Q_k[[v_i:=old$v_i]]))){
   throw new JMLInternalNormalPostconditionError();
 }
```

Figure 3.11: Code for checking normal postconditions of method *MN*.

empty specification case, then no instrumentation code is generated. On the other hand, if a method contains only normal postconditions, only a normal postcondition checking code will be generated.

Static method postcondition translation

In a previous section we presented the translation rules for postconditions. Such rules deal only with non-static method postconditions. Now, we will assume that the method `m` is static.

The code chunk in Figure 3.9 represents the AspectJ `around` advice used to check the two kinds of JML postcondition clauses in the static method *MN*. Similar to preconditions, we have removed the advice parameter `current` and the designator `this`. They were eliminated because we do not want to expose the current executing object properties — in particular properties of the object *TN*. Concerning the rest of the `around` advice works as explained before.

3.2.3 Invariant translation

Invariants are predicates that must hold after constructor execution, and before and after every method execution, even when a method throws an exception. In JML, we can have both static and instance invariants. The auxiliary functions $MAP_{InstanceInvariantBMEToAspectJCode}$ and $MAP_{InstanceInvariantAMEToAspectJCode}$ (Appendix D.1.4 presents the definition of these auxiliary functions) denote the translation of the instance invariants present in *TN* into AspectJ program code. As a result, an AspectJ `before` advice is generated to check the instance invariants before every method execution. Moreover, the two kinds of after advice: (1) `after returning`; and (2) `after throwing` advice are used to check all instance invariants when a method terminates: (1) normally or (2) abnormally. In JML even when a method throws an exception, the invariant check must be performed. The clause (`!static * \textit{TN}.*(..)`), defined by the AspectJ designator `execution`, specifies that the advice becomes applicable to all non-static methods.

```
⟨⟨rethrowJMLException⟩⟩≡
  if(rac$e instanceof JMLInternalNormalPostconditionError){
    throw (JMLInternalNormalPostconditionError) rac$e;
  }
```

Figure 3.12: Code for rethrowing the JML exceptions.

```
⟨⟨checkExceptionalPostcondition⟩⟩≡
  boolean rac$v = true;
  boolean $rac\$pre_1$ = $P_1[\![v_i\text{:=}old\$v_i]\!]$;
  if(rac$v && $rac\$pre_1$){
    if(rac$e instanceof $X_{11}$){
      boolean $flag_1$ = true;
      $X_{11}$ $e_{11}$ = ($X_{11}$)rac$e;
      $flag_1$ = $R_1[\![v_i\text{:=}old\$v_i]\!]$;
      rac$v = rac$v && $flag_1$;
     }
     ...
  }
   ...
  boolean $rac\$pre_k$ = $P_k[\![v_i\text{:=}old\$v_i]\!]$;
  if(rac$v && $rac\$pre_k$){
    if(rac$e instanceof $X_{k1}$){
      boolean $flag_k$1 = true;
      $X_{k1}$ $e_{k1}$ = ($X_{k1}$)rac$e;
      $flag_{k1}$ = $R_k[\![v_i\text{:=}old\$v_i]\!]$;
      rac$v = rac$v && $flag_{k1}$;
    }
     ...
  }

  if(!rac$v){
    throw new JMLInternalExceptionalPostconditionError();
  }
  else{
    if(rac$e instanceof $X_{11}$){
      throw ($X_{11}$) rac$e;
    }
     ...
    if(rac$e instanceof $X_{kj}$){
      throw ($X_{kj}$) rac$e;
    }
  }
```

Figure 3.13: Code for checking exceptional postconditions of method *MN*.

The code in Figure 3.14 represents the general struture of the AspectJ `before` advice to check all local instance invariants of the class *TN*. In addition, the code in Figure 3.15 checks the invariants after every method execution (with or without normal termination). The `before` and `after returning` advices check the instance invariants through the code chunk ⟨⟨*checkInstanceInvariant*⟩⟩ in Figure 3.16. The code in Figure 3.16 represents the evaluation of each local instance invariant $InstInv_i$ and conjoins the results. If the instance invariants do not hold, a `JMLInvariantError` is raised. Moreover, if a method terminates by throwing an exception, the code chunk ⟨⟨*rethrowJMLException*⟩⟩ in Figure 3.17 is generated. If the exception thrown is an instance of any JML violation error, this code rethrows it. Otherwise the invariant check must be performed (recall the code defined by the chunk ⟨⟨*checkInstanceInvariant*⟩⟩ — see Figure 3.16). If the invariants hold, then the exception thrown is transparently preserved.

Concerning instance invariants inheritance (discussed in Section 2.1.6), as with the

```
$MAP_{InstanceInvariantBMEToAspectJCode}[\![TN, Inst\_Inv]\!] \triangleq$
  if Inst_Inv ≠ {}
  then let {InstInv_1 && ... && InstInv_r} = Inst_Inv
       before ( TN current):
         execution (!static * TN.*(..)) &&
         this (current) {
            ⟨⟨checkInstanceInvariant⟩⟩
         }
```

Figure 3.14: General structure of the AspectJ `before` advice for instance invariant check.

```
$MAP_{InstanceInvariantAMEToAspectJCode}[\![TN, Inst\_Inv]\!] \triangleq$
 if Inst_Inv ≠ {}
  then let {Inst_Inv_1 && ... && Inst_Inv_r} = Inst_Inv
       after ( TN current) returning (Object o):
         execution (!static * TN.*(..)) &&
         this (current) {
            ⟨⟨checkInstanceInvariant⟩⟩
         }
       after ( TN current) throwing (Throwable rac$thrown):
         execution (!static * TN.*(..)) &&
         this (current) {
            ⟨⟨rethrowJMLException⟩⟩
            else {
               ⟨⟨checkInstanceInvariant⟩⟩
            }
         }
```

Figure 3.15: General structure of the two AspectJ `after` advices for instance invariant check.

`around` advice for postconditions, the absence of `within(S)` makes the advice affect executions of all non-static methods in subtypes of `S`. In this way, all inherited instance invariants will be automatically checked in a conjoined way, respecting Leavens' definition [38, Definition 2].

Static invariant translation

The auxiliary functions $MAP_{StaticInvariantBMEToAspectJCode}$ and $MAP_{StaticInvariantAMEToAspectJCode}$ (Appendix D.1.4 presents the definition of these auxiliary functions) denote the translation of the instance invariants present in TN into AspectJ program code. As a result, the AspectJ advices `before` and the two kinds of after advices are used. The `before` advice checks all static invariants before every method (static and non-static) execution (see Figure 3.18). The `after returning` and `after throwing` advices checks all static invariants when a method terminates normally or abnormally (see Figure 3.19). In both Figures, the AspectJ designator `execution` defines the clause (`*` TN`.*(..)`), which becomes applicable to all static and non-static

```
⟨⟨checkInstanceInvariant⟩⟩≡
 if (!(Inst_Inv_1 && ... && Inst_Inv_r)){
   throw new JMLInvariantError();
 }
```

Figure 3.16: Code for checking instance invariants.

```
⟨⟨rethrowJMLException⟩⟩≡
 if (rac$thrown instanceof JMLInternalPreconditionError) {
   throw (JMLInternalPreconditionError) rac$thrown;
 }
 ...
 else if (rac$thrown instanceof JMLInvariantError) {
   throw (JMLInvariantError) rac$thrown;
 }
```

Figure 3.17: Code for rethrowing the JML exceptions for instance invariants.

methods — in JML static invariants must be satisfied for both static and non-static methods.

The `before` and `after returning` advices check the static invariants of *TN*. Figures 3.18 and 3.19 both present the code chunk ⟨⟨*checkStaticInvariant*⟩⟩ for checking static invariants. The code in Figure 3.20 evaluates each local static invariant $StatInv_i$ and conjoins the results. If the static invariants do not hold, a `JMLInvariantError` is raised. In addition, if a method terminates by throwing an exception, the code represented by the chunk ⟨⟨*rethrowJMLException*⟩⟩ is generated. This code is the same as presented in Figure 3.17 for instance invariants. Thus, if an exception is thrown and it is an instance of any JML violation errors, then the generated code by the chunk ⟨⟨*rethrowJMLException*⟩⟩ rethrows it. Otherwise the static invariant check must be performed (recall the code in Figure 3.20 — chunk ⟨⟨*checkStaticInvariant*⟩⟩). If the static invariants are preserved, then the exception thrown is kept (transparently).

Invariant translation for interfaces

Interfaces are provided to be extended by another interface or implemented by a class. Thus, the translation rules (describe previously) for instance invariants, as expected, affects subtypes automatically. However, static invariants are only preserved by the type that declares them (this scenario is applied for classes and the translation rules also deals with). But, there is only one situation when static invariants should be inherited, when they are declared by an interface and repassed to types that implement it. To allow a static invariant to be inherited, we use the AspectJ wildcard `+` — by using this, its possible to apply the static invariants declared into interfaces to be checked in types that implement it (Figures 3.18 and 3.19 show the use of the AspectJ wildcard `+` responsible to allow static invariants inheritance).

It is important to notice that our translation rules provide support for instance and static invariants specified into interfaces. However, as mentioned in Section 2.1.5 [Discussion], the current jmlc compiler does not handle instance invariants or static

```
$MAP_{StaticInvariantBMEToAspectJCode}[\![TN, TMod, Stat\_Inv]\!] \triangleq$
  if $Stat\_Inv \neq \{\}$
  then let $\{Stat\_Inv_1$ && ... && $Stat\_Inv_s\}$ = $Inst\_Inv$
       if 'interface' $in$ $TMod$
       then
          before ():
            execution (* $TN$+.*(..)) {
               $\langle\langle checkStaticInvariant \rangle\rangle$
            }
       else
          before ():
            execution (* $TN$.*(..)) {
               $\langle\langle checkStaticInvariant \rangle\rangle$
            }
```

Figure 3.18: General structure of the AspectJ `before` advice for static invariant check.

invariants when declared into interfaces. Invariants and inheritance is an open issue [43, Section 8.2.4].

3.2.4 Ordering of advice executions into an aspect

One AspectJ aspect can have several advices (e.g., `before`) to apply to a particular named or anonymous pointcut. As the advices are declared into the same aspect, we should take into account their order declaration. In this way, the advice that appears first lexically inside the aspect executes first. "The only way to control precedence between multiple advice in an aspect is to arrange them lexically [37]." Thus, our translation rules generate AspectJ advices carefully in order to respect the JML semantics. The translation function $MAP_{JmlToAspectJAspect}$ calls the discussed auxiliary functions in the following order to fulfil the JML semantics to check contracts:

1. $MAP_{StaticInvariantBMEToAspectJCode}$;
2. $MAP_{InstanceInvariantBMEToAspectJCode}$;
3. $MAP_{PreconditionToAspectJCode}$;
4. $MAP_{PostconditionToAspectJCode}$;
5. $MAP_{InstanceInvariantBMEToAspectJCode}$;
6. $MAP_{StaticInvariantAMEToAspectJCode}$.

The above ordering to generate AspectJ code is extremely important to keep the classical ordering of contract checking posed by JML. For example, a method to be executed must obey some conditions in a certain order:

1. check invariants (static and instance invariants) before method execution;
2. check preconditions before method execution;

```
$MAP_{StaticInvariantAMEToAspectJCode}$[[TN, TMod, Stat_Inv]] $\triangleq$
  if Stat_Inv $\neq$ {}
  then let {Stat_Inv$_1$ && ... && Stat_Inv$_s$} = Inst_Inv
       if ‘interface’ in TMod
       then
          after () returning (Object o):
            execution (* TN+.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
          after () throwing (Throwable rac$thrown):
            execution (* TN+.*(..)) {
               ⟨⟨rethrowJMLException⟩⟩
               else {
                  ⟨⟨checkStaticInvariant⟩⟩
               }
            }
       else
          after () returning (Object o):
            execution (* TN.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
          after () throwing (Throwable rac$thrown):
            execution (* TN.*(..)) {
               ⟨⟨rethrowJMLException⟩⟩
               else {
                  ⟨⟨checkStaticInvariant⟩⟩
               }
            }
```

Figure 3.19: General structure of the two AspectJ `after` advices for static invariant check.

3. check postconditions after method execution (normal postconditions when the method terminates normally and exceptional postconditions when the method terminates abnormally);

4. check invariants (static and instance invariants) after method execution.

These ordering are respected by the translation rules that generate AspectJ code to check JML contracts during runtime (as shown above).

```
⟨⟨checkStaticInvariant⟩⟩≡
 if (!(Stat_Inv$_1$ && ... && Stat_Inv$_s$)){
   throw new JMLInvariantError();
 }
```

Figure 3.20: Code for checking all non-static invariants.

```
$MAP_{JmlToAspectJAspect}[\![$Annotated_Java_Type$]\!]$ $\triangleq$
  let TN, TMod, Meth, SCN, SIN,
          Inst_Inv, Stat_Inv = Annotated_Java_Type in
  let $StatInvBefore$ = $MAP_{StaticInvariantBMEToAspectJCode}[\![$Stat_Inv$]\!]$ in
  let $InstInvBefore$ = $MAP_{InstanceInvariantBMEToAspectJCode}[\![$Inst_Inv$]\!]$ in
  if Meth $\neq \langle\epsilon\rangle$
  then let $\langle Meth_1, \ldots Meth_m\rangle$ = Meth in
       let $TMod_1$, $MN_1$, $FP_1$, $PTN_1$, $PIN_1$, $RT_1$, $Ex_1$, $SC_1$ = $Meth_1$ in
                                   ...
       let $TMod_m$, $MN_m$, $FP_m$, $PTN_m$, $PIN_m$, $RT_m$, $Ex_m$, $SC_m$ = $Meth_m$ in
       let $Precondition_1$ = $MAP_{PreconditionToAspectJCode}[\![TN, TMod, SCN, SIN, MN_1,$
                                         $MMod_1, FP_1, PTN_1, PIN_1, RT_1, SC_1]\!]$ in
                                   ...
       let $Precondition_m$ = $MAP_{PreconditionToAspectJCode}[\![TN, TMod, SCN, SIN, MN_m,$
                                       $MMod_m, FP_m, PTN_m, PIN_m, RT_m, SC_m]\!]$ in
       let $Postcondition_1$ = $MAP_{PostconditionToAspectJCode}[\![TN, MN_1, MMod_1, FP_1,$
                                         $PTN_1, PIN_1, RT_1, Ex_1, SC_1]\!]$ in
                                   ...
       let $Postcondition_m$ = $MAP_{PostconditionToAspectJCode}[\![TN, MN_m, MMod_m, FP_m,$
                                         $PTN_m, PIN_m, RT_m, Ex_m, SC_m]\!]$ in
  let $InstInvAfter$ = $MAP_{InstanceInvariantAMEToAspectJCode}[\![$Inst_Inv$]\!]$ in
  let $StatInvAfter$ = $MAP_{StaticInvariantAMEToAspectJCode}[\![$Stat_Inv$]\!]$ in
  if Meth $\neq \langle\epsilon\rangle$
  then let $AspectJCode$ = $StatInvBefore \cup InstInvBefore \cup$
       $\langle Precondition_1, Postcondition_1\rangle$ ... $\cup \langle Precondition_m, Postcondition_m\rangle$
       $\cup$ $InstInvAfter \cup StatInvAfter$ in
  else let $AspectJCode$ = $StatInvBefore \cup InstInvBefore \cup InstInvAfter$
       $\cup$ $StatInvAfter$ in
  if $AspectJCode \neq \epsilon$
  then
     public privileged aspect AspectJMLRac$TN {
       $AspectJCode$
     }
```

Figure 3.21: Complete translation rules.

Complete translation rules

In order to see the complete translation rules of the JML specifications, we show in Figure 3.21 that the translation function $MAP_{JmlToAspectJAspect}$ obey the JML semantics.

3.3 Comparing jmlc with ajmlc

This section describes some differences between the original JML compiler (jmlc) and our compiler (ajmlc).

Compiling empty classes

The jmlc compiler assumes a standard configuration for classes. Thus, even if one defines an empty class, basic instrumentation is generated [39, 43] for:

1. class verification

JML clauses	jmlc generates	ajmlc generates
requires	yes	yes
ensures	yes	no
signals	yes	no
invariant	yes	no

Table 3.1: Difference between jmlc and ajmlc during the generation code.

- Static and non-static invariant/constraint checking;
- Static and non-static constraint pre-state expressions checking.

2. default constructor verification
   - Assertion checking wrapper;
   - Precondition checking;
   - Normal postcondition checking;
   - Exceptional postcondition checking.
3. other methods (e.g., for dynamic calls using reflection)

In this way, the jmlc compiler generates 5.98 KB even for a empty class like:

```
public class Empty { }
```

In contrast to jmlc compiler, our compiler does not generate code for empty classes.

Code instrumentation

Code size is an important issue for Java ME applications. Our compiler avoids code generation as much as possible. Table 3.1 compares the jmlc and ajmlc compilers when no specification is provided. The jmlc compiler generates code for every JML clause. It does not matter if the specification is not defined. The ajmlc compiler generates code for an empty specification only in the case of preconditions of instance methods. To implement inheritance of specification, when the method is overridden, the local precondition (which is empty) must be disjointed with the preconditions defined for the method in the superclasses (see Section 3.2.1 for more details).

Limitations of the jmlc compiler solved by the ajmlc compiler

As mentioned previously (Sections 2.1.4 and 2.1.5), the jmlc compiler has two limitations:

1. When constrained methods are called into static blocks during the class initialization, jmlc does not check the constrains and the method is always executed even if the condition is `false`;
2. Static and non-static invariants are not checked when declared into interfaces.

Concerning the first limitation, properties (e.g., preconditions, invariants) of constrained methods are automatically verified when called into static blocks. Regarding the second problem, as explained in Section 3.2.3, the code generated by the ajmlc compiler checks only instance invariants declared in interfaces.

The predicate same

Another enhancement of our compiler is the proper translation of the JML predicate `same`. The predicate `same` is denoted by the JML keyword `\same`. It is used in preconditions clauses. The predicate is the same of the precondition specified in other (non-same) specification case. Only one `requires` clause is allowed for each specification case. Moreover, if a single specification case is defined, the predicate `\same` can be used only if the method is overridden. As a consequece, since specifications of constructors and static methods cannot be inherited, the predicate `\same` cannot be defined if a single specification case is provided.

According to a bug reported to the JML mail list[2], the jmlc translates the predicate `same` into `false`. Our compiler fixed this bug, by generating AspectJ code to reflect the semantics of the predicate `same`. Appendix C.2 presents an example of specification case with the `same` predicate and the corresponding AspectJ code.

Implementation of JML clauses

If compared with ajmlc, the original JML compiler [14] provides support for a larger subset of the JML language. For instance jmlc can handle abstract specifications and quantifiers, which are not treated yet by ajmlc.

3.4 Chapter Summary

In this chapter we presented the motivation for implementing the ajmlc compiler. In addition, we described the rules to translate JML annotations into AspectJ. The translation rules consider preconditions, postconditions (both normal and exceptional postconditions), and invariants (both static and instance invariants). We have also compared the implementation of the ajmlc compiler against the original JML compiler (jmlc).

[2]The bug report of the same predicate can be seen at `http://sourceforge.net/mailarchive/forum.php?forum_name=jmlspecs-developers`

Chapter 4

Proofs of Concept

In order to evaluate the overhead in code size produced by our compiler, we conducted some studies comparing the code produced by the original JML compiler (jmlc) with the code generated by our compiler (ajmlc). Such analysis is specially important for Java ME applications.

The first study compares both compilers using a Java ME application. The code generated by jmlc is not compliant with Java ME, but this study is only interested in the code size metrics — because the code size is the most important issue when dealing with constrained environments such as Java ME. The second study evaluates the compilers using three different Java SE applications. This study is relevant to evaluate the overhead in code size of our compiler using different AspectJ weavers.

4.1 Study 1: Java ME application

To evaluate our compiler (ajmlc) in environments of resource-constrained devices, we employed a Java ME floating point calculator(MiDlet application [56, 63]) — we choose it because it is a simple open source Java ME application available by Sun web site[1], and because there is no Java ME applications annotated with JML; thus we used it to annotate with JML specifications — with three different scenarios: using our compiler ajmlc (with ajc weaver) (*CalcAspSol*); using the original compiler (jmlc) [14] (*CalcJml-Sol*); and the pure Java compiler, with no bytecode instrumentation (*CalcPureSol*). We compared the three scenarios by analyzing the metrics: MiDlet class size (bytecode instrumentation); JAR size; library API size. Moreover, we compared the three scenarios with the three mentioned metrics, before and after obfuscated. Obfuscation after compilation is a standard practice adopted by industry (we also compared the JAR size metric by using the open source obfuscator ProGuard [71]).

The Java ME floating point calculator application presents a calculator screen where the operands and operations are requested and the result shown. We annotated it with JML constructs — which are fully supported by our compiler — to ensure two properties: it yields only positive results; and it prevents division by zero.

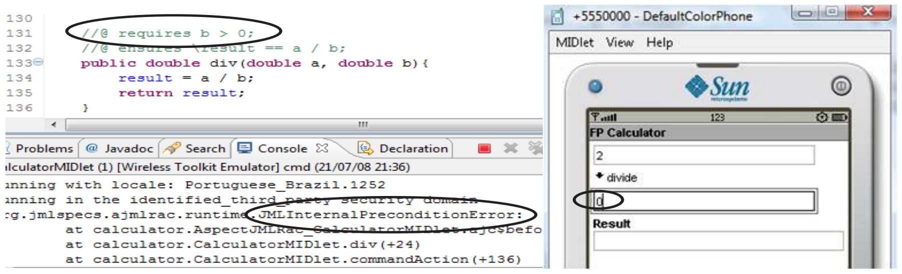
When we compile the *CalcJmlSol* version by using the JML compiler (jmlc) setting the class path to the Java ME API [56, 63], the bytecodes we obtain do not pass the

[1] An open source Java ME application available at `https://meapplicationdevelopers.dev.java.net/demo_box.html`

*Denotes a obfuscated measurement

	CalcAspSol x CalcJmlSol			**CalcAspSol x CalcPureSol**		
	CalcAspSol (KB)	**CalcJmlSol** (KB)	**Ratio** (%)	**CalcAspSol** (KB)	**CalcPureSol** (KB)	**Ratio** (%)
MidLet class size	54.0	49.8	≈ 7.8	54.0	5.5	≈ 89.8
JAR size	14.3 5.4*	80.4 26.2*	≈ -82.2 ≈ -79.3*	14.3 5.4*	3.4 2.6*	≈ 76.2 ≈ 51.8*
Lib JAR size	6.8	46.5	≈ -85.3	6.8	—	—

Table 4.1: Java ME calculator application statistics.



```
//@ requires b > 0;
//@ ensures \result == a / b;
public double div(double a, double b){
    result = a / b;
    return result;
}
```

```
lculatorMIDlet (1) [Wireless Toolkit Emulator] cmd (21/07/08 21:36)
unning with locale: Portuguese_Brazil.1252
unning in the identified_third_party security domain
rg.jmlspecs.ajmlrac.runtime.JMLInternalPreconditionError:
      at calculator.AspectJMLRac_CalculatorMIDlet.ajc$befo
      at calculator.CalculatorMIDlet.div(+24)
      at calculator.CalculatorMIDlet.commandAction(+136)
```

Figure 4.1: A precondition error in the Java ME calculator application.

analysis of the Java ME preverifier tool, which checks bytecode compatibility to run in the Java ME environment. The reason for this failure is that Java ME does not support all features present in Java SE. Despite the incompatibility, we use the code in our study only to analyze the bytecode instrumentation generated by the jmlc [14]).

Results

Table 4.1 presents the result of the analysis. Concerning JAR size, we observe that, *CalcAspSol* is 76.2% bigger than *CalcPureSol*, but only 51.8% bigger when considering obfuscated JAR sizes. Still considering JAR size, we can point out that, *CalcAspSol* is 88.2% smaller than *CalcJmlSol*, but 79.3% smaller when we consider obfuscated JAR sizes. For library API size, *CalcAspSol* showed to be -85.3% smaller than *CalcJmlSol*. This happens because *CalcAspSol* requires far less code than the original JML runtime library to execute instrumented bytecode. It is important to notice that we used only a part of the original JML runtime library, which is compatible to the subset of the JML that ajmlc handles (this leads to a fair comparison). In the case of the MiDlet class size, *CalcAspSol* is 89.8% bigger than *CalcPureSol* and also 7.8% bigger than *CalcJmlSol*. Such results provide evidence that our approach requires less memory space than the original JML compiler only when we take into account Jar or Lib size. Furthermore, this indicate us that we need to improve the JML contracts instrumentation, even though we gain during deployment (Jar application). As a proof of concept, we executed the calculator in a real mobile phone. We performed method calls with arguments that lead to precondition violation as specified by contracts. The application answered properly to these calls. Figure 4.1 illustrates a precondition violation when a division by zero is performed. Appendix E.1 contains the annotations for the Java ME calculator and the corresponding instrumentation code generated by our compiler.

	JML Constructs						
Annotated class	requires	ensures	signals	invariant	old	also	heavyweight
Animal	X	X			X	X	
Person	X	X				X	
Patient	X	X		X	X		
IntMathOps	X	X					X
StackAsArray	X	X	X		X		X

Table 4.2: JML constructs contained in each class.

		ajmlc	
	jmlc (KB)	**(ajc)** (KB)	**(abc)** (KB)
Animal	16.2	19.0	4.7
Person	14.0	19.3	4.8
Patient	16.7	19.9	6.1
IntMathOps	12.7	8.1	2.2
StackAsArray	23.6	29.6	6.1

Table 4.3: Java applications bytecode size results.

4.2 Study 2: Java SE applications

In this section we evaluate ajmlc using the Java SE environment with three Java applications extracted from the JML literature. These applications were chosen because they constitute all most the JML level 0 [43, Section 2.9] constructs (focus of this dissertation).

Scenario

The Java SE applications employed in this study are: (1) the hierarchy classes `Animal`, `Person`, and `Patient` [38]; (2) the class `IntMathOps` [39], and (3) the class `StackAsArray` [14]. Moreover, we have used our ajmlc with two different weaving processes: the standard AspectJ compiler (ajc) [36]; and the abc compiler [5], which is a complete implementation of AspectJ with some optimizations.

Table 4.2 shows the JML constructs that each annotated class addresses. For instance, a heavyweight keyword indicates that the class has a heavyweight specification (see Section 2.1.4 [Heavyweight and Lightweight specifications]). Moreover, the presence of the keyword `also` means that the class can have multiple specification cases or inherits from another type. In order to understand the meaning of the remaining JML constructs, see Sections 2.1.4 and 2.1.5.

Results

Tables 4.3 and 4.4 present the study results. We analyzed bytecode size (with instrumented code present) in kbytes (KB). Considering our compiler (ajmlc), we used the same AspectJ aspect code generated for both weaving processes (ajc and abc). The ajmlc compiler using the ajc weaver introduces a big overhead in the code size. By using

	ajmlc(ajc) x jmlc			ajmlc(abc) x jmlc		
	ajmlc (KB)	jmlc (KB)	Ratio (%)	ajmlc (KB)	jmlc (KB)	Ratio (%)
Animal	19.0	16.2	≈ 14.8	4.7	16.2	≈ -70.9
Person	19.3	14.0	≈ 27.5	4.8	14.0	≈ -65.8
Patient	19.9	16.7	≈ 16.0	6.1	16.7	≈ -63.5
IntMathOps	8.1	12.7	≈ -36.2	2.2	12.7	≈ -82.7
StackAsArray	29.6	23.6	≈ 20.2	6.1	23.6	≈ -74.2

Table 4.4: Java applications statistic results.

ajc weaver, only in one case (`IntMathOps` instrumentation), ajmlc generated a smaller bytecode than jmlc. This is due to JML specifications contained into a Java source file. In this way, since the `IntMathOps` constitutes few JML specifications (only one method annotated with JML), the instrumented bytecode generated by ajmlc is smaller than jmlc. In the remaining cases, when the JML specifications increase (several methods annotated with JML), the instrumented bytecode tends to be bigger than jmlc. On the other hand, our approach produces a far smaller code when the abc weaver is employed (due to optimizations present in the abc weaver). Therefore, we decided to allow users free to choose which AspectJ weaver to instrument the JML contracts. Thus, based on the results (see the ratios in the Table 4.4), we recommend the usage of ajmlc with the abc weaver. This choice is particulary important for Java ME applications. However, when dealing with Java SE applications both ajc or abc can be employed, since memory space in this applications are not quite constrained. Moreover, as ajmlc handles only a subset of the JML language, eventually for Java SE applications the usage of jmlc is suitable due to coverage of more JML features, such as model programs [39]. As a future work, optimizations in the abc compiler can provide a smaller code generation by ajmlc compiler (improving the generated instrumented bytecode in constrained environment required by Java ME applications).

In order to see the source code of the three annotated Java SE applications and the corresponding instrumentation code introduced by our compiler, please refer to Appendix E.2.

4.3 Chapter Summary

In this chapter we conducted proof of concepts used to validate our JML compiler — the ajmlc. We described two studies: (1) applying the jmlc and ajmlc compilers to a Java ME application to validate the ajmlc compiler in a Java ME environment, and compare overhead in code size produced by ajmlc and jmlc; and (2) applying again both compilers in a Java SE environment using three Java applications. These applications already appeared in the JML literature. For the first study, we used only the AspectJ ajc compiler, and then compared the size of the instrumented bytecode provided by the ajc weaver. The results pointed out that the generated code can be used with Java ME applications, but the overhead in instrumented bytecode produced is bigger than the standard jmlc. On the other hand, in the second study, we employed two AspectJ compilers (ajc and abc) and compare the size of the instrumented bytecode inserted by these AspectJ weavers. Such results provide an evidence that our approach generates smaller code than the original JML compiler when using the abc AspectJ weaver, which is important for Java ME applications. Despite the size of the instrumented code generated, the ajmlc with both AspectJ compilers (ajc and abc) can be used with Java SE as well as Java ME applications.

Chapter 5

Related Work

In this chapter, we present the main works related to this dissertation.

5.1 JML-based

In this section, we present the related works which are JML-based.

5.1.1 A Runtime Assertion Checker for the Java Modeling Language

Cheon presents in his doctoral thesis [14] a compiler for runtime assertion checking designed for Java Modeling Language (JML) [39, 43]. It is a runtime assertion checking compiler, which converts JML annotations into runtime checks. It is one of the JML tools [11] developed at Iowa State University. The JML compiler (also known as *jmlc*) adopts Java's reflection facility [54] to support specification inheritance. Moreover, Cheon defines a set of translation rules from JML predicates into Java program code. The translation rules handle various kinds of specifications and expressions, such as method specifications and old expressions. As with Cheon, in this dissertation we define a set of of translation rules from JML predicates into AspectJ program code. The main reason to use AspectJ code is to automatic solve the problem of code instrumentation. Instead of using the Cheon's wrapper approach [14], we delegate this task to the AspectJ weaver. In this way, we also gain in modularity (separation of concern related to contract enforcement) and make the ajmlc ease of modification or extension to treat other JML features, such as model programs [39]. Moreover, unlike jmlc compiler, ajmlc with AspectJ generates instrumented bytecode compliant with Java SE and also Java ME programs. In addition, ajmlc generates an instrumented code that checks constrained methods when they are called into static blocks (class initialization). Such verification is not performed by the instrumented code produced by jmlc. Additionally, different from jmlc the bytecode produced by our compiler generates invariant checks when declared into interfaces. Regarding old expressions, Cheon's approach evaluates all pre-state expressions and stores their results as private fields and, in addition, uses a private stack per method in order to save and restore pre-state values to handle nested assertion checking (recursive calls). In contrast, we only define local variables for the AspectJ advice to handle old expressions, even in the presence of nested assertion checking. This approach is semantically equivalent to Cheon's approach, but by using our approach to handles old expressions is more intuitive and simple than Cheon's. Every recursive call made to any method has its context saved by using AspectJ `around` advice (which has total control under the constrained method, before and after its execution). Finally, our compiler handles a subset of the JML language, whereas the jmlc compiler handles more JML constructs and features, such as abstract specifications.

5.1.2 JCML - Java Card Modeling Language

JMLC (Java Card Modeling Language) [59] is a subset of the JML language. The JCML compiler (jcmlc) generates bytecode compliant with Java Card applications. The motivation of that work is similar to ours. It tackles some limitations of the jmlc compiler to implement Java Card applications. For instance, Java Card does not support the Java reflection mechanism. The jcmlc does not employ AspectJ to instrument JML contracts. It translates the JML specifications into *verification functions*. These verification functions generated by the jcmlc preserve as much as possible the structure of the JML compiler based on *wrapper approach* [14].

In addition, different from our strategy, the jcmlc does not reuse the front-ent of jmlc (the JML type checker) [11]. The jcmlc was designed using the JavaCC (Java compiler compiler) tool [3], which constitutes its front-end and back-end. As a consequence, some verifications performed by the JML type checker (e.g., assignable clause and pureness) are not implemented [11].

Regarding JML specifications, the jcmlc translates only some JML lightweight specifications, in particular only code for instance invariants and preconditions are considered in the first version of the jcmlc compiler. In contrast to jcmlc, our compiler handles both lightweight and heavyweight specifications as well as the two kinds of invariants in JML (static and instance invariants). The jcmlc does not support inheritance of specifications, which our compiler does. On the other hand, the jcmlc handles quantifiers such as forall, which are not treated by ajmlc.

5.1.3 Pipa - A Behavioral Specification Language for AspectJ

Pipa [76] is a behavioral interface specification language (BISL) tailored to AspectJ. It uses the same approach (based on annotations) of JML language to specify AspectJ classes and interfaces, and extends JML with a few new constructs in order to specify AspectJ programs. The Pipa language also supports aspect specification inheritance and crosscutting. Pipa specifies AspectJ programs with pre-, postconditions, and invariants. Moreover, Pipa also can specify aspect invariants and the "decision" whether or not to call the proceed method within the around advice (using the `proceed` extended annotation). The aim in designing Pipa based on JML is to reuse the existing JML-based tools. In order to make this possible the authors developed a tool (compiler) to automatically transform an AspectJ program with Pipa specifications into a standard Java program with JML specifications. To this end, the authors modified the AspectJ compiler (ajc) to retain the comments during the weaving process. After the weaving process, all JML-based tools can be applied to AspectJ programs. Therefore, the main goal of Pipa is to facilitate the use of JML language to verify AspectJ programs. On the other hand, our aim is to use AspectJ to implement JML contracts and verify Java programs.

5.2 Non JML-based

In this section, we discuss other related works, but they are not JML-based.

5.2.1 Jose: Aspects for Design by Contract

Feldman [24] presents a design by contract (DBC) [51] tool for Java, know as *Jose*. This tool adopts a private DBC language for expressing contracts. Similar to our, Jose adopts AspectJ for implementing contracts. The semantics of postconditions and invariants in Jose are distinct from JML. Jose states that postconditions are simply conjoined without taking into account the corresponding preconditions. Moreover, it establishes that private methods can modify invariant assertions. In the JML semantics, if a private method violates an invariant, an exception must be thrown. Concerning recursive assertion checking, Jose, like Eiffel [52], only allows one level of assertion checking. To enforce this policy, Jose uses the control-flow based pointcut (using the AspectJ designator `cflowbelow`). In this way, unlike our compiler, Jose generates bytecode not compliant with Java ME because it uses some AspectJ constructs that make references to Java SE APIs, which are not supported by the Java ME environment. In contrast to Jose, our compiler carefully uses AspectJ constructs that generates instrumented bytecode compliant with both Java SE and Java ME applications.

5.2.2 Contracts as an Aspect

Briand [10] discusses how to implement contracts with Aspect-Oriented Programming (AOP) using AspectJ. He defines how to efficiently instrument contracts and invariants in Java. The two main objectives of this work are: (1) to work at bytecode level avoiding polluting the source code; (2) to apply the Liskov Substitution Principle (LSP) [46] in order to check inheritance hierarchies. Similarly to our work, it presents AspectJ templates that map contracts and invariants into AspectJ *aspects* that represent them. These aspects provide an instrumentation extremely valuable during runtime to detect failures and during maintenance phase to help in locating faults (debugging).

Recalling JML semantics, the postcondition is valid only if the precodition holds in pre-state ($\texttt{\textbackslash old}(pre) \Rightarrow pos$). Unlike JML, the semantics for postconditions does not check the precondition in pre-state. Moreover, when the intercepted method throws an exception, the postcondition is not verified. Such verification is required to provide the JML mechanism to treat *exceptional postconditions* — properties that must be true when the method throws an exception. Thus, it only supports *normal postconditions*, while our JML compiler (based on JML constructs) supports both kinds of postconditions.

5.2.3 Contract4J: A Design by Contract Tool for Java

Contract4J [75] is another AspectJ implementation of contracts. It is an open-source tool that supports design by contract [51] for Java. Like JML, Contract4J allows programmers to specify contracts as annotations. These annotations are based on Java 5 annotations [55] (annotation-based approach). Moreover, if any condition (contract) is not satisfied, the program throws a runtime error and stops the program execution.

The @`Contract` annotation signals that a class has a contract specification defined. Furthermore, Contract4J employs the @`Pre`, @`Post`, and @`Invar` annotations that indicate, respectively, a precondition, postcondition, and invariant test. The Contract4J tool also employs the use of `$old` annotation, which represents the evaluation of old expressions in pre-state (before method call) — it is applied to state variables (attributes) that are used in post-state (after method call) within the postcondition test. The annotation-based approach (annotations) is interpreted and converted into AspectJ aspects (responsible for instrumenting and verifying the contracts during runtime).

Besides the annotation-based approach, the author of the Contract4J provides a second experimental syntax that uses a JavaBeans-style naming convention (method-based approach), which he called "ContractBeans". According to this style, the precondition and postcondition tests for a method named `add`, for example, are respectively written as `preAdd` and `postAdd`. (Compare with the JavaBeans convention for defining a `getResult` method for the `result` field present in a class.) This implementation approach is also based on AspectJ and has a significant runtime overhead, because it uses runtime reflection to discover and invoke the tests (when present). In addition, the work mentions another two drawbacks when using the method-based approach: (1) if the tests are not declared with public visibility, they are not visible for clients; (2) if the tests are not written in a proper JavaBeans-like convention, the tests are ignored. This happens because there is no mechanism in the Contract4J tool to warn the user. With relation to contract support, the method-based approach does not yet support `old` expressions when compared with the annotation-based approach.

Concerning the supported kinds of assertions and their implementation in AspectJ, the work covers only pre-, postcondition (normal), and invariant (instance) when compared with ours. Moreover, in contrast to our work, there is an important issue not covered by Contract4J — the current version does not provide support for inherited contracts — contravariance (used in precondition inheritance mechanism) and covariance (used in postcondition and invariant inheritance) behavior. Nevertheless, the Contract4J tool imposes at least the same contract conditions on derived classes. According to the author, he is planning to work on inheritance of contracts and release a new version[1] soon.

5.2.4 Applying Design by Contract on a framework using AspectJ

An experience in using the suitability of AspectJ for implementing contracts is discussed in work [45]. That work took an existing framework written in Java, the JWAM framework [9], and partially restructured it using AspectJ. The work points out that, with respect to the original implementation (of the framework) in plain Java, the use of AspectJ provides better support for incremental development, better reuse, and suitability in enforcing contracts in applications that use the framework. The JWAM framework is a Java-based object-oriented framework for interactive business applications, devel-

[1]The Contract4J is available on v0.8.0 version. New features will be released on version v1.0.0.

oped at the University of Hamburg. It contains more than 600 classes and interfaces — originally designed and implemented in Java using the design by contract technique [51] and was partially restructured [45] by that work using AspectJ. The framework uses contracts to ensure that callers — client programs — do not use the methods in a wrong way, likewise the implementation of methods must ensure their functionality as described in their specification. In order to signal any violation in the use of the framework, there is an exceptional behavior called `ContractBrokenException`. Besides broken contracts, there are other exceptional mechanism inside JWAM to signal any problem to the applications that use it.

The authors state that, by using AspectJ to partially re-implement the framework JWAM, they drastically reduced the code of the framework (LOC) and eliminated the tangled code referent for contract enforcement and exception detection and handling, thereby, providing a better reusability and high cohesion of the existing modules in the framework.

That work proved to be an excellent example in which AOP and AspectJ are effectively used to enforce the contract concern — the standard JML compiler (jmlc) and the ajmlc compiler (proposed here) would also be used to specify, verify, and ensure the functionalities of the framework described by that work. As with our work, it uses a specific language that expresses its contracts as annotations in the source code (in the framework classes), and these annotations are immediately targeted as AspectJ aspects. However, it differs from ours because it focused on only three kinds of assertions when it restructured the JWAM framework using AspectJ — pre-, postcondition (covering only normal postcondition), invariants (including only instance invariants). Moreover, it does not consider the inheritance of contracts and does not provide a mechanism for using old values in postconditions. Another differential of that work is that it is based on a specific framework (application), providing the reliability of the framework usage, whereas our work is based on constructing reliable Java applications — including applications of the Java ME platform.

5.2.5 Other non JML-based related works

Regarding contracts and aspects, Diotalevi [23] proposes an approach for adopting design by contract [51] in the development of Java application using Aspect-Oriented Programming (AOP) with AspectJ. The work states that inserting pre- and postconditions assertions directly into the application code has serious drawbacks — in terms of code modularity, reusability, and cohesion — that lead to a common OO limitation called tangled code. These assertions are *crosscutting concerns* and mix business logic code with the nonfunctional code that assertions require; they are inflexible because we cannot change or remove assertions without updating the application code. Because of that, the work provides a solution based on the following four requirements:

- transparency — the pre- and postconditions code is not mixed with business logic;
- reusability — most of the solution is reusable;
- flexibility — the assertion modules can be added to, removed, and modified;
- simplicity — assertions can be specified using a simple syntax.

This solution has a “bridge” that is an AspectJ aspect. This aspect specifies the exact point where the contract is to be applied. The AspectJ implementation of contracts covers pre-, postconditions and invariant checks. As a result, this solution provides a clean and flexible solution, because it eliminates the drawbacks previously mentioned. The solution lets one code the contracts of the application separately (untangled) from one’s business logic. Different from our approach, this work concentrates only on pre-, (normal) postcondition, and (instance) invariants.

“Assertion with Aspect” [33] proposes aspects for implementing assertions. The work aims to use AspectJ in order to inject `assert` statements into classes — the `assert` statement is a Java function of the standard library (available since JDK 1.4). Finally, this work enhances reusability of Java programs by eliminating tangled code with assert statements. As a result, programmers can add and remove assertions (contracts) of a class, thus enabling the programmers to separate (untangled) a class from its assertions. The proposed work differs from ours in that it covers only two types of assertions — precondition and (normal) postcondition assertions — and does not consider the inheritance of assertions.

Chapter 6

Conclusions

The Java Modeling Language (JML) is a formal language for specifying the behavior of Java modules, such as pre- and postconditions. These JML constructs are included as JML annotations within Java source code. The JML annotations are compiled into automatic runtime checks by the JML compiler (jmlc) [14], which are used during the program's execution and raising a JML exception when a condition (e.g, precondition) does not hold. However, the jmlc compiler does not work properly when applied to other Java dialects such as Java ME. The jmlc compiler generates instrumented bytecode that uses Java reflection mechanism, and data structures such as *HashSet* which are not supported by Java ME.

In this work we present the implementation of a new JML compiler known as ajmlc (AspectJ JML compiler). It translates programs annotated with JML into AspectJ program code. The result is a code with aspect checking methods that verify the program against JML specifications. This aspect code generated by our compiler is compliant with Java SE and Java ME applications. We mitigate the limitations of the jmlc compiler by avoiding AspectJ constructs that are not compliant with Java ME. For instance, the usage of the `cflow` pointcut would make the code incompatible with Java ME applications.

We also presented a set of translation rules (described in Chapter 3) that express how we deal with JML features, such as normal and exceptional postconditions, invariants, and old expressions. We compared ajmlc and jmlc, pointing out some improvements brought by our implementation when compared with jmlc. The ajmlc compiler generates instrumented bytecode to verify constrained methods within static blocks during the class initialization, which is not performed by the jmlc compiler; and the ajmlc generates instrumented code only when necessary, avoiding pollute the code with useless instrumentation, which probably increases its size. It is important to emphasize that our translation rules implemented by our compiler take into account only a subset of the JML language.

Concerning Tool support, a JML annotated Java source code can be used with: Esc/Java2 [25], Loop tool [73] (both perform static verification), and all tools which are part of the standard distribution of JML [11] (known as JML tools suite). With the approach (using aspects) discussed by this dissertation, static verification (using Esc/Java2 [25] or Loop tool [73]) can be used without problems, since they not require dynamic (runtime) checks. Regarding the JML tools suite [11], only the jmldoc and the jml type checker can be used. The jmlunit [15] tool cannot be used because it uses the standard JML compiler (jmlc) to generate checking methods for testing Java classes and interfaces.

We conducted proofs of concept in order to validate the implementation of our compiler (ajmlc) with Java SE and Java ME applications. We conducted two experiments: (1) to validate the ajmlc compiler when dealing with Java ME applications; (2) to validate the ajmlc compiler when dealing with Java SE applications. In these two experiments we analyzed the overhead in code size produced by our compiler in contrast to code size produced by the jmlc compiler. An another contribution was extracted from the proof of concept carried out with the Java SE applications. In this case we used two AspectJ weavers (ajc and abc) to generate the bytecode. We observed that the ajc weaver produce an important overhead in code size. On the other hand, the abc weaver produces a far smaller code when compared with jmlc. Although the user is free to use our compiler with the preferred AspectJ weaver, but based on the results, we recommend the combination of ajmlc and abc weaver to develop Java ME applications.

We believe that the usage of aspects to implement a JML compiler introduces a new level of modularity. In other words, our approach is not invasive (the Java source code is not tangled and scattered with the generated assertion methods to check JML contracts during runtime). This gives more flexibility to extend the compiler with other JML constructs and to optimize the current implementation. In addition, optimizations in the weaven process are automatically inherited by our compiler.

6.1 Future Work

A comprehensive survey [12] — mainly from industry — demonstrated that JML's assertion semantics, based on classical logic [14], does not satisfy the programmers' expectations. Recently, the JML community agreed to adopt new assertion semantics proposed by Chalin [68], which is closer to Java semantics, and is modeled on three-valued logic. The new assertion semantics was implemented in the last release of the JML compiler [1] (jmlc version 5.5), which is part of the JML tools [11]. In this way, as a future work, we intend to adapt the ajmlc to the new assertion semantics proposed by Chalin.

There are several extensions to the work presented in this dissertation. Most of them include support for more features of the JML language. Hence, we intend to augment the translation rules of our JML compiler (ajmlc) to support more features, such as quantifiers (JML level 0), and abstract specifications (model programs) [43]. In JML, one can also specify the behavior of methods by writing abstract code, known as *model programs*. Model programs are useful to keep away from implementation details; JML provides different types of specification to write model programs, such as specification-only fields (*model fields* from JML level 0), specification-only methods (*model methods* from JML level 1). We also intend to implement more JML type specifications such as history constraints [43] (JML Level 1). The JML language introduced several specification constructs to specify non-functional properties (JML level 2) like time and space requirements. Neither the current JML compiler (jmlc) nor our JML compiler (ajmlc) support non-functional properties. This is an exciting subject we plan to investigate.

[1]the JML 5.5 version is already available to download at http://sourceforge.net/projects/jmlspecs.

In relation to tool support, we plan to implement a new jmlunit [15] which uses the ajmlc compiler for generating test cases (with aspects). This issue would leads to a complete usage of the JML tools suite [11], even using our approach with AspectJ.

We also plan to address a problem suggested by Cheon [14]: to support assertion checking in a concurrent environment (e.g., multi-threaded program).

Finally, we intend to conduct more experiments using weavers that implement optimization techniques for AspectJ, including the works by Cordeiro [20] and Calheiros [47]. Such works provide some optimizations in the AspectJ abc compiler, which can improve the instrumented code generated by the ajmlc.

Appendix A

Java SE classes Supported by the Java ME platform

The following is a summary of the classes from Java SE that Java ME also supports.

A.1 System Classes

```
java.lang.Class    java.lang.Object  java.lang.Runnable (interface)
java.lang.Runtime  java.lang.String  java.lang.StringBuffer
java.lang.System   java.lang.Thread  java.lang.Throwable
```

A.2 Data Type Classes

```
java.lang.Boolean  java.lang.Byte  java.lang.Character
java.lang.Integer  java.lang.Long  java.lang.Short
```

A.3 Collection Classes

```
java.util.Enumeration (interface)  java.util.Hashtable
java.util.Stack                    java.util.Vector
```

A.4 Input/output Classes

```
java.io.ByteArrayInputStream  java.io.ByteArrayOutputStream
java.io.DataInputStream       java.io.DataOutput (interface)
java.io.InputStream           java.io.InputStreamReader
java.io.OutputStreamWriter    java.io.PrintStream
java.io.Writer                java.io.DataInput (interface)
java.io.DataOutputStream      java.io.OutputStream
java.io.Reader
```

A.5 Calendar and Time Classes

`java.util.Calendar` `java.util.Date` `java.util.TimeZone`

A.6 Utility Classes

`java.lang.Math` `java.util.Random`

A.7 Exception Classes

`java.io.EOFException`	`java.io.InterruptedIOException`
`java.io.IOException`	`java.io.UnsupportedEncodingException`
`java.io.UTFDataFormatException`	`java.lang.ArithmeticException`
`java.lang.ArrayIndexOutOfBoundsException`	`java.lang.ArrayStoreException`
`java.lang.ClassCastException`	`java.lang.ClassNotFoundException`
`java.lang.Exception`	`java.lang.IllegalAccessException`
`java.lang.IllegalArgumentException`	`java.lang.IllegalMonitorStateException`
`java.lang.IllegalThreadStateException`	`java.lang.IndexOutOfBoundsException`
`java.lang.InstantiationException`	`java.lang.InterruptedException`
`java.lang.NegativeArraySizeException`	`java.lang.NullPointerException`
`java.lang.NumberFormatException`	`java.lang.RuntimeException`
`java.lang.SecurityException`	`java.lang.StringIndexOutOfBoundsException`
`java.util.EmptyStackException`	`java.util.NoSuchElementException`

A.8 Error Classes

`java.lang.Error` `java.lang.OutOfMemoryError` `java.lang.VirtualMachineError`

A.9 Internationalization

`java.io.InputStreamReader` `java.io.OutputStreamWriter`

Appendix B

JML Grammar Summary

The following is a summary of the grammar for JML language that ajmlc compiler handles.

B.1 JML Reserved Words

\requires	\pre	\same
\ensures	\post	pure
\signals	\exsures	\old
\assignable	\modifiable	\modifies
\result	also	invariant
behavior	normal_behavior	exceptional_behavior
spec_public	spec_protected	instance
\not_specfied	\everything	\nothing
\only_assigned		

B.2 Method Specfication

```
method-specification ::= specification | extending-specification
extending-specification ::= also specification
specification ::= spec-case-seq
spec-case-seq ::= spec-case [ also spec-case ] . . .
spec-case ::= lightweight-spec-case | heavyweight-spec-case

lightweight-spec-case ::= generic-spec-case
generic-spec-case ::= spec-header generic-spec-body
generic-spec-body ::= simple-spec-body |  generic-spec-case-seq
generic-spec-case-seq ::= generic-spec-case
   [ also generic-spec-case ] . . .
spec-header ::= requires-clause [ requires-clause ] . . .
simple-spec-body ::= simple-spec-body-clause
   [ simple-spec-body-clause ] . . .
simple-spec-body-clause ::= assignable-clause | ensures-clause
```

```
   | signals-clause

heavyweight-spec-case ::= behavior-spec-case
   | exceptional-behavior-spec-case
   | normal-behavior-spec-case
behavior-spec-case ::= [ privacy ] behavior-keyword
   generic-spec-case
behavior-keyword ::= behavior | behaviour
normal-behavior-spec-case ::= [ privacy ]
   normal-behavior-keyword generic-spec-case
normal-behavior-keyword ::= normal_behavior
   | normal_behaviour
exceptional-behavior-spec-case ::= [ privacy ]
   exceptional-behavior-keyword generic-spec-case
exceptional-behavior-keyword ::= exceptional_behavior
   | exceptional_behaviour
privacy ::= public | protected | private

requires-clause ::= requires-keyword pred-or-not
   | requires-keyword \same
requires-keyword ::= \requires | \pre
ensures-clause ::= ensures-keyword pred-or-not
ensures-keyword ::= \ensures | \post
signals-clause ::= signals-keyword pred-or-not
signals-keyword ::= \signals | \exsures
assignable-clause ::= assignable-keyword store-ref-list
assignable-keyword ::= assignable | modifiable
   | modifies

store-ref-list :: = ...

pred-or-not ::= predicate | \not_specified
```

B.3 Type Specfication

```
jml-declaration ::= modifiers invariant
invariant ::= invariant-keyword
invariant-keyword ::= invariant
modifiers ::= [ modifier ] . . .
modifier ::= public | protected | private
   | abstract | static |
   | final | synchronized
   | transient | volatile
   | native | strictfp
   | const // reserved but not used in Java
   | jml-modifier
jml-modifier ::= spec_public | spec_protected
   | pure | instance
   | non_null | nullable
   | nullable_by_default
```

B.4 Predicates and Specification Expressions

```
predicate ::= spec-expression
spec-expression ::= expression

I ∈ identifier
E ∈ expression
E ::= E1 <==> E2
   | E1 <=!=> E2
   | E ==> E
   | E <== E2
   | E1 || E2
   | E1 && E2
   | E2 == E2
   | E1 != E2
   | E.I
   | !E
   | ...
   | jml-expressions

jml-expressions ::= result-expression
   | old-expression

result-expression ::= \result
old-expression ::= \old ( spec-expression)
   | \pre ( spec-expression )
```

Appendix C

Examples of Instrumented Code Generated by ajmlc

In this Appendix, we provide translations of two JML examples by the ajmlc.

C.1 Example of omitted and empty specifications

C.1.1 Source code of the example

Considering the code below (`T.java` and `S.java`), the class `T` has two methods `m` and `n` with omitted preconditions, whereas the method `o` has an empty specification. The class `S` that extends the class `T` has the method `n`, which is overridden from class `S` and also is an empty specification.

T.java

```
public class T {
  //@ assignable \nothing;
  public void m(int x){
  }

  //@ assignable \everything;
  public void n(int y){
  }

  public void o(int z){
  }

}
```

S.java

```
public class S extends T{
  //@ also
  //@ requires x > 10;
  public void m(int x){
  }

  public void n(int y){
  }

  //@ also
  //@ requires z > 20;
  public void o(int z){
  }

}
```

C.1.2 Instrumented code generated of the example

By compiling the hierarchy above with the ajmlc compiler (with the translation rules for omitted and empty specifications), we have the following code (`AspectJMLRac_T.aj` and `AspectJMLRac_S.aj`):

AspectJMLRac_T.aj

```
public privileged aspect AspectJMLRac_T {

  /** Generated by JML to insert a precondition
   * checking method for the method m. */
  public boolean T.checkPre$m$T(int x){
    return (((true)));
  }

  /** Generated by JML to insert a precondition
   * checking method for the method n. */
  public boolean T.checkPre$n$T(int y){
    return (((true)));
  }

  /** Generated by JML to insert a precondition
   * checking method for the method o. */
  public boolean T.checkPre$o$T(int z){
    return false;
  }

  /** Generated by JML to insert a precondition
   * checking method for the method o. */
  before (T current, int z) :
  execution(void T.o( int )) &&
  within(T) &&
  this(current) && args(z) {

    if (!current.checkPre$o$T(z)) {
      throw new JMLInternalPreconditionError("");
    }
  }

}
```

AspectJMLRac_S.aj

```
public privileged aspect AspectJMLRac_S {

  /** Generated by JML to insert a precondition
   * checking method for the method m. */
  public boolean S.checkPre$m$S(int x){
    return (((x > 10))) || super.checkPre$m$T(x);
  }

 /** Generated by JML to check the precondition of
  * method m. */
 before (S current, int x) :
 execution(void S.m( int )) &&
 within(S) &&
 this(current) && args(x) {

   if (!current.checkPre$m$S(x)) {
     throw new JMLInternalPreconditionError("");
   }
 }

 /** Generated by JML to insert a precondition
  * checking method for the method n. */
 public boolean S.checkPre$n$S(int y){
   return false || super.checkPre$n$T(y);
 }

 /** Generated by JML to insert a precondition
  * checking method for the method n. */
 before (S current, int y) :
 execution(void S.n( int )) &&
 within(S) &&
 this(current) && args(y) {

   if (!current.checkPre$n$S(y)) {
     throw new JMLInternalPreconditionError("");
   }
 }

 /** Generated by JML to insert a precondition
  * checking method for the method o. */
 public boolean S.checkPre$o$S(int z){
   return (((z > 20))) || super.checkPre$o$T(z);
 }
```

```
   /** Generated by JML to check the precondition of
    * method o. */
   before (S current, int z) :
   execution(void S.o( int )) &&
   within(S) &&
   this(current) && args(z) {

     if (!current.checkPre$o$S(z)) {
       throw new JMLInternalPreconditionError("");
     }
   }

}
```

C.2 Example of the same predicate

C.2.1 Source code of the example

The following code (`T2.java` and `S2.java`) is an example of the usage of the same predicate in JML. The class `T2` illustrates the case when a method has more than one specification case and one of them has the same predicate. This case is employed by the method `m`. On the other hand, the class `S2` illustrates the case when a method has only one specification case with the same predicate, and it is overridden. This case is employed by the overridden method `n`.

T2.java

```
public class T2 {
  //@ requires x > 0;
  //@ also
  //@ requires \same;
  public void m(int x){
  }

  //@ requires y > 0;
  //@ requires y < 100;
  public void n(int y){
  }

}
```

S2.java

```
public class S2 extends T2{
  //@ also
  //@ requires \same;
  public void n(int y){
  }
}
```

C.2.2 Instrumented code generated of the example

The AspectJ code generated by the ajmlc when compiling the code above is shown below (`AspectJMLRac_T2.aj` and `AspectJMLRac_S2.aj`).

AspectJMLRac_T2.aj

```
public privileged aspect AspectJMLRac_T2 {

  /** Generated by JML to insert a precondition
   * checking method for the method m. */
  public boolean T2.checkPre$m$T2(int x){
    return (((x > 0)) || ((x > 0)));
  }

  /** Generated by JML to check the precondition of
   * method m. */
  before (T2 current, int x) :
  execution(void T2.m( int )) &&
  within(T2) &&
  this(current) && args(x) {

    if (!current.checkPre$m$T2(x)) {
      throw new JMLInternalPreconditionError("");
    }
  }

  /** Generated by JML to insert a precondition
   * checking method for the method n. */
  public boolean T2.checkPre$n$T2(int y){
    return (((y > 0) && (y < 100)));
  }

  /** Generated by JML to check the precondition of
   * method n. */
  before (T2 current, int y) :
  execution(void T2.n( int )) &&
  within(T2) &&
  this(current) && args(y) {

    if (!current.checkPre$n$T2(y)) {
      throw new JMLInternalPreconditionError("");
    }
  }

}
```

AspectJMLRac_S2.aj

```
public privileged aspect AspectJMLRac_S2 {

  /** Generated by JML to insert a precondition
   * checking method for the method n. */
  public boolean S2.checkPre$n$S(int y){
    return super.checkPre$n$T2(y);
  }

}
```

Appendix D

Translation Rules

In this Appendix, we list all translation rules and their definition used in the compilation process that translates a JML annotated Java type into an AspectJ aspect.

D.1 Auxiliary functions

In the following we present all auxiliary functions used in the compilation process.

D.1.1 Elements used by the auxiliary functions

The following elements are used by the auxiliary functions:

- The *Annotated_Java_Type* is a tuple composed by the following elements needed for the translation rules:
  - — $TMod$ — is a sequence of type modifiers, $\langle TMod_1, ..., TMod_x\rangle$;
  - — TN — is the name of the *Annotated_Java_Type* to be compiled;
  - — $Meth$ — is a sequence of methods, $\langle Meth_1, ..., Meth_m\rangle$;
  - — SCN — is the name of the superclass, in which the *Annotated_Java_Type* is subtype;
  - — SIN — is a set of superinterface names, $\{SIN_1,...,SIN_n\}$, of the *Annotated_Java_Type*;
  - — $Inst\_Inv$ — is a set of instance invariants, $\{Inst\_Inv_1,...,Inst\_Inv_r\}$, into the *Annotated_Java_Type*;
  - — $Stat\_Invs$ — is a set of static invariants, $\{Stat\_Inv_1,...,Stat\_Inv_s\}$, into the *Annotated_Java_Type*;
- Each method in $Meth$ is a tuple composed by the following elements:
  - — $MMod$ — is a sequence of method modifiers, $\langle MMod_1, ..., MMod_y\rangle$;
  - — MN — is the method name;
  - — FP — is a sequence of formal parameters (pairs of types and identifiers), $\langle T_1, Id_1, ..., T_z, Id_z\rangle$;

— PTN — is a sequence of parameter type names, $\langle T_1, ..., T_z\rangle$;

— PIN — is a sequence of parameter identifier names, $\langle Id_1, ..., Id_z\rangle$;

— RT — is the method return type;

— Ex — is a set of exceptions, $\{E_1,...,E_s\}$;

— SC — is a sequence of local JML specification cases, $\langle SC_1, ..., SC_k\rangle$. Such local specification cases are represented as tuples, $\langle P_i, Q_i, R_i\rangle_{i=1,...,k}$, composed by preconditions P, normal postconditions Q, and exceptional postconditions R.

- Each sequence and each set could also be empty:

— $\langle\epsilon\rangle$ — denotes an empty sequence;

— $\{\}$ — denotes an empty set.

D.1.2 Precondition auxiliary function

```
MAP_PreconditionToAspectJCode: TN x TMod x SCN x SIN x MN x MMod x FP x
PTN x PIN x RT x SC → AspectJ_Code

MAP_PreconditionToAspectJCode[[TN, TMod, SCN, SIN, MN, MMod, FP, PTN, PIN, RT, SC]] ≜
  if SC ≠ ⟨ε⟩
  then if 'static' ∈ MMod
       then ⟨⟨staticBeforeAdviceForPreconditionChecking⟩⟩
            ⟨⟨staticPreconditionChekingMethod⟩⟩
       else if 'interface' ∈ TMod
            then ⟨⟨preconditionChekingMethod⟩⟩
            else ⟨⟨beforeAdviceForPreconditionChecking⟩⟩
                 ⟨⟨preconditionChekingMethod⟩⟩

⟨⟨beforeAdviceForPreconditionChecking⟩⟩≡
   let ⟨T_1,Id_1...T_z,Id_z⟩ = FP in
   let ⟨T_1,...T_z⟩ = PTN in
   let ⟨Id_1...Id_z,⟩ = PIN in
   before (TN current, T_1 Id_1,...,T_z Id_z):
     execution (RT TN.MN (T_1,...,T_z)) &&
     within (TN) &&
     this (current) &&
     args (Id_1,...,Id_n){
        if (!current.check$MN$TN (Id_1,...,Id_z)){
           throw new JMLInternalPreconditionError();
        }
     }

⟨⟨preconditionChekingMethod⟩⟩≡
    let ⟨T_1,Id_1...T_z,Id_z⟩ = FP in
    if (SCN ≠ 'java.lang.Object') ∨ (SIN ≠ {})
    then let ⟨SIN_1,...SIN_n⟩ = PTN in
         public boolean TN.check$MN$TN (T_1 Id_1,...,T_z Id_z){
           return $\langle\langle$preconditionsToCheck$\rangle\rangle$ ||
             checkPre$MN$SCN ($Id_1$,...,$Id_z$) ||
             checkPre$MN$SIN1($Id_1$,...,$Id_z$),...,||
             checkPre$MN$SINn($Id_1$,...,$Id_z$);
         }
    else
       public boolean TN.check$MN$TN ($T_1$ $Id_1$,...,$T_z$ $Id_z$){
         return $\langle\langle$preconditionsToCheck$\rangle\rangle$;
       }
```

```
$\langle\langle$preconditionsToCheck$\rangle\rangle\equiv$
   if SC = $\langle\epsilon\rangle$
   then
      false
   else let $\langle$P,Q,R$\rangle$ = SC in
        if P $\neq$ $\langle\epsilon\rangle$
        then let $\langle P_1$,...$P_k\rangle$ = P in
        $P_1$ || ... || $P_k$
        else
           true
```

```
$\langle\langle$staticBeforeAdviceForPreconditionChecking$\rangle\rangle\equiv$
   let $\langle T_1$,$Id_1$...$T_z$,$Id_z\rangle$ = FP in
   let $\langle T_1$,...$T_z\rangle$ = PTN in
   let $\langle Id_1$...$Id_z$,$\rangle$ = PIN in
   before ($T_1$ $Id_1$,...,$T_z$ $Id_z$):
     execution (RT TN.MN ($T_1$,...,$T_z$)) &&
     args ($Id_1$,...,$Id_n$){
        if (!TN.check$MN$TN ($Id_1$,...,$Id_z$)){
           throw new JMLInternalPreconditionError();
        }
     }
```

```
$\langle\langle$staticPreconditionChekingMethod$\rangle\rangle\equiv$
   let $\langle T_1$,$Id_1$...$T_z$,$Id_z\rangle$ = FP in
   public static boolean TN.check$MN$TN ($T_1$ $Id_1$,...,$T_z$ $Id_z$){
     return $P_1$ || ... || $P_k$;
   }
```

D.1.3 Postcondition auxiliary function

```
$MAP_{PostconditionToAspectJCode}$: TN, x MN x MMod x FP x PTN x PIN x
RT x Ex x SC $\rightarrow$ AspectJ_Code

$MAP_{PostconditionToAspectJCode}$[[TN, MN, MMod, FP, PTN, PIN, RT, Ex, SC]] $\underline{\triangle}$
  if SC $\neq$ $\langle\epsilon\rangle$
  then let $\langle$P,Q,R$\rangle$ = SC in
       if (Q $\neq$ $\langle\epsilon\rangle$) $\vee$ (R $\neq$ $\langle\epsilon\rangle$)
       then let $\langle Q_1$,...$Q_k\rangle$ = Q in
```

```
            let ⟨$R_1$,...$R_k$⟩ = $R$ in
            if ‘static’ $\in$ $MMod$
            then ⟨⟨$staticAroudAdviceForPostconditionChecking$⟩⟩
            else ⟨⟨$aroudAdviceForPostconditionChecking$⟩⟩

⟨⟨$aroundAdviceForPostconditionChecking$⟩⟩≡
   let ⟨$Ex_1$,...$Ex_s$⟩ = $Ex$ in
   let ⟨$T_1$,$Id_1$,...$T_z$,$Id_z$⟩ = $FP$ in
   let ⟨$T_1$,...$T_z$⟩ = $PTN$ in
   let ⟨$Id_1$...$Id_z$,⟩ = $PIN$ in
   $RT$ around ($TN$ current, $T_1$ $Id_1$,...,$T_z$ $Id_z$) throws $Ex_1$,...$Ex_s$ :
     execution ($RT$ $TN.MN$($T_1$,...,$T_z$)) &&
     this (current) &&
     args ($Id_1$,...,$Id_z$) {
        $RT$ rac$result; // represents the return of the method
        try{
          ⟨⟨$saveAllOldValues$⟩⟩
          try{
            // executing the original method
            rac$result = proceed(current, $Id_1$,...,$Id_z$);
            ⟨⟨$checkNormalPostcondition$⟩⟩
          } catch (Throwable rac$e){
              ⟨⟨$rethrowJMLException$⟩⟩
              ⟨⟨$checkExceptionalPostcondition$⟩⟩
          }
        } catch (Throwable rac$cause){
            throw new JMLEvaluationError(rac$cause);
        }
     }

⟨⟨$staticAroundAdviceForPostconditionChecking$⟩⟩≡
   let ⟨$Ex_1$,...$Ex_s$⟩ = $Ex$ in
   let ⟨$T_1$,$Id_1$,...$T_z$,$Id_z$⟩ = $FP$ in
   let ⟨$T_1$,...$T_z$⟩ = $PTN$ in
   let ⟨$Id_1$...$Id_z$,⟩ = $PIN$ in
   $RT$ around ($T_1$ $Id_1$,...,$T_z$ $Id_z$) throws $Ex_1$,...$Ex_s$ :
     execution (static $RT$ $TN.MN$($T_1$,...,$T_z$)) &&
     args ($Id_1$,...,$Id_z$) {
        $RT$ rac$result; // represents the return of the method
        try{
          ⟨⟨$saveAllOldValues$⟩⟩
          try{
            // executing the original method
            rac$result = proceed(current, $Id_1$,...,$Id_z$);
            ⟨⟨$checkNormalPostcondition$⟩⟩
          } catch (Throwable rac$e){
              ⟨⟨$rethrowJMLException$⟩⟩
              ⟨⟨$checkExceptionalPostcondition$⟩⟩
          }
        } catch (Throwable rac$cause){
            throw new JMLEvaluationError(rac$cause);
        }
     }
```

```
⟨⟨saveAllOldValues⟩⟩≡
   $old\$v_1$ := $v_1$;
      ...
   $old\$v_f$ := $v_f$;
```

```
⟨⟨checkNormalPostcondition⟩⟩≡
  if(!((!($P_1[\![v_i$:=$old\$v_i]\!]$ || $Q_1[\![v_i$:=$old\$v_i]\!]$) &&
 ... && (!($P_k[\![v_i$:=$old\$v_i]\!]$ || $Q_k[\![v_i$:=$old\$v_i]\!]$))){
   throw new JMLInternalNormalPostconditionError();
 }
```

```
⟨⟨rethrowJMLException⟩⟩≡
  if(rac$e instanceof JMLInternalNormalPostconditionError){
    throw (JMLInternalNormalPostconditionError) rac$e;
  }
```

```
⟨⟨checkExceptionalPostcondition⟩⟩≡
  boolean rac$v = true;
  boolean $rac\$pre_1$ = $P_1[\![v_i$:=$old\$v_i]\!]$;
  if(rac$v && $rac\$pre_1$){
    if(rac$e instanceof $X_{11}$){
      boolean $flag_1$ = true;
      $X_{11}$ $e_{11}$ = ($X_{11}$)rac$e;
      $flag_1$ = $R_1[\![v_i$:=$old\$v_i]\!]$;
      rac$v = rac$v && $flag_1$;
     }
     ...
  }
   ...
  boolean $rac\$pre_k$ = $P_k[\![v_i$:=$old\$v_i]\!]$;
  if(rac$v && $rac\$pre_k$){
    if(rac$e instanceof $X_{k1}$){
      boolean $flag_k$1 = true;
      $X_{k1}$ $e_{k1}$ = ($X_{k1}$)rac$e;
      $flag_{k1}$ = $R_k[\![v_i$:=$old\$v_i]\!]$;
      rac$v = rac$v && $flag_{k1}$;
    }
     ...
  }

  if(!rac$v){
    throw new JMLInternalExceptionalPostconditionError();
  }
  else{
        if(rac$e instanceof $X_{11}$){
          throw ($X_{11}$) rac$e;
        }
         ...
        if(rac$e instanceof $X_{kj}$){
          throw ($X_{kj}$) rac$e;
        }
      }
```

D.1.4 Invariant auxiliary functions

```
$MAP_{InstanceInvariantBMEToAspectJCode}$: TN x Inst_Inv $\rightarrow$ AspectJ_Code

$MAP_{InstanceInvariantBMEToAspectJCode}[\![TN, Inst\_Inv]\!] \triangleq$
  if Inst_Inv $\neq$ {}
  then let {InstInv$_1$ && ... && InstInv$_r$} = Inst_Inv
       before ( TN current):
         execution (!static * TN.*(..)) &&
         this (current) {
            $\langle\langle$checkInstanceInvariant$\rangle\rangle$
         }

$MAP_{InstanceInvariantAMEToAspectJCode}$: TN x Inst_Inv $\rightarrow$ AspectJ_Code

$MAP_{InstanceInvariantAMEToAspectJCode}[\![TN, Inst\_Inv]\!] \triangleq$
 if Inst_Inv $\neq$ {}
  then let {Inst_Inv$_1$ && ... && Inst_Inv$_r$} = Inst_Inv
       after ( TN current) returning (Object o):
         execution (!static * TN.*(..)) &&
         this (current) {
            $\langle\langle$checkInstanceInvariant$\rangle\rangle$
         }
       after ( TN current) throwing (Throwable rac$thrown):
         execution (!static * TN.*(..)) &&
         this (current) {
            $\langle\langle$rethrowJMLException$\rangle\rangle$
            else {
               $\langle\langle$checkInstanceInvariant$\rangle\rangle$
            }
         }

$\langle\langle$checkInstanceInvariant$\rangle\rangle\equiv$
 if (!(Inst_Inv$_1$ && ... && Inst_Inv$_r$)){
   throw new JMLInvariantError();
 }

$\langle\langle$rethrowJMLException$\rangle\rangle\equiv$
 if (rac$thrown instanceof JMLInternalPreconditionError) {
   throw (JMLInternalPreconditionError) rac$thrown;
 }
 ...
 else if (rac$thrown instanceof JMLInvariantError) {
   throw (JMLInvariantError) rac$thrown;
 }
```

```
MAP_StaticInvariantBMEToAspectJCode: TN x TMod x Stat_Inv → AspectJ_Code

MAP_StaticInvariantBMEToAspectJCode[[TN, TMod, Stat_Inv]] △
  if Stat_Inv ≠ {}
  then let {Stat_Inv_1 && ... && Stat_Inv_s} = Inst_Inv
       if 'interface' in TMod
       then
          before ():
            execution (* TN+.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
       else
          before ():
            execution (* TN.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
```

```
MAP_StaticInvariantAMEToAspectJCode: TN x TMod x Stat_Inv → AspectJ_Code

MAP_StaticInvariantAMEToAspectJCode[[TN, TMod, Stat_Inv]] △
  if Stat_Inv ≠ {}
  then let {Stat_Inv_1 && ... && Stat_Inv_s} = Inst_Inv
       if 'interface' in TMod
       then
          after () returning (Object o):
            execution (* TN+.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
          after () throwing (Throwable rac$thrown):
            execution (* TN+.*(..)) {
               ⟨⟨rethrowJMLException⟩⟩
               else {
                  ⟨⟨checkStaticInvariant⟩⟩
               }
            }
       else
          after () returning (Object o):
            execution (* TN.*(..)) {
               ⟨⟨checkStaticInvariant⟩⟩
            }
          after () throwing (Throwable rac$thrown):
            execution (* TN.*(..)) {
               ⟨⟨rethrowJMLException⟩⟩
```

```
              else {
                 ⟨⟨checkStaticInvariant⟩⟩
              }
           }
```

```
⟨⟨checkStaticInvariant⟩⟩≡
 if (!(Stat_Inv_1 && ... && Stat_Inv_s)){
   throw new JMLInvariantError();
 }
```

D.2 Complete translation rules

In the following we present all translation rules needed to translate a JML annotated Java type into an aspect.

```
MAP_JmlToAspectJAspect[[Annotated_Java_Type]] △
  let TN, TMod, Meth, SCN, SIN,
          Inst_Inv, Stat_Inv = Annotated_Java_Type in
  let StatInvBefore = MAP_StaticInvariantBMEToAspectJCode[[Stat_Inv]] in
  let InstInvBefore = MAP_InstanceInvariantBMEToAspectJCode[[Inst_Inv]] in
  if Meth ≠ ⟨ϵ⟩
  then let ⟨Meth_1,...Meth_m⟩ = Meth in
       let TMod_1, MN_1, FP_1, PTN_1, PIN_1, RT_1, Ex_1, SC_1 = Meth_1 in
                                ...
       let TMod_m, MN_m, FP_m, PTN_m, PIN_m, RT_m, Ex_m, SC_m = Meth_m in
       let Precondition_1 = MAP_PreconditionToAspectJCode[[TN,TMod,SCN,SIN,MN_1,
                                       MMod_1,FP_1,PTN_1,PIN_1,RT_1,SC_1]] in
                                ...
       let Precondition_m = MAP_PreconditionToAspectJCode[[TN,TMod,SCN,SIN,MN_m,
                                     MMod_m,FP_m,PTN_m,PIN_m,RT_m,SC_m]] in
       let Postcondition_1 = MAP_PostconditionToAspectJCode[[TN,MN_1,MMod_1,FP_1,
                                        PTN_1,PIN_1,RT_1,Ex_1,SC_1]] in
                                ...
       let Postcondition_m = MAP_PostconditionToAspectJCode[[TN,MN_m,MMod_m,FP_m,
                                        PTN_m,PIN_m,RT_m,Ex_m,SC_m]] in
  let InstInvAfter = MAP_InstanceInvariantAMEToAspectJCode[[Inst_Inv]] in
  let StatInvAfter = MAP_StaticInvariantAMEToAspectJCode[[Stat_Inv]] in
  if Meth ≠ ⟨ϵ⟩
  then let AspectJCode = StatInvBefore ∪ InstInvBefore ∪
       ⟨Precondition_1,Postcondition_1⟩ ... ∪ ⟨Precondition_m,Postcondition_m⟩
       ∪ InstInvAfter ∪ StatInvAfter in
  else let AspectJCode = StatInvBefore ∪ InstInvBefore ∪ InstInvAfter
       ∪ StatInvAfter in
  if AspectJCode ≠ ϵ
  then
     public privileged aspect AspectJMLRac$TN {
       AspectJCode
     }
```

Appendix E

Source code used in Proofs of Concept

In this Appendix, we make available all source code used in the Chapter 4. The source code listed below contains the annotated classes and their instrumentation code generated by the ajmlc compiler.

E.1 Study with Java ME platform

E.1.1 Source code of the example

The following Java source code is a MIDlet [56, 63] application annotated with JML. It is used as a input source file for all versions (*CalcAspSol*, *CalcJmlSol*, and *CalcPureSol*) of the study conducted in Section 4.1.

CalculatorMIDlet.java

```
public final class CalculatorMIDlet extends MIDlet implements CommandListener {

  /** The number of characters in numeric text field. */
  private static final int NUM_SIZE = 20;

  /** Soft button for exiting the game. */
  private final Command exitCmd = new Command("Exit", Command.EXIT, 2);

  /** Menu item for changing game levels. */
  private final Command calcCmd = new Command("Calc", Command.SCREEN, 1);

  /** A text field to keep the first argument. */
  private final TextField t1 = new TextField(null, "",
  NUM_SIZE, TextField.DECIMAL);

  /** A text field to keep the second argument. */
  private final TextField t2 = new TextField(null, "",
  NUM_SIZE, TextField.DECIMAL);
```

```
 /** A text field to keep the result of calculation. */
 private final TextField tr = new TextField("Result", "",
 NUM_SIZE, TextField.UNEDITABLE);

 /** A choice group with available operations. */
 private final ChoiceGroup cg =
  new ChoiceGroup("", ChoiceGroup.POPUP,
      new String[] { "add", "subtract", "multiply", "divide",
      "invariatViolation" }, null);

 /** An alert to be reused for different errors. */
 private final Alert alert = new Alert("Error", "", null, AlertType.ERROR);

 /** Indicates if the application is initialized. */
 private boolean isInitialized = false;

 /** Added for experiment. */

 //@ public instance invariant result >= 0;
 public double result = 0.0;

 public void method(){
result = -1;
 }

 /**
  * Creates the calculator view and action buttons.
  */
 protected void startApp() {
    if (isInitialized) {
       return;
    }
   Form f = new Form("FP Calculator");
   f.append(t1);
   f.append(cg);
   f.append(t2);
   f.append(tr);
   f.addCommand(exitCmd);
   f.addCommand(calcCmd);
   f.setCommandListener(this);
   Display.getDisplay(this).setCurrent(f);
   alert.addCommand(new Command("Back", Command.SCREEN, 1));
   isInitialized = true;
 }

 /**
  * Does nothing. Redefinition is required by MIDlet class.
  */
 protected void destroyApp(boolean unconditional) {
 }

 /**
  * Does nothing. Redefinition is required by MIDlet class.
  */
 protected void pauseApp() {
 }

 //@ ensures result   ; == a + b;
 public double add(double a, double b){
return a + b;
 }

 //@ requires b <= a;
 //@ ensures result   ; == a - b;
 public double sub(double a, double b){
return a - b;
 }

 //@ ensures result   ; == a * b;
 public double mult(double a, double b){
return a * b;
 }

 //@ requires b > 0;
 //@ ensures result   ; == a / b;
 public double div(double a, double b){
return a / b;
 }

 /**
  * Responds to commands issued on CalculatorForm.
  *
  * @param c command object source of action
  * @param d screen object containing the item the action was performed on.
  */
 public void commandAction(Command c, Displayable d) {
    if (c == exitCmd) {
        destroyApp(false);
        notifyDestroyed();
       return;
    }

    try {
      double n1 = getNumber(t1, "First");
      double n2 = getNumber(t2, "Second");

      switch (cg.getSelectedIndex()) {
      case 0:
            result = add(n1, n2);

            break;

        case 1:
            result = sub(n1, n2);

            break;

        case 2:
            result = mult(n1, n2);

            break;

        case 3:
         result = div(n1, n2);

            break;

        case 4:
         method();

            break;

        default:
        }
    } catch (NumberFormatException e) {
        return;
    } catch (ArithmeticException e) {
        alert.setString("Divide by zero.");
        Display.getDisplay(this).setCurrent(alert);

        return;
    }

   /*
    * The resulted string may exceed the text max size.
    * We need to correct last one then.
    */
    String res_str = Double.toString(result);

      if (res_str.length() > tr.getMaxSize()) {
          tr.setMaxSize(res_str.length());
      }

      tr.setString(res_str);
    }

   /**
```

```
     * Extracts the double number from text field.
     *
     * @param t the text field to be used.
     * @param type the string with argument number.
     * @throws NumberFormatException is case of wrong input.
     */
    private double getNumber(TextField t, String type)
      throws NumberFormatException {
      String s = t.getString();

      if (s.length() == 0) {
         alert.setString("No " + type + " Argument");
         Display.getDisplay(this).setCurrent(alert);
         throw new NumberFormatException();
      }

      double n;
      try {
         n = Double.parseDouble(s);
      } catch (NumberFormatException e) {
         alert.setString(type + " argument is out of range.");
         Display.getDisplay(this).setCurrent(alert);
         throw e;
      }

      return n;
    }

  }// end of class 'CalculatorMIDlet' definition
```

E.1.2 Instrumented code generated of the example

By compiling the Java ME calculator application with ajmlc compiler, we obtain the following instrumented code (represented by the *CalcAspSol* version of the study conducted in Section 4.1.):

AspectJMLRac_CalculatorMIDlet.aj

```
  public privileged aspect AspectJMLRac_T {

    /** Generated by JML to insert a precondition
     * checking method for the method m. */
    public boolean T.checkPre$m$T(int x){
      return (((true)));
    }

    /** Generated by JML to check non-static invariants of
     * class CalculatorMIDlet. */
    before (calculator.CalculatorMIDlet current) :
```

```
execution(!static * calculator.CalculatorMIDlet.*(..)) &&
this(current) {

  if (!(((current.result >= +0.0D)))) {
    throw new JMLInvariantError("");
  }
}

/** Generated by JML to insert a precondition
 * checking method for the method method. */
public boolean calculator.CalculatorMIDlet.checkPre$method$CalculatorMIDlet(){
  return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method startApp. */
public boolean calculator.CalculatorMIDlet.checkPre$startApp$CalculatorMIDlet(){
  return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method destroyApp. */
public boolean calculator.CalculatorMIDlet.
  checkPre$destroyApp$CalculatorMIDlet(boolean unconditional){
    return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method pauseApp. */
public boolean calculator.CalculatorMIDlet.checkPre$pauseApp$CalculatorMIDlet(){
  return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method add. */
public boolean calculator.CalculatorMIDlet.
  checkPre$add$CalculatorMIDlet(double a, double b){
    return (((true)));
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method add. */
double around (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.add( double, double ))
&&    this(current)   && args(a, b) {
  double rac$result = 0;

  // Pre-state environment
  // saving all old values
  rac$result = proceed(current,a, b);//executing the method

  // Post-state environment

  if (!(!((true))  || (((rac$result == (a + b)))))){
    throw new JMLInternalNormalPostconditionError("");
  }

  return rac$result;
}

/** Generated by JML to insert a precondition
 * checking method for the method sub. */
public boolean calculator.CalculatorMIDlet.
  checkPre$sub$CalculatorMIDlet(double a, double b){
    return (((b <= a)));
}

/** Generated by JML to check the precondition of
 * method sub. */
before (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.sub( double, double )) &&
within(calculator.CalculatorMIDlet) &&
this(current) && args(a, b) {

  if (!current.checkPre$sub$CalculatorMIDlet(a, b)) {
    throw new JMLInternalPreconditionError("");
  }
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method sub. */
double around (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.sub( double, double ))
&&    this(current)   && args(a, b) {
  double rac$result = 0;

  // Pre-state environment
  // saving all old values
  rac$result = proceed(current,a, b);//executing the method

  // Post-state environment

  if (!(!(((b <= a))) || (((rac$result == (a - b)))))){
    throw new JMLInternalNormalPostconditionError("");
  }


  return rac$result;
```

```
}

/** Generated by JML to insert a precondition
 * checking method for the method mult. */
public boolean calculator.CalculatorMIDlet.
  checkPre$mult$CalculatorMIDlet(double a, double b){
    return (((true))) ;
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method mult. */
double around (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.mult( double, double ))
&&    this(current)   && args(a, b) {
  double rac$result = 0;

  // Pre-state environment
  // saving all old values
  rac$result = proceed(current,a, b);//executing the method

  // Post-state environment

  if (!(!((true))  || (((rac$result == (a * b))))))){
    throw new JMLInternalNormalPostconditionError("");
  }

  return rac$result;
}

/** Generated by JML to insert a precondition
 * checking method for the method div. */
public boolean calculator.CalculatorMIDlet.
  checkPre$div$CalculatorMIDlet(double a, double b){
    return (((b > +0.0D)));
}

/** Generated by JML to check the precondition of
 * method div. */
before (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.div( double, double )) &&
within(calculator.CalculatorMIDlet) &&
this(current) && args(a, b) {

  if (!current.checkPre$div$CalculatorMIDlet(a, b)) {
    throw new JMLInternalPreconditionError("");
  }
}

/** Generated by JML to check the normal (and/or)
```

```
 * exceptional postcondition of method div. */
double around (calculator.CalculatorMIDlet current, double a, double b) :
execution(double calculator.CalculatorMIDlet.div( double, double ))
&&    this(current)   && args(a, b) {
  double rac$result = 0;

  // Pre-state environment
  // saving all old values
  rac$result = proceed(current,a, b);//executing the method

  // Post-state environment

  if (!(!(((b > +0.0D))) || (((rac$result == (a / b)))))){
   throw new JMLInternalNormalPostconditionError("");
  }

  return rac$result;
}

/** Generated by JML to insert a precondition
 * checking method for the method commandAction. */
public boolean calculator.CalculatorMIDlet.
  checkPre$commandAction$CalculatorMIDlet(javax.microedition.lcdui.Command c,
    javax.microedition.lcdui.Displayable d){
      return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method getNumber. */
public boolean calculator.CalculatorMIDlet.
  checkPre$getNumber$CalculatorMIDlet(javax.microedition.lcdui.TextField t,
    java.lang.String type){
      return true;
}

/** Generated by JML to check non-static invariants of
 * class CalculatorMIDlet. */
after (calculator.CalculatorMIDlet current)returning (Object o) :
execution(!static * calculator.CalculatorMIDlet.*(..)) &&
this(current) {

  if (!(((current.result >= +0.0D)))) {
    throw new JMLInvariantError("");
  }
}

/** Generated by JML to check non-static invariants of
 * class CalculatorMIDlet. */
after (calculator.CalculatorMIDlet current)throwing (Throwable rac$thrown) :
```

```
execution(!static * calculator.CalculatorMIDlet.*(..)) &&
this(current) {

  if (rac$thrown instanceof JMLInternalPreconditionError) {
    throw (JMLInternalPreconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalNormalPostconditionError) {
    throw (JMLInternalNormalPostconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalExceptionalPostconditionError) {
    throw (JMLInternalExceptionalPostconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInvariantError) {
    throw (JMLInvariantError) rac$thrown;
  }
  else {
    if (!(((current.result >= +0.0D))))   {
      throw new JMLInvariantError("");
    }
  }
}


/** Generated by JML to check static and non-static invariants of
 * class CalculatorMIDlet. */
after (calculator.CalculatorMIDlet current)returning (Object o) :
execution(calculator.CalculatorMIDlet.new(..)) &&
this(current) &&
if (current.getClass() == calculator.CalculatorMIDlet.class) {

  if (!(((current.result >= +0.0D)))) {
    throw new JMLInvariantError("");
  }
}

/** Generated by JML to check static and non-static invariants of
 * class CalculatorMIDlet. */
after (calculator.CalculatorMIDlet current)throwing (Throwable rac$thrown) :
execution(calculator.CalculatorMIDlet.new(..)) &&
this(current) &&
  if (current.getClass() == calculator.CalculatorMIDlet.class) {

  if (rac$thrown instanceof JMLInternalPreconditionError) {
    throw (JMLInternalPreconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalNormalPostconditionError) {
    throw (JMLInternalNormalPostconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalExceptionalPostconditionError) {
          throw (JMLInternalExceptionalPostconditionError) rac$thrown;
        }
        else if (rac$thrown instanceof JMLInvariantError) {
          throw (JMLInvariantError) rac$thrown;
        }
        else {
          if (!(((current.result >= +0.0D)))) {
            throw new JMLInvariantError("");
          }
        }
      }

    }
```

E.2 Study with Java SE platform

E.2.1 Source code of the examples

The following five Java source codes are Java SE applications annotated with JML. They were used in an experimentation conducted in Section 4.2.

Animal.java

```
public class Animal {

  public String gender;
  protected boolean gen; //@ in gender;
  protected /*@ spec_public @*/ int age = 0;

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender;
  //@ ensures gender.equals(g);
  public Animal(final String g) {
    gen = g.equals("female");
  }

  public /*@ pure @*/ boolean isFemale() {
    return gen;
  }

  /*@ requires 0 <= a && a <= 150;
    @ assignable age;
    @ ensures age == a;
    @ also
    @ requires a < 0;
    @ assignable age;
    @ ensures age == \old(age);
    @*/
  public void setAge(final int a) {
    if (0 <= a) { age = a; }
  }

  //@ requires k > 0;
  public void j(int k){
  }

}
```

Person.java

```
public class Person extends Animal {

  public String gender;
  protected boolean gen; //@ in gender;
  protected /*@ spec_public @*/ int age = 0;

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender;
  //@ ensures gender.equals(g);
  public Animal(final String g) {
    gen = g.equals("female");
  }

  public /*@ pure @*/ boolean isFemale() {
    return gen;
  }

  /*@ requires 0 <= a && a <= 150;
    @ assignable age;
    @ ensures age == a;
    @ also
    @ requires a < 0;
    @ assignable age;
    @ ensures age == \old(age);
    @*/
  public void setAge(final int a) {
    if (0 <= a) { age = a; }
  }

  //@ requires k > 0;
  public void j(int k){
  }

}
```

Patient.java

```
public class Patient extends Person {
  //@ public invariant 0 <= age && age <= 150;
  protected /*@ spec_public@*/ List history;

  /*@ requires !obs.equals("");
    @ assignable history;
    @ ensures history.size() == \old(history.size()+1)
    @ && history.get(\old(history.size()+1)).equals(obs);
    @*/
  public void recordVisit(String obs) {
    history.add(new String(obs));
  }

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender, history;
  //@ ensures gender.equals(g);
  public Patient(String g) {
    super(g); history = new ArrayList();
  }

}
```

IntMathOps.java

```
public class IntMathOps {

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

}
```

StackAsArray.java

```
public class StackAsArray {

  public Object [] array;

  /*@ public normal_behavior
    @ requires size >= 0;
    @ assignable array;
    @ ensures length() == size;
    @*/
  public StackAsArray(int size) {
  }

  /*@ public normal_behavior
    @ requires e != null;
    @ assignable array;
    @ ensures length() == \old(length() + 1);
    @*/
  public void push(Object e) {
  }

  /*@ public exceptional_behavior
    @ signals (Exception e) isEmpty()
    @ && (e instanceof IllegalStateException);
    @*/
  public void pop() {
  }

  /*@ public behavior
    @ assignable \nothing;
    @ ensures \result != null;
    @ ensures !isEmpty();
    @ signals (Exception e) isEmpty()
    @ && (e instanceof IllegalStateException);
    @*/
  public Object top() {
    return null;
  }

  //@ ensures length() == 0;
  public /*@ pure @*/ boolean isEmpty() {
    return true;
  }

  public /*@ pure @*/ int length() {
    return 0;
  }

}
```

E.2.2 Instrumented code generated of the examples

By compiling the five Java SE applications with ajmlc compiler, we obtain the following instrumented codes:

AspectJMLRac_Animal.aj

```
public privileged aspect AspectJMLRac_Animal {
```

```
/** Generated by JML to check the precondition of
 * method Animal. */
before (Animal current, final java.lang.String g) :
execution(Animal.new( java.lang.String )) &&
this(current) &&
  if (current.getClass() == Animal.class) && args(g) {

  if (!(((g.equals(((java.lang.Object) ("female"))) ||
     g.equals(((java.lang.Object) ("male"))))))) {
       throw new JMLInternalPreconditionError("");
  }
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method Animal. */
void around (Animal current, final java.lang.String g) :
execution(Animal.new( java.lang.String )) &&
if (current.getClass() == Animal.class)
&&  this(current)   && args(g) {

 // Pre-state environment
 // saving all old values
 proceed(current,g);//executing the method
 // Post-state environment

 if (!(!(((g.equals(((java.lang.Object) ("female"))) ||
   g.equals(((java.lang.Object) ("male")))))) ||
   ((current.gender.equals(((java.lang.Object) (g))))))){
     throw new JMLInternalNormalPostconditionError("");
 }
}

/** Generated by JML to insert a precondition
 * checking method for the method isFemale. */
public boolean Animal.checkPre$isFemale$Animal(){
  return true;
}

/** Generated by JML to insert a precondition
 * checking method for the method setAge. */
public boolean Animal.checkPre$setAge$Animal(final int a){
  return ((((0 <= a) && (a <= 150))) || ((a < 0)));
}

/** Generated by JML to check the precondition of
 * method setAge. */
before (Animal current, final int a) :
execution(void Animal.setAge( int )) &&
within(Animal) &&
    this(current) && args(a) {

      if (!current.checkPre$setAge$Animal(a)) {
        throw new JMLInternalPreconditionError("");
      }
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method setAge. */
    void around (Animal current, final int a) :
    execution(void Animal.setAge( int ))
    &&  this(current)   && args(a) {

      // Pre-state environment
      // saving all old values
      int old_age = current.age;
      proceed(current,a);//executing the method
      // Post-state environment

      if (!(!((((0 <= a) && (a <= 150))))
        || (((current.age == a)))) && (!(((a < 0)))
        || (((current.age == old_age))))){
          throw new JMLInternalNormalPostconditionError("");
      }
    }

    /** Generated by JML to insert a precondition
     * checking method for the method j. */
    public boolean Animal.checkPre$j$Animal(int k){
      return (((k > 0)));
    }

    /** Generated by JML to check the precondition of
     * method j. */
    before (Animal current, int k) :
    execution(void Animal.j( int )) &&
    within(Animal) &&
    this(current) && args(k) {

      if (!current.checkPre$j$Animal(k)) {
        throw new JMLInternalPreconditionError("");
      }
    }

  }
```

AspectJMLRac_Person.aj

```
  public privileged aspect AspectJMLRac_Person {
```

```
/** Generated by JML to insert a precondition
 * checking method for the method setAge. */
public boolean Person.checkPre$setAge$Person(final int a){
  return super.checkPre$setAge$Animal(a);
}

/** Generated by JML to check the precondition of
 * method setAge. */
before (Person current, final int a) :
execution(void Person.setAge( int )) &&
within(Person) &&
this(current) && args(a) {

  if (!current.checkPre$setAge$Person(a)) {
    throw new JMLInternalPreconditionError("");
  }
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method setAge. */
void around (Person current, final int a) :
execution(void Person.setAge( int ))
&&  this(current)  && args(a) {

 // Pre-state environment
 // saving all old values
 proceed(current,a); //executing the method
 // Post-state environment

 if (!(!current.checkPre$setAge$Person(a)
     || (((!((65 <= current.age))
     || current.ageDiscount))))){
     throw new JMLInternalNormalPostconditionError("");
 }
}

/** Generated by JML to check the precondition of
 * method Person. */
before (Person current, final java.lang.String g) :
execution(Person.new( java.lang.String )) &&
this(current) &&
if (current.getClass() == Person.class) && args(g) {

  if (!(((g.equals(((java.lang.Object) ("female")))
     || g.equals(((java.lang.Object) ("male"))))))) {
      throw new JMLInternalPreconditionError("");
  }
}
```

```
/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method Person. */
void around (Person current, final java.lang.String g) :
execution(Person.new( java.lang.String )) &&
if (current.getClass() == Person.class)
 &&  this(current) && args(g) {

 // Pre-state environment
 // saving all old values
 proceed(current,g);//executing the method
 // Post-state environment

 if (!(!(((g.equals(((java.lang.Object) ("female")))
    || g.equals(((java.lang.Object) ("male"))))))
    || ((current.gender.equals(((java.lang.Object) (g))))))){
      throw new JMLInternalNormalPostconditionError("");
 }
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method m. */
   int around (int x) :
   execution(static int Person.m( int )) &&
   within(Person)
   && args(x) {
     int rac$result = 0;

     // Pre-state environment
     // saving all old values
     rac$result = proceed(x); //executing the method
     // Post-state environment

     if (!(!((true))  || (((x > 0))))){
       throw new JMLInternalNormalPostconditionError("");
     }

    return rac$result;
   }

/** Generated by JML to insert a precondition
 * checking method for the method j. */
public boolean Person.checkPre$j$Person(int k){
  return super.checkPre$j$Animal(k);
}

/** Generated by JML to insert a precondition
 * checking method for the method j. */
before (Person current, int k) :
execution(void Person.j( int )) &&
    within(Person) &&
    this(current) && args(k) {

      if (!current.checkPre$j$Person(k)) {
        throw new JMLInternalPreconditionError("");
      }
    }

  }
```

AspectJMLRac_Patient.aj

```
  public privileged aspect AspectJMLRac_Patient {

    /** Generated by JML to check non-static invariants of
     * class Patient. */
    before (Patient current) :
    execution(!static * Patient.*(..)) &&
    this(current) {

      if (!((((0 <= current.age) && (current.age <= 150))))) {
        throw new JMLInvariantError("");
      }
    }

    /** Generated by JML to insert a precondition
     * checking method for the method recordVisit. */
    public boolean Patient.checkPre$recordVisit$Patient(java.lang.String obs){
      return ((!(obs.equals(((java.lang.Object) (""))))));
    }

    /** Generated by JML to check the precondition of
     * method recordVisit. */
    before (Patient current, java.lang.String obs) :
    execution(void Patient.recordVisit( java.lang.String )) &&
    within(Patient) &&
    this(current) && args(obs) {

      if (!current.checkPre$recordVisit$Patient(obs)) {
        throw new JMLInternalPreconditionError("");
      }
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method recordVisit. */
    void around (Patient current, java.lang.String obs) :
    execution(void Patient.recordVisit( java.lang.String ))
    &&  this(current) && args(obs) {

     // Pre-state environment
```

```
 // saving all old values
 java.util.List old_history = current.history;
 proceed(current,obs); //executing the method
 // Post-state environment

 if (!(!((!(obs.equals(((java.lang.Object) (""))))))
    || ((((current.history.size() == (old_history.size() + 1))
    && current.history.get((old_history.size() + 1)).
      equals(((java.lang.Object) (obs))))))))){
        throw new JMLInternalNormalPostconditionError("");
 }
}

/** Generated by JML to check the precondition of
 * method Patient. */
before (Patient current, java.lang.String g) :
execution(Patient.new( java.lang.String )) &&
this(current) &&
if (current.getClass() == Patient.class) && args(g) {

  if (!(((g.equals(((java.lang.Object) ("female")))
    || g.equals(((java.lang.Object) ("male"))))))) {
      throw new JMLInternalPreconditionError("");
  }
}

/** Generated by JML to check the normal (and/or)
 * exceptional postcondition of method Patient. */
void around (Patient current, java.lang.String g) :
execution(Patient.new( java.lang.String )) &&
if (current.getClass() == Patient.class)
&&  this(current) && args(g) {

  // Pre-state environment
  // saving all old values
  proceed(current,g); //executing the method
  // Post-state environment

  if (!(!(((g.equals(((java.lang.Object) ("female")))
    || g.equals(((java.lang.Object) ("male"))))))
    || ((current.gender.equals(((java.lang.Object) (g))))))){
      throw new JMLInternalNormalPostconditionError("");
  }
}

/** Generated by JML to check non-static invariants of
 * class Patient. */
after (Patient current)returning (Object o) :
execution(!static * Patient.*(..)) &&
this(current) {

  if (!((((0 <= current.age) && (current.age <= 150))))) {
    throw new JMLInvariantError("");
  }
}

/** Generated by JML to check non-static invariants of
 * class Patient. */
after (Patient current)throwing (Throwable rac$thrown) :
execution(!static * Patient.*(..)) &&
this(current) {

  if (rac$thrown instanceof JMLInternalPreconditionError) {
    throw (JMLInternalPreconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalNormalPostconditionError) {
    throw (JMLInternalNormalPostconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInternalExceptionalPostconditionError) {
    throw (JMLInternalExceptionalPostconditionError) rac$thrown;
  }
  else if (rac$thrown instanceof JMLInvariantError) {
    throw (JMLInvariantError) rac$thrown;
  }
  else {
    if (!((((0 <= current.age) && (current.age <= 150))))) {
      throw new JMLInvariantError("");
    }
  }
}

/** Generated by JML to check static and non-static invariants of
 * class Patient. */
after (Patient current)returning (Object o) :
execution(Patient.new(..)) &&
this(current) &&
if (current.getClass() == Patient.class) {

  if (!((((0 <= current.age) && (current.age <= 150))))) {
    throw new JMLInvariantError("");
  }
}

/** Generated by JML to check static and non-static invariants of
 * class Patient. */
after (Patient current)throwing (Throwable rac$thrown) :
execution(Patient.new(..)) &&
this(current) &&
      if (current.getClass() == Patient.class) {

        if (rac$thrown instanceof JMLInternalPreconditionError) {
          throw (JMLInternalPreconditionError) rac$thrown;
        }
        else if (rac$thrown instanceof JMLInternalNormalPostconditionError) {
          throw (JMLInternalNormalPostconditionError) rac$thrown;
        }
        else if (rac$thrown instanceof JMLInternalExceptionalPostconditionError) {
          throw (JMLInternalExceptionalPostconditionError) rac$thrown;
        }
        else if (rac$thrown instanceof JMLInvariantError) {
          throw (JMLInvariantError) rac$thrown;
        }
        else {
          if (!((((0 <= current.age) && (current.age <= 150))))) {
            throw new JMLInvariantError("");
          }
        }
      }

    }
```

AspectJMLRac_IntMathOps.aj

```
    public privileged aspect AspectJMLRac_IntMathOps {

      /** Generated by JML to insert a precondition
       * checking method for the method isqrt. */
      public static boolean IntMathOps.checkPre$isqrt$IntMathOps(int y){
        return ((((y >= 0))));
      }

      /** Generated by JML to check the precondition of
       * method isqrt. */
      before (int y) :
      execution(static int IntMathOps.isqrt( int )) &&
      within(IntMathOps) && args(y) {

        if (!IntMathOps.checkPre$isqrt$IntMathOps(y)) {
          throw new JMLInternalPreconditionError("");
        }
      }

      /** Generated by JML to check the normal (and/or)
       * exceptional postcondition of method isqrt. */
      int around (int y) :
      execution(static int IntMathOps.isqrt( int )) &&
      within(IntMathOps)
      && args(y) {
```

```
      int rac$result = 0;

      // Pre-state environment
      // saving all old values
      rac$result = proceed(y); //executing the method
      // Post-state environment

      if (!(!(((y >= 0))) || (((((0 <= rac$result)
         && ((rac$result * rac$result) <= y))
         && ((!(((0 <= (((rac$result + 1)) * ((rac$result + 1))))))
         || (y < (((rac$result + 1)) * ((rac$result + 1))))))))))){
           throw new JMLInternalNormalPostconditionError("");
      }

      return rac$result;
    }

  }
```

AspectJMLRac_StackAsArray.aj

```
  public privileged aspect AspectJMLRac_StackAsArray {

    /** Generated by JML to check the precondition of
     * method StackAsArray. */
    before (StackAsArray current, int size) :
    execution(StackAsArray.new( int )) &&
    this(current) &&
    if (current.getClass() == StackAsArray.class) && args(size) {

      if (!((((size >= 0))))) {
        throw new JMLInternalPreconditionError("");
      }
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method StackAsArray. */
    void around (StackAsArray current, int size) :
    execution(StackAsArray.new( int )) &&
    if (current.getClass() == StackAsArray.class)
    &&  this(current) && args(size) {

      // Pre-state environment
      // saving all old values
      proceed(current,size); //executing the method
      // Post-state environment

      if (!(!(((size >= 0))) || (((current.length() == size))))){
        throw new JMLInternalNormalPostconditionError("");
      {
    }

    /** Generated by JML to insert a precondition
     * checking method for the method push. */
    public boolean StackAsArray.checkPre$push$StackAsArray(java.lang.Object e){
      return ((((e != null))));
    }

    /** Generated by JML to check the precondition of
     * method push. */
    before (StackAsArray current, java.lang.Object e) :
    execution(void StackAsArray.push( java.lang.Object )) &&
    within(StackAsArray) &&
    this(current) && args(e) {

      if (!current.checkPre$push$StackAsArray(e)) {
        throw new JMLInternalPreconditionError("");
      }
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method push. */
    void around (StackAsArray current, java.lang.Object e) :
    execution(void StackAsArray.push( java.lang.Object ))
    &&  this(current) && args(e) {

      // Pre-state environment
      // saving all old values
      int old_length = current.length();
      proceed(current,e); //executing the method
      // Post-state environment

      if (!(!(((e != null))) || (((current.length() == (old_length + 1)))))){
        throw new JMLInternalNormalPostconditionError("");
      }
    }

    /** Generated by JML to insert a precondition
     * checking method for the method pop. */
    public boolean StackAsArray.checkPre$pop$StackAsArray(){
      return (((true))) ;
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method pop. */
    void around (StackAsArray current) :
    execution(void StackAsArray.pop(  )) &&
    this(current) {
      // Pre-state environment
      // saving all old values
     try {

      proceed(current); //executing the method

      // Post-state environment

      if (!(!((true))  || ((false)))){
        throw new JMLInternalNormalPostconditionError("");
      }

     } catch (Throwable rac$e) {
       if(rac$e instanceof JMLInternalNormalPostconditionError){
           throw (JMLInternalNormalPostconditionError) rac$e;
       }

       boolean rac$v = true;
       boolean rac$pre0 = (true);

       if(rac$v && rac$pre0){
         if(rac$e instanceof java.lang.Exception){
           boolean flag1 = true;
           java.lang.Exception e = (java.lang.Exception)rac$e;
           flag1 = ((current.isEmpty() &&
             (e instanceof java.lang.IllegalStateException)));
           rac$v = rac$v && flag1;
         }
       }

       if(!rac$v){
          throw new JMLInternalExceptionalPostconditionError("");
       }
     }
    }

    /** Generated by JML to insert a precondition
     * checking method for the method top. */
    public boolean StackAsArray.checkPre$top$StackAsArray(){
      return (((true)));
    }

    /** Generated by JML to check the normal (and/or)
     * exceptional postcondition of method top. */
    java.lang.Object around (StackAsArray current) :
    execution(java.lang.Object StackAsArray.top(  ))
    &&  this(current) {
      java.lang.Object rac$result = null;
      // Pre-state environment
      // saving all old values
     try {

       rac$result = proceed(current); //executing the method

      // Post-state environment

       if (!(!((true))  || (((rac$result != null)) && (!(current.isEmpty())))))){
         throw new JMLInternalNormalPostconditionError("");
       }

     } catch (Throwable rac$e) {
       if(rac$e instanceof JMLInternalNormalPostconditionError){
          throw (JMLInternalNormalPostconditionError) rac$e;
       }

       boolean rac$v = true;
       boolean rac$pre0 = (true);

       if(rac$v && rac$pre0){
         if(rac$e instanceof java.lang.Exception){
           boolean flag1 = true;
           java.lang.Exception e = (java.lang.Exception)rac$e;
           flag1 = ((current.isEmpty() &&
             (e instanceof java.lang.IllegalStateException)));
           rac$v = rac$v && flag1;
         }
       }

       if(!rac$v){
          throw new JMLInternalExceptionalPostconditionError("");
       }
     }

       return rac$result;
    }

    /** Generated by JML to insert a precondition
     * checking method for the method isEmpty. */
    public boolean StackAsArray.checkPre$isEmpty$StackAsArray(){
      return (((true))) ;
    }

    /** Generated by JML to insert a precondition
     * checking method for the method isEmpty. */
    before (StackAsArray current) :
    execution(boolean StackAsArray.isEmpty(  )) &&
    within(StackAsArray) &&
      this(current) {

        if (!current.checkPre$isEmpty$StackAsArray()) {
          throw new JMLInternalPreconditionError("");
        }
      }

      /** Generated by JML to check the normal (and/or)
       * exceptional postcondition of method isEmpty. */
      boolean around (StackAsArray current) :
      execution(boolean StackAsArray.isEmpty())
      &&  this(current) {
        boolean rac$result = false;

        // Pre-state environment
        // saving all old values
        rac$result = proceed(current); //executing the method

        // Post-state environment

        if (!(!((true))  || (((current.length() == 0))))){
          throw new JMLInternalNormalPostconditionError("");
        }

        return rac$result;
      }

      /** Generated by JML to insert a precondition
       * checking method for the method length. */
      public boolean StackAsArray.checkPre$length$StackAsArray(){
        return true;
      }

    }
```

Bibliography

[1] AJDT AspectJ Development Tools., 2008. http://www.eclipse.org/ajdt/.

[2] Eclipse Project., 2008. http://www.eclipse.org/.

[3] JavaCC (Java compiler compiler), 2008. http://javacc.dev.java.net/.

[4] The K Virtual Machine White Paper On-line, 2008. http://java.sun.com/products/cldc/wp/.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[7] Mike Barnett et al. The Spec# Programming System: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362, pages 49–69, 2005.

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[9] Holger Breitling, Carola Lilienthal, Martin Lippert, and Heinz Zllighoven. The JWAM Framework: Inspired by research, reality-tested by commercial utilization. In *OOPSLA 2000 Workshop: Methods and Tools for Object-Oriented Framework Development and Specialization*, 2000.

[10] Lionel C. Briand, W. J. Dzidek, and Yvan Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[12] Patrice Chalin. Are the Logical Foundations of Verifying Compiler Prototypes Matching user Expectations? *Form. Asp. Comput.*, 19(2):139–158, 2007.

[13] Patrice Chalin and Frèdèric Rioux. Non-null References by Default in the Java Modeling Language. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*, volume 31(2) of *ACM Software Engineering Notes*. ACM, 2005.

[14] Yoonsik Cheon. *A runtime assertion checker for the Java Modeling Language.* Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA, April 2003. The author's Ph.D. dissertation.

[15] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255, Malaga, Spain, June 2002. Springer Verlag.

[16] Yoonsik Cheon and Gary T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 149–158, New York, NY, USA, 2005. ACM.

[17] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[18] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[19] Constantinos Constantinides and Therapon Skotiniotis. Reasoning about a Classification of Cross-cutting Concerns in Object-Oriented Systems. In *Second Workshop on Aspect-Oriented Software Development (Workshop Aspektorientierte Softwareentwicklung der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung)*, Bonn, Germany, February 21-22, 2002.

[20] Eduardo S. Cordeiro, Roberto S. Bigonha, Mariza A. S. Bigonha, and Fabio Tirelo. Optimized Compilation of Around Advice for Aspect Oriented Programs. *Journal of Universal Computer Science*, 13(6):753–766, 2007.

[21] Frederico de Oliveira Jr., Ricardo Lima, Márcio Cornélio, Sérgio Soares, Paulo Maciel, Raimundo Barreto, Meuse Oliveira Jr., and Eduardo Tavares. CML: C Modeling Language. *Journal of Universal Computer Science*, 13(6):682–700, 2007.

[22] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[23] Filippo Diotalevi. Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ. July 2004. Avaliable at http://www.ibm.com/developerworks/library/j-ceaop.

[24] Yishai A. Feldman, Ohad Barzilay, and Shmuel Tyszberowicz. Jose: Aspects for Design by Contract80-89. *sefm*, 0:80–89, 2006.

[25] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[26] Lisa Friendly. The Design of Distributed Hyperlinked Programming Documentation. In *IWHD*, pages 151–173, 1995.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[29] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[30] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

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

[32] Charles Antony R. Hoare. Proof of Correctness of Data Representations. *Acta Inf.*, 1:271–281, 1972.

[33] Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Assertion with Aspect. In *International Workshop on Software Engineering Properties for Aspect Technologies (SPLAT2004)*, March 2004.

[34] Cliff B. Jones. *Systematic software development using VDM (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[35] Gregor Kiczales. TheServerSide.COM: Interview with Gregor Kiczales, topic: Aspect-oriented programming (AOP)., July 2003. http://www.theserverside.com/tt/talks/videos/GregorKiczalesText/interview.tss.

[36] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting Started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[37] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[38] Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[39] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[40] Gary T. Leavens and Yoonsik Cheon. Preliminary design of larch/c++. In *Proceedings of the first First International Workshop on Larch*, pages 159–184, London, UK, 1993. Springer-Verlag.

[41] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[42] Gary T. Leavens and Peter Müller. Information Hiding and Visibility in Interface Specifications. In *ICSE*, pages 385–395. IEEE Computer Society, 2007.

[43] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JMLReference Manual. Available from http://www.jmlspecs.org, October 2007.

[44] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java Programs via Guarded Commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.

[45] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM.

[46] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[47] Fernando Henrique Calheiros Lopes. Otimizando Compiladores de AspectJ para Java ME, 2007. The author's B.Sc. dissertation.

[48] C. Marche, Paulin C. Mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[49] Marius Marin. Formalizing typical crosscutting concerns. *CoRR*, abs/cs/0606125, 2006.

[50] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[52] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[53] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, PTR, 2nd edition, 2000.

[54] Sun MicroSystems. Java 2 platform, standard edition, v 1.5.0 api specification. Available from http://java.sun.com/j2se/1.5.0/docs/api/ (Date retrieved: August 20, 2007).

[55] Sun MicroSystems. Java Annotations. 2007. At http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[56] John W. Muchow. *Core J2ME Technology and MIDP.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[57] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[58] Peter Müller, Arndt Poetzsch-Heffter, and Gary T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency Computation Practice and Experience*, 2002.

[59] Plácido Antônio Souza Neto. JCML - Java Card Modeling Language: Definição e Implementação. master's dissertation, Universidade Federal do Rio Grande do Norte, Departamento de Informática e Matemática Aplicada Programa de Pós-Graduação em Sistemas e Computaç ao, Novembro 2007. The author's MS.c. dissertation.

[60] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[61] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag Inc., New York, NY, USA, 1994.

[62] Dennis K. Peters and David Lorge Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Trans. Softw. Eng.*, 24(3):161–173, 1998.

[63] Vartan Piroumian. *Wireless J2me Platform Programming.* Prentice Hall Professional Technical Reference, 2002. Foreword By-Mike Clary and Foreword By-Bill Joy.

[64] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Habilitation thesis, Technical University of Munich, January 1997. Available.

[65] Arun David Raghavan and Specification Techniques. Design of a JML Documentation Generator. Technical report, 2000.

[66] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, and Sérgio Soares. A JML compiler based on AspectJ. In *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 541–544, Lillehammer, Norway, april 9-11, 2008. IEEE Computer Society.

[67] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Sérgio Soares, and Leopoldo Ferreira. Implementing Java Modeling Language Contracts with AspectJ. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[68] Frederic Rioux and Patrice Chalin. Effective and Efficient Runtime Assertion Checking for JML Through Strong Validity. In *Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs (FTfJP'07)*, 2007.

[69] Ian Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[70] M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[71] The Coq Development Team. ProGuard: Java Class File Shrinker, Optimizer and Obfuscator, Version 4.2, July 2008. http://proguard.sourceforge.net.

[72] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.1, May 2008. http://pauillac.inria.fr/coq/doc/main.html.

[73] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In *TACAS 2001: Proceedings of the 7thInternationalConference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer-Verlag.

[74] Joe Verzulli. Getting started with JML: Improve your Java programs with JML annotation., march 2003. http://www-128.ibm.com/developerworks/java/library/j-jml.html.

[75] Dean Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *ACP4IS Workshop at AOSD 2006*, pages 27–30, March 2006.

[76] Jianjun Zhao and Martin C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003) of ETAPS'2003*, Lecture Notes in Computer Science, April 2003.

Dissertação de Mestrado apresentada por **Henrique Emanuel Mostaert Rebêlo** à Pós-Graduação em Engenharia da Computação da Escola Politécnica de Pernambuco da Universidade de Pernambuco, sob o título **Implemeting JML Contracts with AspectJ**, orientada por **Ricardo Massa Ferreira Lima** e aprovada pela Banca Examinadora formada pelos professores:

[illegible]

Prof. Sérgio Castelo Branco Soares
Departamento de Sistemas e Computação – Escola Politécnica – UPE

[illegible]

Prof. Paulo Henrique Monteiro Borba
Centro de Informática – UFPE

[illegible]

Prof. Ricardo Massa Ferreira Lima
Departamento de Sistemas e Computação – Escola Politécnica – UPE

Visto e permitida a impressão.
Recife, 30 de maio de 2008.

[illegible]

Prof. Adriano Lorena Inácio de Oliveira
Coordenador do Pós-Graduação em Engenharia da Computação da Escola Politécnica de Pernambuco da Universidade de Pernambuco

POLI UNIVERSIDADE DE PERNAMBUCO
Escola Politécnica de Pernambuco

Prof. Adriano Lorena Inácio de Oliveira
Coordenador Mestrado em Engenharia da Computação
Dep. de Sistemas e Computação - Mat.: 8578-2