Spring 5-22-2017

Library for Writing Contracts for Java Programs Using Prolog

Yogesh Dixit

San Jose State University

Library for Writing Contracts for Java Programs Using Prolog

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Yogesh Dixit

May 2017

The Designated Project Committee Approves the Project Titled

Library for Writing Contracts for Java Programs Using Prolog

by

Yogesh Dixit

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

ABSTRACT

Library for Writing Contracts for Java Programs Using Prolog

by Yogesh Dixit

Today many large and complex software systems are being developed in Java. Although, software always has bugs, it is very important that these developed systems are more reliable despite these bugs.

One way that we can help achieve this is the Design by Contract (DbC) paradigm, which was first introduced by Bertrand Meyer, the creator of Eiffel. The concept of DbC was introduced for software developers so that they can produce more reliable software systems with a little extra cost. Using programming contracts allows developer to specify details such as input conditions and expected output conditions. Doing this makes it easy for the system to assign blame whenever software runs into some erroneous state. Once the blame is assigned it is easier for the developer to detect the cause, so that the appropriate actions can be taken to resolve the issue.

My project develops a library in Java that allows developers to write contracts for their Java programs in Prolog. These contracts are then evaluated by the library with the help of a Prolog dictionary which acts as the database. Prolog's declarative style is a natural fit for writing contracts. With this project, I hope to simplify writing contracts for Java developers. In this paper, I review my implementation. I further discuss some performance tests to show the added overhead.

ACKNOWLEDGMENTS

I am very thankful to Dr. Thomas Austin for his continuous guidance through out the project journey, and also for believing in me. I would also like to thank the committee members Dr. Jon Pearce and Dr. Robert Chun for their valuable time and feedback. Finally, I am very thankful to my family and friends for their unending support and for keeping me motivated throughout my journey of Masters degree.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

Introduction

The software development lifecycle has evolved from just developing software systems to developing robust and reliable software systems. The reliability of software depends on its ability to perform its functions according to the given specifications and to handle exceptional or abnormal cases [1].

The Design by Contract (DbC) methodology provides software developers with the ability to construct reliable software systems without much extra effort. DbC is useful throughout the process of building software, from analysis and design to implementation, documentation, debugging, and even project management[2]. A contract is made up of pre-conditions and post-conditions, which are used for making assertions in the given system. These different conditions define a relationship between the client (end user) and the supplier (software developer). This relationship is said to be broken if any of the conditions do not hold true.

1.1 Contract Conditions

- A precondition is a condition that should hold true when a call is made to the method. If this condition fails, then the call to the method fails and blame can be assigned to the client for providing incorrect input values.
- A postcondition is a condition on a method that should hold true when the execution of the method successfully completes. If it does not, then it can be asserted that something went wrong during execution and blame can be assigned to the supplier for providing an erroneous system that does not work according to the given specifications.
- An invariant is something that needs to be true from the start until the end (throughout the execution), of the call to the method.

Given these constructs, if an application completes the execution without the failure of any of the pre or post conditions provided in the contract, then we can assert that the written code is doing what it is meant for and nothing less or nothing extra [2]. However, the quality of the assertion made depends on how well the contracts conditions are written.

1.2 A Contract for an ATM System

An example of an ATM system given below illustrates the use of contracts. In this example, the ATM system is our supplier and a person using the ATM system is the client. Let's suppose the supplier provides two functions for depositing and withdrawing money.

- Withdraw : Here the client is obliged to enter a non-zero amount to be withdrawn from his/her account, which also should be less than the balance in his/her account. This obligation for the client forms the precondition of our contract. Now, once the client provides the correct amount to be withdrawn, the supplier (ATM system) is obliged to update the balance by decrementing the input amount from it. This obligation for the supplier forms the postcondition of our contract.

```
@Contract(pre_cond = { "isPositive(amount)", "lessThan(
   amount,␣@balance)" },
                              post_cond = { "checkbalance(ans
                                 )" }, source_files = { "
                                 bankprolog.pl" })
         public double withdraw(Double amount)
         {
                  this.balance = balance - amount;
                  return balance;
         }
```

Listing 1.1: Withdraw Contract Example

- Deposit : Here the client is obliged to insert the amount to be deposited, which should be non-zero and less than the maximum limit allowed (let's say $1500). This forms the precondition for the contract of the Deposit function. On successful execution of Deposit function, the supplier is obliged to increase the balance by the deposited amount. This forms the postcondition of the contract.

```
@Contract(pre_cond = {"nonnegative(amount)","
    maxLimitNotBreached(amount)"},post_cond = { "
    checkbalance(ans)" }, source_files = { "bankprolog.pl"
     })
            public double deposit(Double amount)
            {
                        this.balance = balance + amount;
                        return balance;
            }
```

Listing 1.2: Withdraw Contract Example

1.3 Prolog for Contracts

The above two examples show how contract annotations can be specified using preconditions and postconditions to form queries to a Prolog program. This Prolog program acts as database file for the library. When these queries are executed against these rules given in the Prolog program, it results in a boolean assertion that helps the library evaluate if all the contracts were successful or if they had some failures. Listing 1.3 shows an example of a Prolog file that can be used with the bank ATM example illustrated in the above section. The logical syntax of the Prolog code makes it very easy to understand and implement these rules.

```
isPositive(Var):- Var > 0.

lessThan(Var1, Var2):- Var1 < Var2.

checkbalance(Bal):- Bal > 0.

nonzero(Var) :- Var \= 0.

maxLimitNotBreached(Amt):- Amt < 1000.
```

Listing 1.3: Prolog File with Contract Rules

CHAPTER 2

About Contracts

Contract Programming, also known as Design by Contract, is a methodology that can be used for software design and development [2]. This chapter gives a brief history and walks through different aspects of it.

2.1 History and Background

Use of contracts in the form logical assertions was introduced by Pranas in the year 1972 [3]. But, the full fledged contract system was introduced in the form of Design by Contract philosophy to the public by Bertrand Meyer as a part of his programming language named Eiffel [3]. Contract programming is an integral part of Eiffel, but the methodology in itself can be used in any language. Listing 2.1 shows an example of how contracts are written in the Eiffel programming language [2].

```
put (x: ELEMENT; key: STRING) is
            -- Insert x so that it will be retrievable
               through key.
      require
            count <= capacity
            not key.empty
      do
            ... Some insertion algorithm ...
      ensure
            has (x)
            item (key) = x
            count = old count + 1
      end
```

Listing 2.1: Contract in Eiffel [4]

Another important aspect of contract programming is that it helps build and design a robust and fault tolerant software system [5], which is why it has been used repeatedly by many developers in the form of libraries or as a language feature. Contract programming uses a contract agreed upon by both the developer of the software and the user of the software. In Meyer's terms, a developer is the supplier and a user is the client [1]. Here the user of the software must agree and abide to the preconditions that the contract specifies. If the user has done so, then the method should return the results that satisfy the post-conditions. In this way, when a software system runs into an error state it is very easy for one to detect who is to be blamed [6]. If the problem is with the preconditions, then it is the client who is to be blamed [7].

If the problem is with the results, which means the post-conditions are not met, then the software system has a defect on the side of the supplier. This way it is ensured that the software system does what it is supposed to and that any errors can be easily identified.

2.2 Benefits of Design by Contract (DbC)

Developing a software system using the DbC methodology not only makes it more reliable but also has some other added advantages. Using DbC ensures that the software system has a better design, meaningful exceptions, better documentation, and easier debugging [2].

2.2.1 Better Design

Careful use of DbC by software developers can yield better designed systems. This is because a relationship between the client and the supplier is more clearly expressed in the form of conditions. While writing method routines programmers have to clearly think about the preconditions and post-conditions that are declared. This ensures that the system being developed adheres to all the functional specifications. It also makes programmer think about all the exceptional situations that the program may run into while writing the code. This makes it more reliable and at the same time helps to achieve a clearer design.

2.2.2 Meaningful Exceptions

In the case of DbC, a program runs into an exception only when it fails the contract. As a result, it is very easy to identify the exact cause behind its occurrence. For example, if the exception occurred because of a failed pre-condition then the cause is that some inconsistent or bad input values were passed to the method routine. This helps the programmer develop clear and meaningful exceptions that can be easily understood by the client. Given below is an example of an exception thrown as a result of a failed contract.

```
Exception in thread "main" annotations.ContractFailException: Contract failure :
preconditions failed for lessThan(900.0, 600.0)
at annotations.JIPInitializer.checkPreCond(JIPInitializer.java:92)
at annotations.asp.ajc$before$annotations_asp$1$78590bef(asp.java:68)
at com.yd.contractprogramming.Bank.withdraw(Bank.java:34)
at com.yd.contractprogramming.Bank.main(Bank.java:51)
```

2.2.3 Better Documentation

Contracts defined for the system by the developer are part of the code that is visible to the client. The client can read through the contracts that are defined by the supplier for the system, forming an easy means of documentation. Also, it may happen that the user made some modifications to the code but failed to update the document. However, since contracts are an integral part of the code that change if the associated code logic has changed, they form a consistent form of documentation. Also, contracts provide specific and precise information about the method or routine that they are attached to.

2.2.4 Fault Isolation and Easy Debugging

Determining ans analyzing faults once failure is detected is a time consuming and difficult process [8]. When some software program runs into an issue, developers end up debugging the code to find the cause of the failure. These debugging techniques consume a lot of time and developers spend days or weeks isolating the fault. Specifying preconditions and postconditions, helps developers to minimize their debugging efforts. Writing contracts makes it easier for developers to find the cause of the failure and blame assigning becomes easy. When a program without a contract runs into an error, the point where the error occurred is a point inside the code. Whereas if the program has contract associated with it point where the error occurred is the location where contract is specified[8]. Because when a program with contract fails, the reason will be failure of some contract which makes it easy for developers to find the cause and whom to blame [6].

2.3 Limitations of Contracts

Using contracts adds a little cost (time) to the development process and to processing time. This cost includes the cost of writing contract rules (in this case writing Prolog rules) and the cost of executing those contracts at run time for validation checking. These overheads should be considered while making a decision of whether to use the DbC methodology in the development process.

2.3.1 Cost of Writing Contract Rules

Developers will have to invest additional time towards writing contracts along with the overall code writing time. Some developers might neglect this work, which can ultimately result into the poor quality contracts [1]. Also, developers need to think about and invest time in writing contracts in the early phases of the software development process, which many developers might think of as an unnecessary task.

2.3.2 Contract writing Skills

Writing good contracts is a skill [2]. For developers who are not used to DbC, learning to write contracts might prove to be a time consuming process. It might be a very difficult task to find experienced developers who already know about writing contracts as it is not a commonly followed practice. Developers will have to invest some extra time initially to learn and understand the skill of writing well designed contract rules.

2.3.3 False Sense of Security

Using contracts increases code reliability, but it does not make them perfect. It can improve the overall quality of the code, but developers should not assume that their code is free of bugs simply because all of the contracts hold true.

CHAPTER 3

Motivation and Contracts in Other Languages

When thinking of developing new software, people think of using those software development tools and methods that will result in an increased overall productivity for the greater benefit. In the object-oriented world, productivity benefits are not just the result of the correct approach but also depend on how much emphasis is given to quality [4]. Quality of a software system depends primarily on how reliable the software system is [4]. In the object-oriented world, a reliable piece of code is given an extra importance because of its reusable nature. Reusability is an important property of object-oriented programs, which will lose its relevance if the piece of code to be reused is not reliable and correct. Java is one of the most widely used object-oriented programming language that is used for commercial software development [9]. There are different ways such as static typing and automatic garbage collection in Java that help ensure reliability. But this is not enough and we still need a better approach towards developing reliable software systems using Java. This forms the motivation behind the need of a contract system in Java. This chapter focuses on different existing implementations of contract programming in Java and other languages along with their examples.

3.1 Existing Implementations

This section of the chapter lists a few existing implementations of contract programming available in different programming languages.

3.1.1 Contracts in Racket

Contracts in the Racket programming language are mainly applied at module boundaries [10]. Thus, contract constraints and promises are imposed on the values that are exported from the module. Contracts in Racket can be attached to a definition or a function using the `provide` keyword

[10]. Listing 3.1 shows an example that illustrates a basic contract in Racket.

```
#lang racket
(provide (contract-out [amount positive?]))
(define amount ...)
```

Listing 3.1: Basic contract example in Racket

Specification in the Listing 3.1 states that the value of the `amount` variable should always be positive. Every time the client refers to the amount, Racket's contract system keeps a check on the validation of the specified contract. If at some point the amount is bound to a non-positive number or to some value which is not a number, the contract system will signal a contract violation and blame the module breaking the promise [10]. Listing 3.2 shows an example where the contract in Listing 3.1 fails and contract violation error will be thrown.

```
#lang racket
(provide (contract-out [amount positive?]))
(define amount 0)
```

Listing 3.2: Contract violation example in Racket

Racket also allows contracts to be attached to functions [11]. Contracts for functions in Racket are specified using the -> notation.

Listing 3.3 gives an example of a bank module in which contracts are applied over functions of the module using the -> notation.

```
#lang racket
(provide (contract-out
          [deposit (-> natural-number/c any)]
          [balance (-> natural-number/c)]))


(define amount 0)
(define (deposit a) (set! amount (+ amount a)))
(define (balance) amount)
```

Listing 3.3: Contract over functions in Racket

In this example, contracts are applied over two functions : `deposit` and `balance` using the -> notation. The contract specified here states that `deposit` is a function which accepts a non-negative integer and returns some value that is not specified in the contract. And, `balance` is a function that takes in no argument and returns a non-negative integer.

3.1.2 The Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language for Java modules [12]. It provides set of some basic constructs that can be used to write contracts, in precondition and postcondition style for Java programs. Contracts in JML are provided using special annotation comments and are part of the Java code [12]. This contract definition, written in the form of comments are converted into an executable code by the compiler. Thus, if any violation is detected while the code is executing, it can be immediately detected. Simple example of how contracts are written in Java using JML is given in Listing 3.4 [13].

```
//@ requires 0 < amount && amount + balance < MAX_BALANCE;
//@ assignable balance;
//@ ensures balance == \old(balance) + amount;
public void credit(final int amount)
{
  this.balance += amount;
}
```

Listing 3.4: JML Contract Example[13]

In this example, lines starting with the characters `@` denote a JML notation, which will be picked up by compiler and converted into executable code for assertions. JML uses a `requires` clause to implement a precondition and an `ensures` clause to implement a postcondition. The precondition in the 3.4 states that the amount should be greater than zero and the sum of the amount and the balance should remain less than the MAX_BALANCE allowed. The postcondition in the example given above states that the balance result should be equal to the sum of the old balance and the amount. In JML, contracts can also be specified in the form of functions. Example given in 3.5 illustrates this type of contract [12].

```
package org.jmlspecs.samples.jmltutorial;
import org.jmlspecs.models.JMLDouble;
public class SqrtExample {
public final static double eps = 0.0001;
//@ requires x >= 0.0;
//@ ensures JMLDouble.approximatelyEqualTo(x, \result * \
   result, eps);
public static double sqrt(double x) {
return Math.sqrt(x);
}
}
```

Listing 3.5: JML Contract Example with JML function [12]

In the listing 3.5, the postcondition is specified with the help of a function `approximatelyEqualsTo`. `approximatelyEqualsTo` function checks if the produced result multiplied by itself is approximately equal to the input parameter x. If not, the code will fail the postcondition, causing an exception in the code execution. Thus, JML provides set of constructs and tools that allow a Java programmer to specify Eiffel-like contracts for their Java code. On similar lines, my project provides systematic approach of writing contracts in Java using custom annotations. However, there are no extra compilation steps like in the case of JML, that one needs to use. Contracts are execution ready as they are defined using Java annotations unlike JML where they are specified using Java comments.

3.1.3 Contracts.js for JavaScript

Contract.js is a library for JavaScript that provides a way to implement a higher-order behavioral contract system. It uses Sweet.js [14] and lets JavaScript programmers write contracts that dictate how exactly the program should behave. This library implements a runtime check for validity of contracts; if fault occurs, it pinpoints the exact section of the code that caused the failure with a descriptive message [14]. Syntax for writing contracts using Contract.js. is shown in Listing 3.6

```
@ (...) -> ...
function name(...) {
    ...
}
```

Listing 3.6: Contract.js syntax

```
@ ({age: Num}) -> Bool
function isAdult(o) {
    return o.age > 18;
}
```

Listing 3.7: Contract using Contract.js

Listing 3.7 shows an example of a contract written for a JavaScript function using Contract.js. In this example, there are two parts to the contract specified for the function `isAdult`. The first part states that any object o passed as an input parameter to the `isAdult` function should have a property named `age` with a value of type `Num`. In the second part of the contract, a valid input function is obligated to return a `Bool` value at the end of the execution. Here we can relate the first part of the contract as a precondition and the second part of the contract as a postcondition. If the program fails to satisfy any of these two, the program will exit with a well defined error message.

```
isAdult({
    name: "John",
});
```

Listing 3.8: Example of a failed contract

Listing 3.8 shows an example of an error message that is generated when a contract fails to hold true. Here, the `age` property is missing from the object that is passed in as an input parameter to the `isAdult` function, which results in a failed contract. Listing 3.9 shows an error that code in Listing 3.8 generates.

```
Error: isAdult: contract violation
expected: Num
given: undefined
in: the age property of
    the 1st argument of
    ({age: Num}) -> Bool
function isAdult guarded at line: 2
blaming: (calling context for isAdult)
```

Listing 3.9: Error message on failed contract

Another example of a failed contract with respect to the contract in Listing 3.8, where `name` property exists but with an invalid type, is given in Listing 3.10.

```
isAdult({
    name: "John",
    age: "Five"
});

Error: isAdult: contract violation
expected: Num
given: Str
in: the 1st field of
    the age property of
    the 1st argument of
    ({age: Num}) -> Bool
function isAdult guarded at line: 2
blaming: (calling context for isAdult)
```

Listing 3.10: Example of a failed contract

From these examples, it is clear that the contract system enforced by Contract.js is for checking type related errors that may occur in the code. It also provides number basic contracts that check for first order properties [14].

3.2 Contract Library for Java by Neha Rajkumar

Rajkumar [7] developed a contracts library for Java using custom annotations and AspectJ. This library provides a custom Java annotation `@contract` using which a developer can provide preconditions and postconditions over a Java method. These pre and postconditions constraints are then checked at runtime for their validity using AspectJ and reflection. At runtime preconditions and postconditions are validated using Java functions that are executed using custom annotation processing. I will be extending this approach towards building my library, which uses Prolog files to validate contract conditions.

CHAPTER 4

Implementation

This chapter focuses on the implementation details of my project and how to use the library for implementing contracts in Java.

4.1 Java Custom Annotations

Annotations in Java can be used to retrieve information or data about the data. It can be termed as a form of metadata which provides more information at run-time or compile-time about the part of the code that is being annotated. Java annotations always start with the `@` symbol and can be of different forms. Some annotations like the `@override` annotation do not have any elements whereas some annotations like `@SuppressWarnings("unchecked")` come with a element defined inside the parentheses. Java also provides a way to define custom annotations using `@interface`. In my library, I have used this method to create a custom `@contract` annotation, which developers can use to specify the contracts for their Java methods. Listing 4.1 shows code block for creating the custom annotation type `@contract`.

```
package annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD )
@Retention(RetentionPolicy.RUNTIME)

public @interface Contract {
        String [] pre_cond () default "";
        String [] post_cond() default "";
        String [] source_files() default "no file to load";
 }
```

Listing 4.1: Custom Annotation Type

The `Target` element specifies where the annotation type can be used in the code [15]. In the implementation given in Listing 4.1, it specifies that the `@contract` annotation type can be used only with methods. The `@Retention` element specifies until what point in the execution cycle of the code should the annotation of this type be available [15]. In the case of `@contract`, `@Retention` specifies that it will be made available until runtime. The `@Contract` custom annotation has 3 elements: pre_cond, post_cond, and source_files. All these three elements are string arrays; that is, each element can have multiple string values assigned when writing a contract. All these tree elements come with default values associated to them, which makes them non-compulsory elements of the contract type. Use of each of these elements is as follows:

- pre_cond: This element is used to specify the preconditions of the contract.
- post_cond: This element is used to specify the postconditions of the contract.
- source_files: This element is used to specify the Prolog files which should be referred to validate the preconditions and postconditions.

Listing 4.2 gives an example that illustrates how a contract can be written using the previously defined custom annotation type.

```
@Contract(
pre_cond = { "isPositive(amount)", "lessThan(amount, @balance
    )" },
post_cond = { "checkbalance(ans)" }, source_files = { "
    bankprolog.pl" })
```

Listing 4.2: using custom annotation to write contract

In the Listing 4.2, `isPositive(amount)` and `lessThan(amount, balance)` are declared as preconditions, `checkbalance(ans)` is the postcondition. "bankprolog.pl" is a Prolog file that is specified as a source for validating the contract conditions.

4.2 AspectJ and Reflection

Once the contract is specified over a method using the `@contract` annotation, its conditions are validated at runtime. Specifically, preconditions are evaluated just before execution enters the method routine and postconditions should be evaluated immediately after the method has executed. AspectJ, an extension of Java, provides this exact granularity and control over the Java program. AspectJ is an aspect-oriented programming extension created for the

Java programming language [16]. It provides `pointcuts`, which specify well defined moments in the execution of a program, such as a method call [16]. It also provides `before()` and `after()` routines which can be used for implementing precondition and postcondition contracts. Listing 4.3 pointcuts that I am using in this library.

```
//pointcut to catch execution context of any method
pointcut f () : execution (* *(..) ) ;


//pointcut to catch annotations from the code
pointcut g () : @annotation(Contract);
```

Listing 4.3: Pointcut in AspectJ

- f : This pointcut is used to catch the execution context of the running Java program. It uses the wild card syntax "(* * (..))" which specifies that, this pointcut will pick the execution moment of any method in the executing Java program, irrespective of its signature.
- g : This pointcut specifically checks for the elements in the executing Java program that are annotated by the Contract annotation type.

Composing these two pointcuts using the && operation makes it possible to achieve a pointcut that will be picked up only when a method annotated with the Contract annotation is called for execution.

```
// Joining both the pointcuts it will catch
//annotations in the execution context
before () : f() && g()
{
          ...
}

after () returning ( Object objret ): f () && g()
{
          ...
}
```

Listing 4.4: Pointcut Composition and before-after routines

```
before () : f() && g()
{
String[] parameterNames <- Extract names of all the input
   parameters of the method
Object[] arguements <- Extract values of all the input
   parameters of the method
instanceVarNamesList <- Extract all the instance variables
   names
instanceVarNameToValue map <- Create map of name to value for
    all the instance variables of class for the current
   instance

\\ get method annotation of type @Contract
Annotation [] annost = method.getDeclaredAnnotationsByType(
   Contract.class) ;

for each declared annotation from annost[]
  String [] pre_cond <- get preconditions specified in the
     annotation
  String [] source_files <- get prolog source files specified
      in the annotation

  \\load prolog file
  for each source file
    load the prolog file using JIProlog API
  \\convert precondition text into Prolog query by replacing
     variable names with values
  \\example convert pre_cond = { "isPositive(amount)", "
     lessThan(amount, @balance)" } to
  \\pre_cond = { "isPositive(300)", "lessThan(300, 900)" }
  for each precondition in pre_cond
    if precondition contains instance variable name
      get the corresponding value from instanceVarNameToValue
          map
      replace precondition variable name text with its value
         in precondition
    else if precondition contains input parameter name
      inputVarValue <- get the corresponding value of the
         input parameter from arguements[]
      replace input parameter name with its value
         inputVarValue

    evaluate the precondition using JIProlog API
    if evaluation fails
      throw an exception
}
```

Listing 4.5: Pseudocode for before() routine

```
after () returning ( Object objret ): f () && g()
{
instanceVarNamesList <- Extract all the instance variables
   names
instanceVarNameToValue map <- Create map of name to value for
    all the instance variables of class for the current
   instance

\\ get method annotation of type @Contract
Annotation [] annost = method.getDeclaredAnnotationsByType(
   Contract.class) ;

for each declared annotation from annost[]
  String [] post_cond <- get postconditions specified in the
     annotation
  String [] source_files <- get prolog source files specified
      in the annotation

  \\load prolog file
  for each source file
    load the prolog file using JIProlog API
  \\convert postcondition text into Prolog query by replacing
      variable names with values
  \\ example convert post_cond = { "checkPivotValid(ans,@arr)
     ." } to
  \\ post_cond = { "checkPivotValid(2, [5, 2, 6, 126])"}
  for each postcondition in post_cond
    if postcondition contains instance variable name
      get the corresponding value from instanceVarNameToValue
          map
      replace postcondition variable name text with its value
          in postcondition
    else if postcondition contains "ans"
      \\ using ans as keyword for return values
      returnvalue <- get the value from objret
      replace "ans" with its returnvalue.toString()

    evaluate the postcondition using JIProlog API
    if evaluation fails
      throw an exception
}
```

Listing 4.6: Pseudocode for after() routine

4.3 Prolog for Contract Validation

Prolog is widely known for its implementations in the area of Artificial Intelligence and Natural Language Processing. In my implementation of contract programming, Prolog files that contain set of facts and rules will form the basis of contract evaluation. After analyzing and understanding its basic constructs, I found that Prolog's declarative style can be efficiently used to write the set of rules and facts to specify contracts. Once you have these rules in place, the library will query the Prolog file to evaluate the validity of the contract conditions.

4.3.1 Basic Prolog Constructs and Syntax

Prolog has three basic constructs that I will be focusing in this part of the chapter [17].

- Facts
- Rules
- Queries

4.3.1.1 Facts

A fact is a simple statement of the form " chinese (chow_mein). " which results in true or false value. Given this fact, we can now ask `is chow_mein a chinese dish ?` , which will return true. This can be done using Prolog Queries.

4.3.1.2 Rules

A rule is collection of one or more facts. Multiple facts in conjunction or disjunction form the result of a rule. Rules are of the form `nonnegative(Var):- Var >= 0.` . Using this rule, one can query and check if a number is positive or not.

4.3.1.3 Queries

Queries are an important construct of Prolog which allows us to ask questions to the Prolog engine and get answers from it. The Prolog file contains one or more facts and rules based on which our queries will be answered. Listing 4.7 gives an example of a simple Prolog code and some associated queries.

```
likes(sam,Food) :-
          indian(Food),
          mild(Food).
likes(sam,Food) :-
          chinese(Food).
likes(sam,Food) :-
          italian(Food).
likes(sam,Food) :-
          spanish(Food).
likes(sam,chips).


spanish(chicken_chillie).
indian(chicken_curry).
mild(chicken_curry).
chinese(chow_mein).
italian(pizza).
italian(spaghetti).
```

Listing 4.7: Prolog Queries

For the above Prolog program we can formulate different queries as shown in Listing 4.8,

```
?- likes(sam,What).
What = chicken_curry ;
What = chow_mein ;
What = pizza ;
What = spaghetti ;
What = chicken_chillie ;
What = chips.


?- likes(sam,pizza).
true
?- likes(sam,burger).
false.
?- italian(chicken_chillie).
false.
?- chinese(chow_mein).
true.
```

Listing 4.8: Prolog example

4.3.2 Prolog for Contracts

Using the facts and rules introduced in the section above we can specify contract rules effectively. Developers can write their own Prolog files to create custom contracts. For instance, a developer might write a postcondition contract for a quicksort partition function. This contract needs to validate that at each iteration a valid pivot is chosen. This contract can be written using custom annotation like this: `@Contract( post_cond =  "checkPivotValid(ans,@arr)." )`. Once this is done, the developer has to think on the logic that `checkPivotValid` should follow. Here, the logic would check if the selected pivot element is greater than all the left elements and less than all the right elements at each iteration. Once this logic is decided, it can be easily converted into a Prolog rule. Listing 4.9 shows the Prolog code for this contract.

```
sublist(S,M,N,[_A|B]):-
          M > 0,
          M < N,
           sublist(S,M-1,N-1,B).
sublist(S,M,N,[A|B]):-
           0 is M,
          M < N,
           N2 is N-1,
           S=[A|D],
           sublist(D,0,N2,B).
sublist([],0,0,_).

checkPivotValid(Pivotindex,List):-
           Pindex is Pivotindex,
           sublist(S,0,Pindex,List),
           nth0(Pivotindex,List,Pivotelement),
           check_prelist_util(S,Pivotelement),
           countElements(List,Count),
           sublist(N,Pindex+1,Count,List),
           check_postlist_util(N,Pivotelement).
```

Listing 4.9: Prolog example

```
countElements([],0).
countElements([_|Xs],Count):-
          countElements(Xs,Count1),
          Count is Count1+1.

check_prelist_util([H|T], N) :-
     H =< N,
      check_prelist_util(T, N).
check_prelist_util([H|[]], N) :-
     H =< N.

check_postlist_util([H|T], N) :-
     H > N,
      check_postlist_util(T, N).
check_postlist_util([H|[]], N) :-
     H > N.
```

Listing 4.10: Prolog example

checkPivotValid rule from the above Prolog code checks if the pivot selected is greater than all the elements to its left and less than all the elements to its right in the list.

4.4 JIProlog

JiProlog is a Prolog interpreter written in Java [18].As seen in above sections, rules for Java contracts are specified in a Prolog file. We need to evaluate these rules to check the validity of the contracts. For this, we need some way using which we can query the Prolog files from Java AspectJ code. JIProlog provides APIs using which we can establish this connectivity between a Java and Prolog code [18]. Using these APIs we can submit a Prolog query from Java code and get the results. This solves the problem of evaluating contract rules from Java.

Listing 4.11 shows a code snippet that illustrates connection achieved between Java and Prolog code using JIProlog library APIs.

```
public class JIPInitializer {
        public final JIPEngine jip = new JIPEngine();
        public JIPInitializer()
        {
                jip.setDebug(false);
                jip.setTrace(false);
                jip.setEnvVariable("debug", "off");
                try
                {
                        jip.consultFile("
                           default_prolog_library.pl");
                }catch(JIPSyntaxErrorException ex)
                {
                        System.out.println("Exception loading
                            prolog file : " + ex.getMessage()
                           );
                    System.exit(0);
                }
        }
}
```

Listing 4.11: Creating JIPEngine instance

Code snippet given in Listing 4.11 illustrates how to create an instance of the

JIPEngine class, which then will be used for loading a Prolog file and making API calls. JIPEngine is the main class of the JIProlog library [18]. It supplies all the methods required for the Java-Prolog connection and making queries. Code snippet in given in Listing 4.12 illustrates how to load a Prolog file using JIPEngine instance.

```
public void loadFile(String fileName)
        {
                try
                {
                        jip.consultFile(fileName);
                }
                catch(JIPSyntaxErrorException ex)
                {
                        System.out.println("Exception loading
                            prolog file : " + ex.getMessage()
                          );
                        System.exit(0);
                }
        }
```

Listing 4.12: Loading a Prolog file using JIProlog API

JIPEngine's consultFile method compiles and loads the Prolog file, whose name is passed to it as parameter. Once the file is loaded it is ready to be queried using query API's. There are two ways to submit Prolog queries using JIPEngine: Synchronously and asynchronously [18]. For my library I have used synchronous API calls.

Listing 4.13 shows how we can submit a query using JIPEngine instance and read the solution of the submitted query.

```
JIPQuery jipQuery = jip.openSynchronousQuery(queryString);
boolean queryResult = readSolution(jipQuery);
```

Listing 4.13: Prolog Query Using JIProlog API

JIProlog makes it very easy, querying Prolog rules and facts within Java scope and forms the connecting piece of the contract library. Next chapter will focus on a contract example for a QuickSort program and its performance results.

CHAPTER 5
Examples and Performance

This chapter illustrates sample programs using my contract library and also reviews the performance results of it on a Quicksort example. This chapter also provides a conclusion based on the results and dofferent aspects of contract programming discussed throughout this report.

5.1 Sample Contract for Bank System

Listing 5.1 shows an example, which illustrates the usage of my contract library for a Bank class with withdraw and deposit methods.

```
public class Bank {
        private String accOwner;
        private Double balance;

        public Bank(String name)
        {
                this.accOwner = name;
                this.balance = 0.0;
        }

        public Double getBalance() {
                return balance;
        }

        public void setBalance(Double balance) {
                this.balance = balance;
        }
```

Listing 5.1: Java Contracts for Bank class - I

```
@Contract(pre_cond = { "isPositive(amount)", "lessThan(amount
   , @balance)" },post_cond = { "checkbalance(ans)" },
   source_files = { "bankprolog.pl" })
public double withdraw(Double amount)
{
        this.balance = balance - amount;
        return balance;
}

@Contract(pre_cond = { "nonnegative(amount)" , "
   maxLimitNotBreached(amount)"}, post_cond = { "checkbalance
   (ans)" }, source_files = { "bankprolog.pl" })
public double deposit(Double amount)
{
        this.balance = balance + amount;
        return balance;
}
```

Listing 5.2: Java Contracts for Bank class - II

```
isPositive(Var):- Var > 0.
lessThan(Var1, Var2):- Var1 < Var2.
checkbalance(Bal):- Bal > 0.
maxLimitNotBreached(Amt):- Amt < 1000.
```

Listing 5.3: bankprolog.pl

```
public static void main(String [] args)
{
        Bank newAccount = new Bank("Test Account");
        newAccount.deposit(100.00);
        newAccount.deposit(700.00);
        newAccount.withdraw(900.00);
}
```

Listing 5.4: Java Contracts for Bank class

As we can see in the code given in the Listing 5.4, total amount in the `Test Account` after two calls for `deposit` method is 800. When a withdraw method call with 900 as input parameter is made it fails the `lessThan(amount, @balance)` precondition on withdraw method. This condition rule checks if the amount (parameter) that is to be withdrawn from the account is less than the total balance in the account. In our case since amount is greater than the balance it causes the contract failure and program exits with the exception given below.

```
Exception in thread "main" annotations.ContractFailException: Contract failure :
 preconditions failed for lessThan(900.0, 800.0)
at annotations.JIPInitializer.checkPreCond(JIPInitializer.java:92)
at annotations.asp.ajc\$before\$annotations_asp\$1\$78590bef(asp.java:68)
at com.yd.contractprogramming.Bank.withdraw(Bank.java:34)
at com.yd.contractprogramming.Bank.main(Bank.java:51)
```

5.2 Contract for Quicksort and Performance Results

Listing 5.6 shows an example of a quicksort program that uses contract library to validate functionality.

```
private void sort(int low, int high)
{
        if(low < high)
        {
                int pi = partition(low,high);
                sort(low, pi-1);
                sort(pi+1, high);
        }
}
```

Listing 5.5: Quicksort Program with Contract - 1

```
@Contract(pre_cond = { "" }, post_cond = { "checkPivotValid(
   ans,@arr)." }, source_files = {"Sublist.pl"})
int partition(int low, int high)
{
        int pivot = arr[high];
        int i = (low-1);
        for(int j=low;j<=high-1;j++)
        {
                if(arr[j] <= pivot){
                        i++;
                        int temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                }
        }

        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i+1;
}
```

Listing 5.6: Quicksort Program with Contract - 1

In the code given in Listing 5.6 a postcondition contract is used to validate the `partition` function. This contract checks, if the selected pivot is correct at each recursive call. Execution time metrics for this code, against execution time of a quicksort program without any contracts, collected over different size inputs is given in the table 1.

Table 1: Execution Time Metrics for Contract Over Partition Method

(A) Input Size	(B) Execution Time With Contract (In nano-seconds)	(C) Execution Time Without Contract (In nano-seconds)	(D = B/C)
100	2690400443	31262	86509
500	152304489998	369960	411678
1000	652321093085	538378	1211641

In this example, contract is specified over the partition method which has the almost all the logic code required for the quicksort function. Thus contract specified, is validating the pivot value at each recursive call, ultimately resulting into an increased execution time. But, developer can be sure about the correctness of the code and if there is some error, it can be easily traced. Execution time can be reduced if the position of the contract is changed as shown in the code given in Listing 5.7. Now, contract is used over the `sortwrapper` method instead of `partition` method. Thus, contract will validate the result only once the complete execution is over and not at each recursive call. This will still validate, if the result array is sorted and if not will throw an error. But in this code developer won't be able to figure out the error, if there exists one in the partition method if something goes wrong.

```
int partition(int low, int high)
{
        int pivot = arr[high];
        int i = (low-1);
        for(int j=low;j<=high-1;j++)
        {
                if(arr[j] <= pivot){
                        i++;
                        int temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i+1;
}
private void sort(int low, int high)
{
        if(low < high){
                int pi = partition(low,high);
                sort(low, pi-1);
                sort(pi+1, high);
        }
}
```

```
@Contract(pre_cond = { "" }, post_cond = { "ordered(ans)." },
    source_files = {"mypl.pl"})
public int[] sortwrapper(int low, int high)
{
        sort(low, high);
        return this.arr;
}
```

Listing 5.7: Quicksort Program with Contract - 2

Table 2 gives the time metrics for the code given in Listing 5.7.

Table 2: Execution Time Metrics for Contract Over sortwrapper Method

(A) Input Size	(B) Execution Time With Contract (In nano-seconds)	(C) Execution Time Without Contract (In nano-seconds)	(D = B/C)
100	23999612	31262	767
500	206506677	369960	558
1000	472319108	538378	877

There can be cases, like quicksort program example where, code can be validated once using contracts and once validated there should be some way where developer can bypass the contract system to avoid the time overhead. This way developer can be sure that code is correct and time overhead can be avoided. Code snippet given in the Listing 5.8, shows the modified quicksort code to illustrate this. In this code, original `partition` method which has all the sorting logic wont have any contract associated to it. We will introduce another dummy method `partitionWithContract` which will call the actual `partition` method and will also have contract associated.

Once we have this structure in place, now when the developer wants to run the system through the contract system he will call the `partitionWithContract` method and when he wants to bypass the contract system he will call the `partition` method.

```
int partition(int low, int high)
{
        int pivot = arr[high];
        int i = (low-1);
        for(int j=low;j<=high-1;j++)
        {
                if(arr[j] <= pivot){
                        i++;
                        int temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i+1;
}
```

```
@Contract(pre_cond = { "" }, post_cond = { "checkPivotValid(
   ans,@arr)." }, source_files = {"Sublist.pl"})
int partitionWithContract(int low, int high)
{
        return partition(low, high);
}
private void sort(int low, int high)
{
        if(low < high){
        //int pi = partitionWithContract(low,high);
        int pi = partition(low,high);
        sort(low, pi-1);
        sort(pi+1, high);
        }
}
```

Listing 5.8: Quicksort Program with Contract - 3

CHAPTER 6

Conclusion

With the increasing volume and complexity of code involved in different software systems, reliability and robustness have become important aspects of the software development industry. Use of DbC methodology in software development can help in developing such reliable software systems.

The library designed as a part of this project allows developers to write contracts for their Java programs. Although it adds few overheads to the overall execution time, side-effects of using contracts of increased execution time can be reduced using some simple strategies discussed in Chapter 5.

In the future work scope for this library, it can be extended to support invariants along with preconditions and postconditions in the contract definition. Support for, accessing old value of instance variables also needs to be added in the library. This will help developers write more accurate and efficient contracts for their Java methods.

LIST OF REFERENCES

[1] B. Meyer, ‘‘Applying’design by contract,’’ *Computer*, vol. 25, no. 10, pp. 40--51, 1992.

[2] R. Mitchell and J. McKim, *Design by Contract, by Example.* Addison-Wesley, 2002.

[3] R. B. Findler, ‘‘Behavioral software contracts,’’ in *ACM SIGPLAN Notices*, vol. 49, no. 9. ACM, 2014, pp. 137--138.

[4] B. Meyer, ‘‘Building bug-free oo software: An introduction to design by contract,’’ *Availabe at http://www.eiffel.com/values/design-by-contract/introduction/*, 1998.

[5] R. Ceballos, R. M. Gasca, and D. Borrego, ‘‘Constraint satisfaction techniques for diagnosing errors in design by contract software,’’ in *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 2. ACM, 2005, p. 11.

[6] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen, ‘‘Correct blame for contracts: no more scapegoating,’’ in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 215--226.

[7] N. Rajkumar, ‘‘Designing a programming contract library for java,’’ Master’s thesis, San Jose State University, 2015.

[8] L. C. Briand, Y. Labiche, and H. Sun, ‘‘Investigating the use of analysis contracts to support fault isolation in object oriented code,’’ in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 70--80.

[9] ‘‘Why is java the most popular programming language? (oracle university),’’ https://blogs.oracle.com/oracleuniversity/entry/why_is_java_the_most, (Accessed on 04/04/2017).

[10] ‘‘7.1 contracts and boundaries,’’ https://docs.racket-lang.org/guide/contract-boundaries.html, (Accessed on 04/12/2017).

[11] ‘‘7.2 simple contracts on functions,’’ https://docs.racket-lang.org/guide/contract-func.html, (Accessed on 04/14/2017).

[12] G. T. Leavens and Y. Cheon, ‘‘Design by contract with jml,’’ 2006.

[13] ‘‘Java modeling language - wikipedia,’’ https://en.wikipedia.org/wiki/Java_Modeling_Language, (Accessed on 04/30/2017).

[14] “Contracts.js,” http://www.contractsjs.org/, (Accessed on 03/25/2017).

[15] “Java annotations tutorial with examples,” http://beginnersbook.com/2014/09/java-annotations/, (Accessed on 05/01/2017).

[16] “Aspectj - wikipedia,” https://en.wikipedia.org/wiki/AspectJ, (Accessed on 05/01/2017).

[17] “Learn prolog now!” http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlch1, (Accessed on 05/01/2017).

[18] “jiprolog/jiprolog wiki,” https://github.com/jiprolog/jiprolog/wiki, (Accessed on 05/01/2017).