Exploring means to facilitate software debugging

SOLTANI, M.S.

6 The Use of Contracts in Open Source Software

Design by Contract (DBC) is a software development approach in which contracts are formally specified between client and supplier components. Specified contracts can be used in program verification, automated testing, and API documentation, thereby, helping improve software quality. Assertions and other built-in features of programming languages, as a lightweight form of contracts, have been investigated in related work. However, the use of DBC in popular languages has been underexplored. In this study, we present results of an empirical evaluation on the use of contracts in 124 open source projects, written in Java, C++, and Python. Our findings show that the average use of different types of contracts differ depending on the program language. In addition, we derived the following use cases of contracts: checking null conditions, as well as checking equality and semantics of objects, data collections, strings, and numbers. Furthermore, the results of regression analysis shows there is a negative relation between the number of contracts and frequency of defect occurrence in a method. These results are statistically significant for all Java, C++, and Python projects.

6.1 Introduction

Delivering a reliable software product is a pressing concern in software development. Software reliability is especially important when software components are designed to

be reused in various applications. In order to support the goal of developing reliable software, Meyer [162] [163] presents pragmatic guidelines based on the theory of *Design By Contract (DBC)*.

The underlying idea in DBC is that software components are expected to collaborate based on mutual expectations and benefits, which are formally specified in software programs. In a program, if a client routine calls a supplier routine, the client must meet certain requirements on entry, which are typically referred to as *preconditions*. At the exit, the supplier routine may guarantee certain properties, named *postconditions*. *class invariants* are certain conditions which must be always guaranteed. Thus, program contracts are used for various purposes such as static and runtime verification, API documentation, and automated software testing [98].

Recently, Casalnuovo et al. [78] explored the connection between the use of assertions and occurrence of defects in C and C++ projects from Github. They report that the use of assertions does have small (yet significant) effect on reducing the defect density. In addition, they report that assertions tend to be added to methods by developers who have a larger ownership of those methods. Kochhar and Lo [144] performed a partial replication of the study by Casalnuovo et al. [78] on Java programs from Github. Their results confirm the findings reported by Casalnuovo et al. [78]. Furthermore, Estler et al. [98] study the use of contracts in 21 projects developed in Eiffel, C#, and Java. Their findings show that contracts are quite stable and may change infrequently. In addition, they report that there is no strong preference for different types of contracts.

In this study, we intend to broaden the scope of the studies which were previously performed with regards to the use of contracts. Therefore, we aim to investigate how often different types of contracts are used and for what purposes. In addition, similar to the studies by Casalnuovo et al. [78] and Kochhar and Lo [144] which analyze the relation between use of assertions and occurrence of defects, we aim to analyze the relation between the use of contracts and occurrence of defects. Therefore, we devise the following research questions:

- **RQ_1:** How often are different types of contracts used?
- **RQ_2:** For which use cases do developers use contracts?
- **RQ_3:** Does the use of contracts relate to occurrence of defects?

To answer the above research questions, we study three popular programming languages in Github [16] [17]: Java, Python and C++, which support *design by contract (DBC)*. Java and C++ are popular instances of strongly typed object-oriented and semi object-oriented programming languages, respectively. Python is a popular

instance of a high-level interpreted programming language. For each language we selected popular libraries that support DBC, namely: JML [31, 149], Valid4j [37], Cofoja [23], Boost.contract [20], Icontract [27, 146], and Pycontracts [34]. We selected projects that use these libraries from Github, thereby, we created a corpus that contains 124 projects.

For each library we developed a static source code parser which can detect and record the use of contracts. In addition, for each language, we developed a Git commit log parser, which can detect and record methods that are changed in commits. Moreover, we manually analyzed a subset of 1505 (out of 18494) automatically identified contracts that we randomly selected from each project.

Our results indicate that in Java projects, on average, 56% of the contracts are preconditions while 31% of them are postconditions, and 13% are class invariants. These results contradict the results reported by Estler et al. [98] showing that there is indeed strong preference for using preconditions, compared to postconditions and invariants. On the other hand, in C++ projects, on average, 46% of the contacts are postconditions, %29 are preconditions, and 19% are class invariants. For Python projects, on average, 77% of the contracts are preconditions, 16% of the contracts are postconditions, and 7% of the contracts are invariants. In addition, we derived five categories of use cases for contracts: contracts for objects, contracts for data collections, contracts for numbers, contracts for strings, and contracts for null conditions. These categories are further divided into: equality checks on objects, boundary checks and equality checks for numbers, semantic checks and equality checks for strings, size checks and semantic checks for data collections.

Regarding the analysis on the relation between use of contracts and occurrence of defects, we parsed the Git commit logs and filtered the fixing commits using the same heuristics used by Kochhar and Lo [144]. We used Poisson regression analysis [198] on the parsed commits. The results show negative relation between the use of contracts and occurrence of defects. For all Java, C++, and Python libraries, the results are statistically significant.

The contributions in this paper are the following:[1]

1. a set of six open source static parsers written in Python, for six different libraries that support DBC in Java, C++, and Python programs,
2. a set of three open source Git commit log parsers written in Python, for Java, C++, and Python projects,

[1]The corpus of projects, extracted contracts, extracted Git commit logs, the results of Git commit analysis and R scripts are provided through the following DOI link: 10.5281/zenodo.3610696
Upon acceptance of the paper, all contributions will be made publicly available.

3. the collection of 124 Java, C++, and Python projects in which DBC is used,
4. an extensive data set that contains the results of automated contract detection on 124 projects, and automated change detection from Git logs,
5. the results of the manual analysis on 1505 contracts which were automatically detected, and
6. all R scripts used to analyze the results.

The remainder of the chapter is organized as follows: Section 6.2 provides related work. Section 6.3 presents the research methodology. Section 6.4 presents the results. Section 6.5 and 6.6 provide discussion and threats to validity, respectively. Section 6.7 concludes the paper.

6.2 Related Work

In this section, we present the related work about the use of assertions and their effect on defect occurrence, in addition to the related work about empirical studies using projects which are hosted on Github.

6.2.1 Assertion Use and Impact on Quality

Prior to the analysis Casalnuovo et al. [78] performed on the use of assertions in Github projects, Kudrjavets et al. [147] performed an empirical case study on two software components from Microsoft to investigate the relation between software assertions and software faults. According to their observations, with an increase in the assertion density in a file, there is a statistically significant decrease in the fault density.

Later, Casalnuovo et al. [78] studied the use of assertions in Github projects which were developed in C and C++. They found out that assertions are widely used in popular C and C++ projects. They further explored the connection between the use of assertions and occurrence of defects in these projects. Therefore, they report that the use of assertions does have small, but significant, effect on reducing the frequency of defect occurrence. In addition, they report that assertions tend to be added to methods by developers who have a larger contribution to those methods.

Recently, Kochhar and Lo [144] performed a partial replication of the study by Casalnuovo et al. [78] on 185 Java projects from Github. Their results confirm the findings

reported by Casalnuovo et al. [78]. Additionally, they performed manual analysis and identified eight categories of assertions that developers used in the Java projects. The identified categories are: null condition check, process state check, initialization check, resource check, resource lock check, minimum and maximum value constraint check, collection data and length check, and implausible condition check.

In order to support development of reliable software, Meyer [162] [163] provided pragmatic techniques which are based on the theory of Design By Contract (DBC). Contracts are executable form of formal specifications, which are typically expressed as method preconditions, method postconditions, and class invariants. Estler et al. [98] performed an empirical study on 21 Eiffel, C# and Java projects to investigate which types of contracts are used more often, and how contracts are evolved over time. Estler et al. [98] report that they did not observe strong preference for a certain type of contract. However, when preconditions are used, they typically include more predicates than when postconditions are used. In addition, they observe that the use of contracts tends to be quite stable over time.

Moreover, Dietrich et al. [94] report from an empirical study on 200 Java programs and highlight that while the adoption of contracts has been slow in reality, the adoption of lightweight contracts through utilizing built-in features of programming languages and runtime checking has progressed. Thus a wide range of techniques and constructs are used to represent contracts. Often the same program uses different techniques at the same time. Therefore, Dietrich et al. [94] catalogue 25 techniques and tools for lightweight contract checking, using built-in features of the language such as assertions and exceptions.

6.2.2 Empirical Studies on Github Projects

Jiang et al. [133] explore why and how developers fork what from whom in GitHub. Therefore they collect a dataset containing 236,344 developers and 1,841,324 forks. Their observations indicate developers fork repositories to submit pull requests, fix bugs, add new features and keep copies. Developers find repositories to fork from various sources: search engines, external sites (e.g., Twitter, Reddit), social relationships, etc. More than 42 % of developers that they surveyed agree that an automated recommendation tool is useful to help them pick repositories to fork, while more than 44.4 % of developers do not value a recommendation tool. Moreover, their findings indicate that a repository written in a developer's preferred programming language is more likely to be forked.

Kochhar et al. [141] [142] explore 50,000 projects and investigate the correlation

between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams. According to their findings, projects having test cases are bigger in size than projects without test cases. However, as projects get larger the number of tests per line of code decreases. Moreover, they report that projects having bigger team size have higher number of test cases whereas the number of test cases per developer decreases with an increase in the size of the development team.

Kochhar et al. [145] perform a large scale empirical study where they gather a large dataset consisting of popular projects from Github (628 projects, 85 million SLOC, 134 thousand authors, 3 million commits, in 17 languages) to understand the impact of using multiple languages on software quality. They build multiple regression models to study the effects of using different languages on the number of bug fixing commits while controlling for factors such as project age, project size, team size, and the number of commits. Their findings show that in general implementing a project with more languages has a significant effect on project quality, as it increases defect proneness. Moreover, they find specific languages that are statistically significantly more defect prone when they are used in a multi-language setting. These languages are popular languages like C++, Objective-C, and Java.

6.3 Research Methodology

The overarching goal of this paper is to investigate the use of Design By Contract (DBC) in open source projects. More specifically, we aim to identify how often different types of contracts are used as well as the use cases in which contracts are specified. To this end, we develop a set of 6 parsers to statically analyze source code and detect contract specifications.

To derive the use cases in which contracts are used we took the following approach. First, we select a random number of contracts from each project. To preserve a lower bound in the selection process, we made sure to select at least 10 contracts from each project. We then analyze the source code manually to understand the context and rationale for specifying the contracts. We identify a category for each contract as we analyze it. These categories are not predefined, but rather formed as we comprehend the context and rationale behind the usage of the contracts. At the end, we revise the identified categories and if the categories are too fine-grained, we may combine a number of categories to form larger groups of contracts. To extend our perspective and minimize the risk of making mistakes, we ask five independent (non-author) developers to double check our analysis. Each of the developers has at least 5 years

of professional experience in programming with either Java, or Python, or C++.

In addition, we aim to investigate the relation between using contracts and occurrence of defects. To that end, we develop a set of three commit log parsers for Java, C++, and Python. Similar to Casalnuovo et al. [78] and Kochhar and Lo, [144] for each project, we extract the commit logs using *git log -U1 -W*. The *-U1* argument is used to get the commit patches and *-W* is used to get the source contexts in which the patches were provided. After extracting the Git commit logs, we filter the fixing commits, using the same heuristics that were also used by Casalnuovo et al. [78] and Kochhar and Lo, [144], namely: "fix", "issue", "bug", "defect", "incorrect", "error", "fault", "mistake", and "flaw". We then use the commit log parsers to record an account of every method that is changed in the fixing patches.

To analyze the relation between using contracts and occurrence of defects, we use Poisson regression analysis [198], as opposed to Casalnuovo et al. [78] and Kochhar and Lo, [144] who use Hurdle regression model to analyze relation between developer experience, use of assertions, and frequency of defect occurrences. Hurdle regression analysis is used when the minimum value in the dependent variable is 0. Therefore, Hurdle analysis has two components: Hurdle and Count. Hurdle component measures the effect of overcoming the hurdle, which is 0. On the other hand, the count component measures the effect of going from a non-zero value to another non-zero value.

In our case, the minimum value of the defect occurrence is 1 because all methods were recorded from bug fixing commits where the methods were updated. As a result, we use Poisson regression which is similar to regular multiple regression analysis, which is used to model observed counts. Therefore, the possible values of the dependent variable are non-negative integers in this analysis. In our case, the dependent variable is the number of times a method is updated in the fix patches. The independent variable is the number of contracts used in the methods.

We devise the research questions presented in Section 6.1. In what follows, we further describe our approach to developing the contract and commit log parsers.

6.3.1 Automated Contract Detection

Dataset Collection. According to The State of Octoverse [16, 17], Java, C++ and Python are among the ten most popular languages used in Github. At the same time, these languages are popular instances of strongly typed object oriented, semi object oriented, and high-level interpreted programming languages, respectively, which sup-

port the use of DBC. Therefore, we collected 124 projects which are written in these languages from Github.

Table 6.1: Strings used to search for projects

Library	**String**
Cofoja	“com.google.java.contract”
Valid4j	“org.valid4j”
JML	“org.jmlspecs.models”
Icontract	“import icontract”
Pycontracts	“from contracts import”
Boost.contract	“boost::contract::check”

Table 6.2: Statistics of the projects

Language	**Library**	**Projects**	**Files**	**KLOC**
Java	Cofoja	21	6340	767,809
Java	Valid4j	15	1353	72,900
Java	JML	8	15171	1376,265
C++	Boost.contract	48	389146	54239,413
Python	Icontract	9	408	49,010
Python	Pycontracts	23	12787	1839,081

For each language we selected popular libraries that support DBC, namely: JML [31, 149], Valid4j [37], and Cofoja [23] for Java projects, Boost.contract [20] for C++ projects, Icontract [27, 146], and Pycontracts [34] for Python projects. Next, in order to collect the projects using these libraries, we used Github explore [26]. To this end, we first familiarized ourselves with the use of these libraries through their official tutorials. We identified specific key strings, presented in Table 6.1, which are used within source code to import the aforementioned libraries. Therefore, using Github explore, we used the identified strings to search for projects that use a certain library. We then manually investigated the repositories. As long as the contracts were used in the source code and not only in the test code, we included the projects. Later on, contract parsers filter the test files in the projects in order to prevent combining test case assertions with source code contracts in the evaluation. Thereby, we collected 124 projects. Table 6.2 presents the statistics of the collected projects, which are produced by CLOC [19, 22].

Listing 6.1: An example of Cofoja contract specifications.

```
import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;

@Invariant({ "elements != null", "isEmpty() || top() != null" }) // (1)
public class CofojaStack<T> {

  private final LinkedList<T> elements = new LinkedList<T>();

  @Requires("o != null") // (2)
  @Ensures({ "!isEmpty()", "top() == o" }) // (3)
  public void push(T o) {
    elements.add(o);
  }
  ... }
```

Developing Contract Analyzers. Each of the DBC libraries provide unique syntax for specifying the contracts. Using Cofoja, to define class invariants *@Invariants* are used in the source code. Preconditions and postconditions are specified through *@Requires* and *@Ensures*, respectively. To analyze these contracts, every time these tags are included in a source code line, the Cofoja parser expects to arrive at a class or method declaration line. Therefore, every successive line would be checked until a line where a method declaration is defined arrives. Once the line is found, all the predicates defined within the contract blocks are summed and an account of file names, class names, method names, and the identified contracts is recorded. Listing 6.1 is an example from [25] which illustrates how Cofoja contracts are specified.

On the other hand, using Valid4j, to define preconditions and postconditions, *require* and *ensure* blocks are specified within the method declarations, respectively. In addition, for error handling purposes, *validate* blocks are specified, and unreachable code is specified using *neverGetHere* checks. Therefore, Valid4j parser first identifies method declarations. Once a method declaration is found, the parser checks every successive line in order to identify the contracts. Once end of a method is reached, an account of the file name, class name, method names, and identified contracts are recorded. Listing 6.2 is an example [38] of specifying a precondition for a client method that invokes the *Country* constructor.

Using JML, several different types of contracts can be specified, namely: *requires*,

ensures, *signals*, *invariant*, *non_null*, *pure*, etc. Similar to Cofoja, JML contracts are specified before method declarations begin. Therefore the JML parser takes the same approach as earlier presented for Cofoja. Thus, the JML parser first looks for JML tags, and if any of them is detected, then the following method together with an account of the detected contracts are recorded. Listing 6.3 shows an example [30] of specifying jml contracts.

Listing 6.2: An example of precondition specification using Valid4j.

```
public class Country {

  // Use contract to specify that it is the _clients_ responsibility
  // to make sure only valid country codes are given to the constructor.
  // Invoking this constructor with an invalid country code is considered
  // to be a programming error on the clients part.
  public Country(String code) {
    require(isValidCountryCode(code));
    //
  }
   ...}
```

Listing 6.3: An example of contract specification using JML.

```
/*@ protected normal_behavior
    @   assignable size, theStack;
    @   assignable_redundantly theItems, nextFree;
    @   ensures nextFree == 0;
    @*/
  public BoundedStackImplementation( )
  {
    theItems = new Object[MAX_STACK_ITEMS];
    nextFree = 0;
  }
```

Class invariants, preconditions, postconditions, exception guarantees, and old value copies can be specified as contracts, using Boost.contract [21]. Listing 6.6 is an example [21] that shows how these contracts are specified using Boost.contract.

As Listing 6.6 illustrates, Class invariants are declared using the *void invariant() const* method declaration. Therefore, when the Boost parser identifies the invariant declar-

ations, it will expect the successive lines to include assertions. The parser continues to sum the number of assertions until it reaches the end of the method declaration. As for the rest of the contracts, since they are embedded within the method declarations, the Boost parser detects them in a similar way as the Valid4j parser. Thus, every time a method declaration is detected, Boost parser counts the embedded assertions, if there is any.

Using Icontract, it is possible to specify preconditions, postconditions, invariants, and snapshots, using *@icontract.require*, *@icontract.ensure*, *@icontract.invariant*, and *@icontract.snapshot*, respectively.

@icontract.snapshot are similar to *old* contracts using Boost.contract which record old values of the arguments before state transitions. Therefore, this type of contract can be used to verify the state transitions of arguments [27]. Listing 6.5 shows an example of using *@icontract.snapshot* and *@icontract.ensure* contracts.

Preconditions and postconditions can be specified in three ways, using Pycontract. *@contract* decorators can be used to embed the contract specifications. Therefore, every time the Pycontract parser detects a *@contract* decorator, it counts the number of predicates that are defined within the *@contract* block. In addition, if annotations are used, using Python 3, preconditions and postconditions can also be defined within the method signatures. Therefore, if the Pycontract parser detects a *@contract* tag and Python 3 annotations, it looks for possible contract specifications within the method signatures. Otherwise, if the parser detects a *@contract* tag and no Python 3 annotation is detected, then the parser looks for docstrings with *:type:* and *:rtype:* tags which are used to specify preconditions and postconditions. Listing 6.4 illustrates three example of how to specify contracts, using Pycontracts.

Listing 6.4: Three examples of contract specification using Pycontract.

```
### using an @contract decorator
@contract(a='int,>0', b='list[N],N>0', returns='list[N]')
def my_function(a, b):
    ...
### using an @contract tag and Python 3 annotations
@contract
def my_function(a : 'int,>0', b : ' list [N],N>0') -> 'list[N]':
    # Requires b to be a nonempty list, and the return
    # value to have the same length.
    ...

### using an @contract tag and docstrings
```

```
@contract
def my_function(a, b):
    """ Function description.
        :type a: int,>0
        :type b: list [N],N>0
        :rtype: list [N]
    """
    ...
```

Listing 6.5: An example of contract specification using Icontract.

```
>>> @icontract.snapshot(lambda lst: lst[:])
... @icontract.ensure(lambda OLD, lst, value: lst == OLD.lst + [value])
... def some_func(lst: List[int], value: int) -> None:
...     lst.append(value)
...     lst.append(1984) # bug
```

6.3.2 Parsing Commit Logs

Listing 6.7 illustrates an excerpt from git commit log from the library-manager [32] project, which was generated by the *git log -U1 -W* command. As the listing shows, a git diff that is included in a commit is indicated by the line that starts with *diff –git*. When this line is detected, the parsers record the file name that is referred to in this line. After the commit index, a relative number of changes are indicated by + or − symbols next to the file name. A pair of @@ opening symbols indicate chunk headers where the changed lines are shown. Additional context information such as method names may be provided after the closing @@ symbols, which to a large extent depends on the programming language used in the project and if git internal configurations support the language.

Listing 6.7: Excerpt of a sample git commit log.

```
diff --git a/src/main/java/com/mykosoft/librarymanager/options
/common/BookSelectingById.java
b/src/main/java/com/mykosoft/librarymanager/options/common
/BookSelectingById.java
index 27feeac..a905857 100644
```

Listing 6.6: An example of contract specification using Boost.contract.

```
void invariant() const { // Checked in AND with base class invariants.
    BOOST_CONTRACT_ASSERT(size() <= capacity());
}

virtual void push_back(T const& value,
        boost::contract::virtual_* v = 0) /* override */ { // For virtuals.
    boost::contract::old_ptr<unsigned> old_size =
            BOOST_CONTRACT_OLDOF(v, size()); // Old values for virtuals.
    boost::contract::check c = boost::contract::public_function< // For overrides.
            override_push_back>(v, &vector::push_back, this, value)
        .precondition([&] { // Checked in OR with base preconditions.
            BOOST_CONTRACT_ASSERT(size() < max_size());
})
        .postcondition([&] { // Checked in AND with base postconditions.
            BOOST_CONTRACT_ASSERT(size() == *old_size + 1);
});
vect_.push_back(value);

}
```

```
--- a/src/main/java/com/mykosoft/librarymanager/options/
common/BookSelectingById.java
+++ b/src/main/java/com/mykosoft/librarymanager/options/
common/BookSelectingById.java
@@ -10,28 +10,27 @@
 public class BookSelectingById implements BookSelectingStrategy {
     private static ConsoleReader reader = new ConsoleReader();

     @Override
     public Book selectBookFromCollection(Collection<Book> booksByTitle)
         throws IOException{
         Book book = null;

         if (booksByTitle.size() > 1) {
             System.out.println("Multiple books exist with that title!");
             ...

         } else {
-            reader.readLine("One book found");
-            System.out.println(book);
             book = booksByTitle.iterator().next();
+            reader.readLine("One book found\n" + book.toString());
         }

         return book;
     }
 }
```

The context of changes begins after the chunk headers. In Listing 6.7, this is where the *BookSelectingById* class is declared. Commit parsers record class names at this point depending on the programming language being analyzed. For example, Python scripts may or may not define classes. From this point onwards, the parsers analyze each successive line until a method declaration arrives. Once a method declaration is detected, the method name and method signature are temporarily recorded. If inside the method, lines indicate changes (by starting with + or −), then the recorded method name and signatures are stored permanently. The parsers continue analyzing the lines and looking for changed methods until the next patch (*diff –git*) or the next commit is detected.

Listing 6.8 is another example of an excerpt from git commit log from the Mapry project [28] which is developed in Python. As the listing shows, the initial lines which indicate the *diff –git*, commit index, and header chunk follow the same format. In this case, the name of the method where the changes were made is also indicate after the closing @@. However, to detect the defective method, parsing Python syntax is required. This is why we developed three git commit parsers for Java, C++, and Python.

In addition, prior to detecting defective methods, merge commits are filtered. In addition, test files are excluded from the analysis in order to maintain the focus on the use of contracts in source code.

Listing 6.8: Another excerpt of a sample git commit log.

```
diff --git a/docs/source/conf.py b/docs/source/conf.py
index 787f476..086ca83 100644
--- a/docs/source/conf.py
+++ b/docs/source/conf.py
@@ -40,11 +40,13 @@ mapry_meta.__version__
 # Add any Sphinx extension module names here, as strings. They can be
 # extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
 # ones.
 extensions = [
     'sphinx.ext.autodoc',
     'sphinx.ext.doctest',
     'sphinx_autodoc_typehints',
-    'sphinx_icontract'
+    'sphinx_icontract',
+    'sphinx.ext.autosectionlabel'
 ]
+autosectionlabel_prefix_document = True

 # Add any paths that contain templates here, relative to this directory.
```

6.4 Results

In this paper, we aim to investigate the use of contracts in open source projects which are developed in popular programming languages. In this section, we present the

results of our empirical analysis on the use of contracts in open source Java, C++, and Python programs. Thereby, we answer the three research questions presented in Section 6.1.

6.4.1 RQ_1. How often are different types of contracts used?

To answer **RQ_1**, we used the set of parsers we developed in Python in order to automatically detect the specified contracts in Java, C++, and Python projects. As Table 6.3 shows, 57% of the contracts are preconditions in Cofoja projects while 36% are postconditions, and 7% are invariants. 59% of the contracts are preconditions in Valid4j, while 23% are exception guarantees, and 18% are postconditions.

In JML, 35% of the contracts are generic postconditions, while 24% are generic preconditions. In addition, specific contracts are also used as the following: 13% of the contracts in JML are none_null checks, 11% are assignable contracts, 7% are invariants, and 2% are exception signals.

In C++ projects, 46% of the contracts are postconditions, while 29% are preconditions, 18% are invariants, 4% indicate old parameter values, and 2% are exception guarantees.

In Icontract, 65% of the contracts are preconditions, 20% are postconditions, and 15% are invariants. Finally, in Pycontracts, 89% of the contracts are preconditions, while 11% are postconditions.

Table 6.3: Percentages of contract use in Java, C++, and Python projects. **Pre.**, **Post.**, **Inv.**, **Exc.**, **Val.**, **Pure**, **Nnull**, **Sig.**, **Assign.**, and **Nget** indicate "precondition", "postcondition", "invariant", "exception", "non_null", "signals", "assignable", "nevergethere" contracts, respectively. **-** indicates that the type of contract is not supported by the given library.

Library	**Pre.**	**Post.**	**Inv.**	**Old**	**Exc.**	**Val.**	**Pure**	**Nnull**	**Sig.**	**Assign.**	**Nget.**
Cofoja	57	36	7	-	-	-	-	-	-	-	-
Valid4j	59	18	-	-	-	23	-	-	-	-	0
JML	24	35	7	-	-	-	8	13	2	11	-
Boost. contract	29	46	18	4	2	-	-	-	-	-	
Icontract	65	20	15	-	-	-	-	-	-	-	-
Py contracts	89	11	-	-	-	-	-	-	-	-	-

6.4.2 RQ_2. For which use cases do developers use contracts?

To answer **RQ_2**, we randomly selected at least 10 contracts from each project (unless there were less than 10 contracts in a project), which are in total 1505 contracts out of 18494 (8%). We performed manual analysis to identify categories for each contract. At the end, if the categories were too fine-grained, we aggregated them to form a larger group, and vice versa, if a category seemed too coarse grained, we further broke it down to more accurate categories.

As a result, we identify five use cases for contracts, namely: checking null conditions, evaluating objects, evaluating data collections, evaluating strings, and evaluating numbers. The evaluations on the data types are further divided into qualitative and quantitative checks on the values. In what follows, we provide examples we identified during the manual analysis.

Listing 6.9 presents four invariants, from the Cofoja-api project [24], which perform semantic checks on a string. In this case, the string under question is a class name which must always be a simple, qualified, and binary name, according to the invariants.

Listing 6.9: Examples of invariants that evaluate string format.

```
@Invariant({
  "isSimpleName(getSimpleName())",
  "isQualifiedName(getQualifiedName())",
  "isQualifiedName(getSemiQualifiedName())",
  "isBinaryName(getBinaryName())"
})
```

In addition, Listing 6.10 presents examples of contracts, from the Streamline project [36], which provide quantitative and qualitative checks on two lists, `joinPointA` and `joinPointsB`. According to the contracts, the lengths of the lists must be equal while none of the lists can have duplicates. Furthermore, both lists must be sorted, while each member of the lists must be within certain bounds.

Listing 6.10: Examples of qualitative and quantitative checks on lists.

```
public void validateStreamlineConfiguration(
      Integer[] jointPointsA, Integer[] jointPointsB, Integer capacityA,
          Integer capacityB) {
  valid4jValidator.validate(jointPointsA.length == jointPointsB.length,
          "Number of point pairs must be mutually equal");
```

```
    valid4jValidator.validate(ifNoRepetitions(jointPointsA, jointPointsB),
            "None joint points array must contain duplicates");
    valid4jValidator.validate(ifSorted(jointPointsA, jointPointsB),
            "Joint points within arrays must be sorted in increasing order");
    valid4jValidator.validate(ifPointsAreWithinBounds(joint
             PointsA, jointPointsB, capacityA, capacityB),
            "Joint point must be within capacity bounds");
    }
```

Furthermore, Listing 6.11, illustrates contracts, from the MiniUrl project [33], which perform null checks, string semantics checks, and boundary checks on objects. In this case, the contracts require that `originalUrl` and `hashedUrl` arguments are not null, not can `hashedUrl` be an empty string. In addition, the `ttl` which is an immutable instance of `Duration`, cannot be null, 0, or negative.

Listing 6.11: Examples of preconditions and postconditions which perform null checks and boundary checks, respectively.

```
    //Proper domain validation when instantiating
    public ShortenedUrl(URI originalUrl,
                        String hashedUrl,
                        Duration ttl) {
        this.originalUrl = require(originalUrl, notNullValue());
        this.hashedUrl = require(hashedUrl, notEmptyString());
        this.ttl = require(ttl, notNullValue());
        ensure(!(ttl.isZero() && ttl.isNegative())); //Ensure ttl is positive
    }
```

Finally, Listing 6.12, from the INF3143_TP2 project [29], illustrates preconditions which perform semantic checks on an object called `target`. In this case, `target` represents a player, which can neither be null, nor can it be the same as the current instance of a player. In addition, `target` must be still alive so it can be attacked by the current player.

Listing 6.12: An example of preconditions which perform semantic checks on an Object.

```
    @Requires({ "target != null", // A target can not be null.
            "target != this", // A Player can not attack himself.
            "isAlive()" }) // A Player can not attack if he is dead.
```

```
// The attacking player wins.
@Ensures("getXp() > old(getXp())")
public void attack(Player target) {
    int dmg = 20 + this.getStrength() − target.getEndurance();
    Logger.getLogger().log("Player " + this + " attacks " + target);
    target.hurt(dmg);
    this.gainXp(dmg * 2);
}
```

6.4.3 RQ_3. Does the use of contracts relate to occurrence of defects?

To answer **RQ_3**, we performed Poisson regression analysis in order to identify if there is a relation between the use of contracts and occurrence of defects in a method. Table 6.4 presents the results of the regression analysis. According to the table, all estimates are negative, which means that there is a negative relation between the use of contracts and the frequency of defect occurrence. In other words, the results of the regression analysis show that defect density is lower when contracts are used in projects. In addition, as the table shows, all p values are less than 0.05, which means the observed relation is statistically significant.

In addition, we further performed regression analysis for each project separately in order to observe the results of each project independently. The same observation holds for the majority of the projects in Java, C++, and Python. For 2% of the projects which use Cofoja, 3% of the projects which use Valid4j, 7% of the projects which use Boost.contract, and 5% of the projects which use Pycontracts, we did not observe statistically significant effects for using contracts on defect occurrence.

6.5 Discussion

6.5.1 Preference for Different Contracts

Estler et al. [98] investigated 21 contract-equipped Eiffel, C#, and Java projects to identify which types of contracts are used more often, and how contracts evolve over

Table 6.4: The results from Poisson regression analysis to analyze the relation between the number of contracts and frequency of defect occurrence.*** indicates that $p < 0.001$. * indicates that $p < 0.05$.

Library	Coefficients:	Std. Error	Z value	Pr(>|z|)
Cofoja	-5.473595	0.288201	-18.99	$2e^{-16}$ ***
Valid4j	-0.79738	0.35435	-2.25	0.0244 *
JML	-5.451936	0.407862	-13.37	$2e^{-16}$ ***
Boost	-3.61906	0.15130	-23.92	$2e^{-16}$ ***
Icontract	-4.43053	0.46262	-9.577	$2e^{-16}$ ***
Pycontracts	-3.706112	0.226755	-16.34	$2e^{-16}$ ***

time. They report that contracts are stable over time and that they did not observe strong preference for different contracts.

The results of our analysis on 124 Java, C++, and Python projects show, in average, preconditions form the majority of the contracts used in projects which use Cofoja, Valid4j, Icontract and Pycontracts. However, in the case of JML, in average, generic postconditions were used more often compared to generic preconditions. At the same time, JML provides more specific contracts, namely, *pure*, *non_null*, *signal*, and *assignable*, which together, in average, form 33% of the JML contracts that are used in Java projects.

In the case of Boost.contract, in average, postconditions form the majority of the contracts used in C++ projects. Finally, in the case of Icontract and Pycontracts, similar trends as Cofoja and Valid4j are observed, meaning, preconditions are, in average, the majority of the contracts used in Python projects.

Therefore, we observe that our results contradict the results reported by Estler et al. [98] in that we observe strong preferences for using generic preconditions in several cases. Future research may investigate developers' underlying reasons for preferring different types of contracts for specification purposes.

6.5.2 Automated Semantic Analysis of Contracts

We observe that JML, Boost.contract, and Valid4j provide more specific constructs to indicate specific conditions. On the other hand, Cofoja, Icontract, and Pycontract provide standard constructs to specify generic preconditions, postconditions, and invariants.

Using specific contracts facilitates means to develop automated approaches to detect-

ing the semantic use cases of contracts. However, currently there is little support for fine-grain contract specification.

Furthermore, in the case of JML, even though various specific constructs are available for contract specification, in average, generic postconditions still form the majority of the contracts that are used in open source projects. Future work may pursue further advancements in development and analysis of using fine-grained constructs for contracts specifications.

6.5.3 Effect of Using Contracts

As Dietrich et al. [94] highlight, projects which use contracts continue to do so, as a result, they will expand the use of contracts as they evolve. However, in general, contracts are used less than expected. On the other hand, according to the results from Poisson regression analysis, we observe that using contracts does impact the frequency of defect occurrences in open source projects. Despite the advantages of contracts, the use of contracts does not progress fast. Future research may investigate the perception of developers in this regard.

6.6 Threats to Validity

In what follows, we present the threats to internal and external validities, respectively.

6.6.1 Threats to Internal Validity

In order to minimize the risk of having faults in the developed Python code, we reviewed the source code and wrote test cases. However, with this approach, it is not possible to entirely guarantee absence of defects. As mentioned in Section 6.1, we provide a replication package and intend to make all contributions publicly available in the future. We believe taking this approach and fostering openness increases the possibilities to identify potential faults.

With reards to defect identification from commit logs, we used the same approach as Casalnuovo et al. [78] and Kochhar and Lo, [144], instead of using defect databases. This approach entails false positives and or false negatives, similar to links to defects in defect databases which may also contain errors according to Bird et al. [66]. On the other hand, according to Bissyandé et al. [67], not all projects use issue trackers.

Also, not all bugs are recorded in issue trackers. In our case, a defect may appear in several bug fixing commits. In addition, if bug fixing commits are not sufficiently accurate, bug fixing commits may be missed.

During the manual analysis process, we asked feedback from five independent developers who had several years of professional experience in programming. Incorporating the feedback from professional developers helped us minimize the risk of making mistakes in our manual analysis.

6.6.2 Generalizability of Findings

We used Github to collect the projects. Github is a popular platform which hosts over 96 million repositories [35]. Over 31 million developers contribute to projects which are hosted on Github. In addition, we studied three different programming languages and formed a large corpus of projects.

Moreover, we used statistical regression model to perform analysis on the relation of defect occurrence and contracts use. Since the results of the regression analysis are statistically significant, it is possible to extrapolate the observations reported in this paper.

However, while we tried to minimize the threats to generalizability, we cannot argue that our results generalize to industrial closed-source software. Therefore, future studies may replicate this study in closed-source projects and compare the outcomes.

6.7 Conclusion

Delivering robust and reliable software is a pressing demand for software developers. These qualities matter even more when software components are developed to be reused in many applications. To support development of reliable software, Meyer [162] [163] provides pragmatic techniques and guidelines based on the concept of Design By Contract (DBC).

DBC is based on the idea that software components collaborate with each other based on mutual expectations and benefits. Thus is a client component uses services of a supplier component, the client needs to guarantee preconditions of the supplier. The supplier in return guarantees certain conditions at the time of delivering the services. Thus, contracts are a kind of formal specifications that can be used for different purposes such as automated testing, and static and dynamic verifications.

Despite the advantages of using contracts, contracts are used less than expected according to Dietrich et al. [94]. On the other hand, assertions, exceptions and other built-in features of programming languages are used as lightweight contracts. Therefore, the use of assertions and other lightweight contracts have been studied in depth by Kudrjavets et al. [147], Casalnuovo et al. [78], and Kochhar and Lo [144]. However, the use of contracts has been underexplored to the best of our knowledge. In this paper, we aim to investigate the use of contracts in open source projects. In addition, we aim to analyze the relation between using contracts and frequency of defect occurrences. Therefore, we studied 124 Java, C++, and Python projects. We developed a set of contract parsers which we used to parse contract-equipped source code. The results of automated contract detection show in most cases, except for C++ projects, the use of preconditions, in average, forms the majority of the contract specifications. We derived five categories of contracts which are: null checks, checks on objects, checks on data collections, checks on strings, checks on numbers, Moreover, we use Poisson regression analysis to identify the relation between the use of contracts and defect occurrences. The results of the regression analysis confirms there is a statistically significant negative relation between the use of contracts and defect occurrences. Thus, when methods use contracts, the rate of defect occurrences becomes smaller.