CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The spec is out there

Extracting contracts from code

Master's thesis in Computer Science

Christoffer Medin
Pontus Doverstav

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017

The spec is out there

Extracting contracts from code

Christoffer Medin
Pontus Doverstav

Department of Computer Science and Engineering
Division of Formal methods
Chalmers University of Technology
Gothenburg, Sweden 2017

The spec is out there
Extracting contracts from code
CHRISTOFFER MEDIN
PONTUS DOVERSTAV

Supervisor: Carlo A. Furia, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Division of Formal methods
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

The spec is out there
Extracting contracts from code
CHRISTOFFER MEDIN
PONTUS DOVERSTAV
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

A contract is a formal specification of the properties of a method. It has many uses in testing, verification and documentation. Despite all these benefits, developers almost never write contracts for their code due to the large amount of time required to write correct and expressive contracts. The project described in this thesis sets out to evaluate the possibility of extracting contracts from code using syntactic analysis, which is a not yet evaluated technique. Syntactic analysis is done by looking at the operations performed by code, rather than the result of those operations. Contracts were extracted by first parsing target source code into an abstract syntax tree, which was then analyzed to find defined code patterns which signaled the presence of some implicit contract. The information found within these patterns was then extracted and written as an explicit contract. In experiments involving 4 projects over 35 000 methods, our syntactic analysis tool extracted close to 60 000 behaviors, of which one-third were regarded as successful. The results are promising, even though performance of the syntactic analysis was somewhat inconsistent, it could produce contracts on a comparable level to those written by a human.

Keywords: formal specification, contracts, syntactic analysis

Acknowledgements

We would like to thank our supervisor Carlo A. Furia for his great support during this project. His ideas and feedback have been vital for us to succeed with this project. We would also like to thank Victor Nilsson for taking his time writing contracts for us to compare our results with.

Christoffer Medin & Pontus Doverstav, Gothenburg, July, 2017

Contents

1
Introduction

Bertrand Meyer coined the term Design by Contract based on the idea of defining the properties of a function as a contract, stating what the function requires (preconditions) and what it ensures (postconditions) [1]. The preconditions typically depend on parameters to the function and object fields and tell the user what they need to supply the function in order to use it. The postconditions on the other hand tell the user what they can expect from the function given that they have satisfied the corresponding preconditions.
Contracts have uses other than telling the user which inputs the function allows. A contract can provide useful information when testing a program as it can be used as specification to determine if the method behaves correctly or not [2].
Formal verification is another very powerful way to use a contract. Formal verification is done by fully specifying the desired behavior of a function using contracts and then using the specification with the actual function to prove that the function will behave according to specification. This provides strong guarantees that the code is doing what it is expected to. In contrast to merely testing the code this guarantee of correctness is for all possible inputs rather just the ones that were tested.

1.1 Purpose and goals

In this thesis we will explore the possibilities of automatically extracting these kinds of implicit contracts from code. Since the contracts are used to describe the code, the idea that you could get this description by analyzing the code does not seem too far off. Finding contracts by analyzing code or other artifacts such as comments has already proven to be successful [2].
The project will consist of two parts: first we will develop a tool, henceforth called SpecIT, which extracts contracts from code using syntactic analysis. Then we will test SpecIT to evaluate the contracts generated. This evaluation will consist of statistical analysis, looking at the amount of contracts generated, their size etc. We will also evaluate the contracts readability and in general analyze their quality manually.

1.2 Scope and limitations

This thesis will focus on syntactic analysis of code and not semantic analysis such as static analysis. Static analysis consists of testing the code without executing it,

this can be done via symbolic execution which uses contracts to prove properties about programs.
The line between syntax and semantics is sometimes fuzzy but the main difference is that the syntax is what the code says whereas the semantics is what the code does. By not doing semantic analysis we will not create contracts that capture the result of operations, only the operations themselves. For example doing mathematical operations on an integer would result in the semantic contract being the result of the operations, where as the syntactic contract would be the mathematical operations on the integer. A very basic example of this is shown in figure 1.1.

```
int f(int x){
    return x*x/x
}

/* Syntactic contract
 * \ensures result == x*x/x

 * Semantic contract
 * \requires x != 0
 * \ensures result == x
 */
```

Figure 1.1: Syntactic vs Semantic

It is important to note that the syntactic analysis performed in this project is not able to produce contracts that are expressive enough for any useful formal verification. Since the contracts will be generated purely through syntactic analysis of the code and not its intent, they will not be expressive enough to do formal verification of non-trivial properties. However, they may be used as a basis to construct detailed specification and as aid when doing automated testing.
This study will not evaluate the value of using the extracted contracts when doing automated testing, although this is a possible use of the contracts that might be worth exploring.
The projects analyzed will be limited to those freely available and will focus on those who already have contracts as it provides a good base for comparing the extracted contracts to manually created contracts.

2
Design by Contract

Design by contract is a software development methodology which attempts to increase the reliability in software systems, where reliability is defined as correctness and robustness, or as the absence of bugs [1]. In contrast to defensive programming which tries to achieve reliability through a plethora of checks, design by contract relies on the notion of *contracts*.
Contracts can be seen as an agreement between a client and supplier, or in the case of software systems, an agreement between a caller and a called routine. A contract protects both sides - it specifies both how *much* and how *little* is expected to be done. The caller knows what kind of result can be expected and the callee knows what tasks it can be expected to perform. Both parties are subject to some obligation, but in turn usually gain some benefit.
Contracts specify individual routines, or methods, but included in design by contract are also *class invariants*. Class invariants are properties that apply to all instances of a class and to all methods of the class. An invariant must be satisfied by every instance of the class after creation and it must be preserved by every routine of that class. If an invariant is satisfied upon entry it must also be satisfied upon exit. An example of an invariant can be seen in figure 2.1, which describes that for every element in a double-linked list, the previous element of an element's next in a double-linked list must be the element itself.

```
class Element {
    Element next;
    Element previous;

    //@ invariant next != null --> next.previous == this;

    ...
}
```

Figure 2.1: An example of a class invariant

The mechanism for expressing contracts and class invariants are called assertions. The assertions for contracts can be separated into two classes, pre- and post-conditions. As is implied by their names, preconditions are assertions that must hold before a method is executed and postconditions must hold after execution is finished. The responsibility that preconditions hold is given to the caller, and responsibility for postconditions is handed to the called method. If some case requires special treat-

ment but is not included in the contract it must be handled by the called method. A clause not stated in the contract cannot be guaranteed by the caller. What should be demanded as a precondition and what should be checked manually by the method is a design-decision and can vary from case to case. In some methods you may wish to handle special, or exceptional, cases in some way while in other you would just reject them. This should guide the decision whether to include it in the contract or not.

```
/*@
    public normal_behavior
    requires d > 0;
    ensures x == \result * d;
@*/
public double division1(int x, int d){
    return x/d;
}

/*@
    public normal_behavior
    ensures d > 0 ==> x == \result * d;
    ensures d <= 0 ==> \result == -1;
@*/
public double division2(int x, int d){
    if(d <= 0){
        return -1;
    } else {
        return x/d;
    }
}
```

Figure 2.2: Two methods with differently demanding contracts

In figure 2.2 two almost identical methods can be seen, but with responsibility placed on either the caller or the callee. As no precondition is specified in the second method, it is expected that the responsibility of checking that the input is correct lies on the called method instead of the calling method. In the same way, when a precondition is stated in a contract as in the first method, responsibility lies on the calling method to make sure that it is fulfilled.

3
Programming languages

In order to do syntactic analysis a decision has to be made on what language to analyze. The considered languages are C#, Eiffel and Java. These languages have previously appeared in earlier studies on the usage of contracts [3], having been the target for automatic contract generation [4][2] and also been used to examine the value of writing stronger specifications [5]. We will now discuss what criteria were used to determine which of these languages was better suited for this project and how well the languages fulfilled these criteria.

3.1 Selection criteria

The first criterion to be met is supporting contracts, this does not have to be native support, however in that case there must exist some extension or tool which allows support for contracts. The second criteria is having available testing tools with support for contracts. The third criteria is having a substantial amount open source projects available in order to find a good basis for evaluating SpecIT.

3.2 Selection of language

Eiffel meets most of the criteria well. It has support for contracts such as pre- and postconditions and assertions, however it lacks functionality in the form of quantifiers. Eiffel has support for contract checking using contracts with EiffelStudio, which also serves as the official IDE [6]. The main problem for Eiffel is that there is a very limited amount of open source projects available, in fact on GitHub we only managed to find around 300 projects using eiffel. Additionally, since Eiffel supports contracts natively there may not be many implicit contracts to extract that are not already written explicitly.
Java was deemed a good fit for the project due to its popularity and support for contracts using the extension JML. JML allows for contracts such as pre- and postconditions and assertions, which also support quantifiers and variables unique to those conditions and assertions [7]. The tool JMLUnitNG provides automatic testing using with support for contracts [8]. The popularity of the language allowed for a large amount of projects to select from when identifying suitable projects to evaluate SpecIT on.
C# shares many of the good qualities that Java has. It has support for contracts called Code Contracts, it supports automatic testing using these contracts and it has a large open source presence.

Both Java and C# were both deemed to be good target languages. The tiebreaker was the authors' previous experience with Java which should reduce the amount of time spent on researching the language and its contracts.

4

Java Modeling Language

The Java Modeling Language (JML) is formal behavioral interface specification language for Java [9]. This means that using JML *interfaces* and *behaviors* of Java code can be specified. Here, interfaces refer to the names and static information found in Java declarations and behaviors to how the code behaves once executed. As such, JML can be used as a design by contract tool for Java.
In addition to supporting design by contract, JML can also be used as a tool for model-based specification [7]. However, the focus of this thesis is on contracts, and thus features related to model-based specification will not be explained. The contracts generated in this thesis will be simpler, and therefore only need a subset of JML's functionality [10], which will be explained briefly below.

```
public class Example{
    private boolean error = false;
    private int timesInverted = 0;

    public int abs(int b){
        if (b > 0){
            return b;
        } else if (b < 0) {
            timesInverted++;
            b = b * -1;
            return b;
        } else {
            error = true;
            throw new IllegalArgumentException();
        }
    }
}
```

Figure 4.1: Example of code

4.1 normal & exceptional behavior

In the event that a method has several distinct behaviors, the specification can be split into different behaviors as well. Behaviors can be defined as general, normal or

exceptional, where normal and exceptional behaviors mostly are general behaviors with some syntactic sugar added.
Normal behaviors have an implicit `signals` clause added (`signals (java.lang.Exception) false`) which specifies that this behavior cannot throw any exception. Exceptional behaviors have an implicit `ensures` clause added (`ensures false`). Additionally, further `ensures` clauses are not allowed, meaning that an exceptional behavior specifies a behavior that must throw an exception in order to terminate. Applying this to the code seen in figure 4.1 gives a contract as seen in figure 4.2.

```
/*@
    public normal_behavior
    (* Further specification *)

    also

    public normal_behavior
    (* Further specification *)

    also

    public exceptional_behavior
    (* Further specification *)
@*/
```

Figure 4.2: Specification of the various behaviors identified in method abs in figure 4.1

4.2 requires & ensures

The `requires` and `ensures` clause are used to specify pre- and post-conditions of a contract, respectively. These are the most basic blocks around which contracts are generated. Each clause consists of a boolean predicate, and several clauses in one behavior will be evaluated as one clause with a predicate that is the conjunction of all predicates. Related to post-conditions are the clauses `\result` and `\old`, which are used when writing specification about return values or values held in a methods pre-state. Figures 4.3 shows how the contract changes with the addition of pre- and post-conditions.

4.2.1 \result

In a method specification, the keyword `\result` denotes the value returned by the method. Its value is the same as that returned by the method and it is used together with an `ensures` clause.

4.2.2 \old

When evaluating the contract of a method, JML considers two states, the pre-state and the post-state. As their names imply, pre-state refers to the state that existed before the method call, and post-state refers to the state after the method has completed its execution. Through using `\old`, one can refer to the value of an object or expression in the pre-state. An example of `\old` being used can be found in figure 4.3.

```
/*@
    public normal_behavior
    requires b > 0;
    ensures \result == b;

    also

    public normal_behavior
    requires b < 0;
    requires !(b > 0);
    ensures timesInverted = \old(timesInverted) + 1;
    ensures \result == b * -1;

    also

    public exceptional_behavior
    requires !(b < 0);
    requires !(b > 0);
@*/
```

Figure 4.3: Further specification of method abs in figure 4.1 by adding pre- and post-conditions

4.3 signals_only & signal

When generating contracts for methods that might throw exceptions, the `signals` and `signals_only` clauses are used. The `signals_only` clause is used to specify what exceptions may be thrown by a method, while the `signals` clause specifies what holds in the post-state after a certain exception is thrown.

In figure 4.4 is an example of how `signals` and `signals_only` may be used. When `b` is equal to zero, it is specified that an exception must be thrown, that exception must be of the type `IllegalArgumentException` and in the post-state, `error == true` must also hold.

```
/*@
    public normal_behavior
    requires b > 0;
    ensures \result == b;

    also

    public normal_behavior
    requires b < 0;
    ensures timesInverted = \old(timesInverted) + 1;
    ensures \result == b * -1;

    also

    public exceptional_behavior
    requires b == 0;
    signals_only IllegalArgumentException;
    signals IllegalArgumentException (error == true);
@*/
```

Figure 4.4: Further specification of figure 4.1 through use of `signals` and `signals_only`

4.4 assignable

The `assignable` clause is used for framing, i.e. to specify what locations or variables can be assigned to during execution of a method. It can either be given a list of field names or one of two special values, `\everything` or `\nothing`, which specify that either every location or no location can be assigned to. Figure 4.5 shows how `assignable` can be used to specify figure 4.1. Note that although `b` is assigned a value in one of the behaviors, it is not used in an `assignable` clause. This is because method paramaters are not allowed to be used in such clauses.

4.5 Pure methods

In order to use a method in the specification of another method in JML, it has to be specified as `pure`. A `pure` method is defined to not have any side effects when executed, i.e. it has the specification seen in figure 4.6.

For a method to be considered `pure` it has to always terminate normally or throw an exception. Additionally, it has to be deterministic, i.e. when called in a given state, it always returns the same value. This is only an issue when dealing with either randomness or concurrency as they can modify the results of statements and in which order they are executed.

```
/*@
    public normal_behavior
    requires b > 0;
    ensures \result == b;
    assignable \nothing;

    also

    public normal_behavior
    requires b < 0;
    ensures timesInverted = \old(timesInverted) + 1;
    ensures \result == b * -1;
    assignable \nothing;

    also

    public exceptional_behavior
    requires b == 0;
    signals_only IllegalArgumentException;
    signals IllegalArgumentException (error == true);
    assignable error;
@*/
```

Figure 4.5: Completed specification of method abs in figure 4.1

```
/* diverges false;
 * assignable \nothing;
 */
```

Figure 4.6: Implicit specification of a `pure` method

4.6 Visibility of fields and methods

In order for a variable or method to be usable in a JML contract it must have public visibility. In general these fields and methods are non-public for a reason and thus changing their visibility would have unwanted consequences. JML has a workaround for this namely annotating the variable of method with `spec_public` which allows JML to consider it as public for specification purposes. An example of this can be seen in figure 4.7.

```
    private /*@ spec_public @*/ boolean error = false;
    private /*@ spec_public @*/ int timesInverted = 0;
```

Figure 4.7: Specification of fields in figure 4.1 to allow usage in contract.

5
Code Patterns

In order to extract implicit contracts from code syntactically, programming patterns containing this information must be defined. This section presents, in a general form, the different patterns identified. These transformations will be the basis for the contracts generated.

5.1 Assertions

An `assertion`-statement is used to test assumptions about a program by documenting what it is believed to be true at that point in the code. This gives information about what is expected by the program and can be used to construct contracts. Depending on how or were an assertion is used, it can give different information about the expected behavior.
In figure 5.1 is an example of two different placements of assertions and how this affects the corresponding contract. In `f1` the use of the assertion in the beginning signifies that it is linked to some precondition, while in `f2` the assertion is used after some action and thus signifies some postcondition. Due to this, the two methods will generate different contracts.

5.2 Conditionals

In any case where a return statement can be accessed through a conditional statement, e.g. an `if`-statement, this condition directly influences the behaviour of the code and thus the contract. In figure 5.2 is an example of a conditional and its corresponding contract.

5.3 Switch-case

When a standard `switch-case`-statement is written such as the one in figure 5.3 it can be seen as a large `if-else`-statement as can be seen in figure 5.4. From this the contract in figure 5.5 is generated. It is however possible to write a `switch-case` without neither break nor return statement in each case. This results in the case below being executed as well, regardless if it matches the case or not, this will continue until a case with break or return is found or the end of cases. Since that kind of `switch-case` is not as common and very complex to generate a contract for, it was omitted from this project.

```
/*@
    public normal_behavior
    requires P(x);
@*/
public Object f1(int x){
    assert P(x);
    ...
    return result;
}

/*@
    public normal_behavior
    ensures P(x);
@*/
public Object f2(){
    ...
    assert P(x);
    return result;
}
```

Figure 5.1: Example of different uses of `assertion`-statements and their contracts

5.4 Exceptional behaviour

The handling of errors and exceptional behaviours in code contain information about what circumstances need to be met for the code to exhibit such exceptional behaviour. Figure 5.6 is a simple example of exceptional behavior.

5.5 Assignment

In cases where an assignment is done to a instance variable the statement influences the behaviour of the code and should thus be part of the contract. An example of this can be seen in figure 5.7.

5.6 Null checks

Whenever fields or methods of an instance are accessed via the dot operator, that instance must be instantiated, i.e. not null. Thus, whenever a method uses the dot operator on either a parameter or an instance field, a pre-condition can be added which states that particular instance can not be null, as is shown in figure 5.8.

```
/*@
    public normal_behavior
    requires b1;
    ensures \result == a1

    also

    public normal_behavior
    requires !b1;
    ensures \result == a2
@*/
public String f(boolean b1){
    if ( b1 ) {
        return a1;
    } else {
        return a2;
    }
}
```

Figure 5.2: Example of an `if`-statement and its contract

```
string f(int a){
    switch ( a ) {
        case 1: return "hello"
        case 2: return "world"
        default: return "!"
    }
}
```

Figure 5.3: Example of a switch-case statement

5.7 Larger example

In this section, a larger, more interesting example is presented where various patterns are used together. Given a method such as the one seen in figure 5.9 which shows of the different patterns combined; several behaviors are created due to the `if-else` statement, one of them being exceptional as `IllegalArgumentException` is thrown.

5.8 Loops

Due to limitations of syntactic analysis little information can be extracted from loops. It cannot be determined if a loop will run at all or forever. Due to this limitation the best that can be accomplished is to try to retain as much information as possible while still ensuring soundness. This can be done by discarding the value of variables being modified within the loop and setting the purity of the method.

```
string f(int a){
    if(a==1){
        return "hello";
    } else if(a==2){
        return "world";
    } else {
        return "!"
    }
}
```

Figure 5.4: Figure 5.3 as an if-statement.

```
/*@
    public normal_behavior
    requires a == 1;
    ensures \result == "hello";

    also

    public normal_behavior
    requires a == 2;
    ensures \result == "world";

    also

    public normal_behavior
    requires a != 1 && a != 2;
    ensures \result == "!";
@*/
```

Figure 5.5: The contract that describes the behavior of figure 5.4

5.9 Try-Catch

There is no way to express a statement non-explicitly throwing an exception in JML, which results in the `try-catch` pattern being impossible to model. For some exceptions it is possible to express this behavior regardless, which can be seen in figure 5.10. The contract would have to express the fact that if either of the statements `a.toString()` or `b.equals(a)` throws a `NullPointerException` the result is null. The most likely way for the exception to be thrown is if either `a` or `b` is null which makes the contract `ensures a == null || b == null ==> \result == null` seem reasonable to produce. However the exception could be thrown in `toString()` or `equals(a)` which would create a lot of issues. The antecedent `a == null || b == null` would then not be expressive enough and every statement in `toString()` and `equals(a)` would have to be examined to see if they might throw `NullPointerException` and in that case be added to the antecedent. The specific

```
/*@
    public exceptional_behavior
    requires b1;
    signals_only Exception;
    signals (Exception) b1;
@*/
f(var b1){
    if ( b1 ) {
        throw new Exception();
    }
    // Rest of method
}
```

Figure 5.6: Contract generated from exceptional behavior

```
/*@
    public normal_behaviour
    ensures a == 2;
    ensures b == 3;
    assignable a,b;
@*/
public void f(){
    a = 2;
    b = 3;
}
```

Figure 5.7: An example of a contract created from assignments

analytic behavior for `NullPointerException` would also have to be replicated for each expressions to cover that specific case and would only work if the catch clause catches a specific exception.

5.10 Method calls

A very common occurrence in Java programs is methods calling methods. Since these methods might return values and modify fields it becomes important for the soundness of the contract to include the contract of the methods called. A basic example is shown in figure 5.11 where `doubleinc()` uses the method `inc()` twice. The modifications `inc()` does to `i` is thus applied twice.
Calling methods without meaningful contracts presents an issue as it at that point exists no certainty of their return values or what fields they modify. In order to ensure soundness a contract will not be created for a method if it calls a method for which a good contract cannot be generated.
When using JML to prove a methods adherence to its contract using extended static checking, it is required that all methods used in the contract are pure. Since the

```
public class CustomInteger{

    Integer i;

    /* Rest of class */

    /*@
        public normal_behavior
        requires i != null;
        requires sub != null;
    @*/
    public int difference(Integer sub){
        return i.intValue() - sub.intValue();
    }
}
```

Figure 5.8: Example showing how null checks can be extracted from code

contracts generated by this project will not be used for this kind of verification it was decided to keep the impure method calls in the contracts which increases the expressiveness of the contract.

5.11 Using standard libraries

There are methods that are too advanced for syntactic analysis to provide any meaningful contracts which can cause problems when these methods are called. The lack of information about these method calls means that a lot of information has to be discarded in order to ensure soundness of the contract being generated. This can be remedied by writing manual contracts for these type of methods. However, this is a difficult and time consuming task which is only worth the time if the method sees extensive usage. Examples of such methods are those in the standard Java library which is used across a large amount of projects.
An example of a method that is too complex for syntactic analysis is `Math.random()`. syntactic analysis could not produce a meaningful contract as it only makes another method call, which in turn contains loops which makes it impossible to say anything about the return value. According to the documentation of the method it "Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0." [11]. Using this, a contract can be manually added for it, such as the one in figure 5.12.
Another example is the `Math.sqrt(double a)` method which implementation calls the method `StrictMath.sqrt(double a)`. This method is native which means that the implementation is not written in Java, nor easily available, making syntactic analysis of the original method pointless. However the contract itself is not very complex as seen in figure 5.13.

```
/*@
    public normal_behavior
    requires b > 0;
    ensures \result == b;

    also

    public normal_behavior
    require b < 0;
    ensures \result == b*-1;

    also

    public exceptional_behavior
    requires !(b < 0) && !(b > 0);
    signals_only IllegalArgumentException;
    signals (IllegalArgumentException) (!(b < 0) && !(b > 0));
@*/
public int f(int b){
    if (b > 0){
        return b;
    } else if (b < 0) {
        b = b * -1;
        return b;
    } else {
        throw new IllegalArgumentException();
    }
}
```

Figure 5.9: Example showing how various patterns create a complete contract

```
try{
    a.toString();
    b.equals(a);
} catch (NullPointerException E) {
    return null;
}
```

Figure 5.10: Example showing a try catch statement

```
class A{
    int i;
    /*
        public normal_behavior
        ensures i == \old(i) + 1;
    */
    public void inc(){
        i++;
    }
    /*
        public normal_behavior
        ensures i == \old(i) + 1 + 1;
    */
    public void doubleinc(){
        inc();
        inc();
    }
}
```

Figure 5.11: Example showing the contract of a method making a method call

```
/*@
    ensures \exists double d; 0 <= d ; d < 1; \result == d;
@*/
```

Figure 5.12: Manually created contract for the Math.random() method.

```
/*@
    public normal_behavior
    requires a >= 0;
    ensures \result >= 0;

    public normal_behavior
    requires a < 0 || a != a;
    ensure \result != \result;
@*/
```

Figure 5.13: Manually created contract for the Math.sqrt(double a) method. The `a != a` and `\result != \result` is used to denote NaN.

6
Implementation

This section will briefly explain the external dependencies of SpecIT and why they were deemed necessary as well as give a general explanation of the structure and flow of the tool.

6.1 Structure

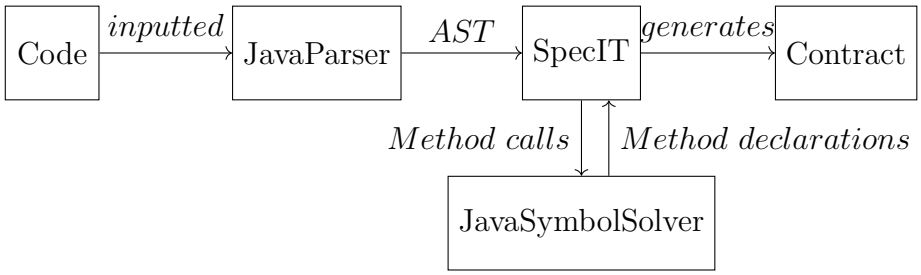

Figure 6.1: Contract generation with SpecIT and JavaParser

Given a file directory, SpecIT searches it to find all Java files and outputs one contract for each method found within those files. In order to do this, it follows a few steps:

Parsing - Given a directory, SpecIT identifies files containing Java source code. These files are then parsed which generates an Abstract Syntax Tree (AST) which in turn is sent to the contract generator.

Extracting - Given an AST, the contract generator finds all nodes representing methods and processes all their child nodes, extracting information and generating the contract as each node is processed.

Writing - Generated contracts are associated with their corresponding methods and are written to a new file identical to the original, with the exception of the added contracts.

Parsing and writing is done mostly with the help of JavaParser, SpecIT does relatively little work here. It is instead in the extracting that most of the work and time has been devoted. Here, SpecIT processes nodes from the AST and creates contracts from them, based on the patterns specified in section 5.

As mentioned before, contracts are generated by processesing nodes from the AST and progressively building a specification for the method that is currently being processed. This is done by first creating a base contract with a single behavior. The contract is then built piece by piece by modifying its behavior, which in turn is

modified based on what nodes are encountered in the AST. In the case that several behaviors are needed in a contract, such as when conditional statements are present, the behaviors are kept in a tree. When branching, the new behavior is created as a child for the current behavior, which ensures that no information is lost and that every behavior is correct. Behaviors themselves contain all the necessary information such as pre- and post-conditions, assigned values and more.

6.2 How SpecIT generates a contract

```
1  public class Example{
2      private boolean error = false;
3
4      public int abs(int b){
5          if (b > 0){
6              return b;
7          } else if (b < 0) {
8              b = b * -1;
9              return b;
10         } else {
11             error = true;
12             throw new IllegalArgumentException();
13         }
14     }
15 }
```

Figure 6.2: Example of code

To give a better idea of how SpecIT works, consider the example given in section 4, repeated in figure 6.2. The various nodes encountered are handled according to the logic presented in section 5. The first node to be processes is the field declaration of `error` on line 2. As it is set as private it is annotated with `spec_public` in order to make it public for specification purposes.
The next step is to generate the contract for the method `abs`. As a conditional statement is encountered, two child behaviors are added to the base behavior, as there are at least two distinct behaviors in the contract due to the conditional statement. A precondition is added to the first of these behaviors based on the condition of the if statement, and then a postcondition is added as there is an immediate return statement.
Then, the next part of the conditional is processed by moving to the other newly created child behavior represented as the top `else` in figure 6.3. A precondition that is the negation of the condition in the if-statement on line 5 is added to it. The next statement is the else-if on line 7 which can be seen as a if-else statement when the initial if has already been considered. Two children are added to the current behavior, as there are two distinct behaviors in the current behavior. The condition of the `else if` is added as a precondition to the first and the negation is added to

the other. The behavior that enters the `else if` is then evaluated and the value assigned to `b` is saved. When `b` is later returned, SpecIT remembers the assigned value and correctly puts this into the contract in the postcondition.
Then, the else statement is processed and as this means that there are no more behaviors, no more child behaviors are created. The assignment to a field is recognized and added as an `assignable` clause, and the throwing of an exception changes the behavior from normal to exceptional, removes all preconditions and created `signals` and `signals_only` clauses. As the end of the method is reached, SpecIT generates the complete contract. This is done by visiting all leaf behaviors in the behavior tree, and adding them as separate behaviors. Figure 6.3 shows how the conditional is interpreted by SpecIT and how the behavior tree is structured after the method has been completely processed. Lastly, the contract and any other annotations are added to the AST and are written to file, resulting in the code seen in figure 6.4.

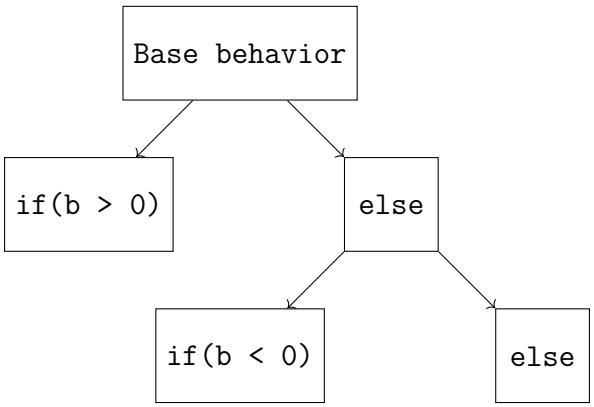

Figure 6.3: Tree diagram showing the structure of behaviors in SpecIT and how conditionals are interpreted

6.3 Dependencies

SpecIT relies on two external tools for parsing code into an AST and extracting information - JavaParser [12] and JavaSymbolSolver [13].

6.3.1 JavaParser

JavaParser is a set of tool to parse, analyze, transform and generate Java code. In SpecIT, JavaParser is used to parse and analyze code, and to insert generated contracts into it. The decision to use JavaParser was made to speed up development and to allow more time put into generating contracts, rather than creating a custom parsing solution.

6.3.2 JavaSymbolSolver

JavaSymbolSolver is a tool built as an extension on JavaParser and it allows for pairing symbols with their declarations. In SpecIT this is used to find the declarations

of methods called in the analyzed code. It is also able to find information about variables, such as their type and declaration. It was decided to use JavaSymbolSolver since SpecIT already had a dependency on JavaParser which made JavaSymbolSolver simple to integrate.

```
public class Example{
    private  /*@ spec_public @*/ boolean error = false;
/*@
    public normal_behavior
    requires b > 0;
    ensures \result == (b);
    assignable \nothing;

    also

    public normal_behavior
    requires !(b > 0);
    requires b < 0;
    ensures \result == (\old(b) * -1);
    assignable \nothing;

    also

    public exceptional_behavior
    requires !(b > 0);
    requires !(b < 0);
    signals_only IllegalArgumentException;
    signals (IllegalArgumentException) (error == true);
    assignable error;
@*/
    public int abs(int b) {
        if (b > 0) {
            return b;
        } else if (b < 0) {
            b = b * -1;
            return b;
        } else {
            error = true;
            throw new IllegalArgumentException();
        }
    }
}
```

Figure 6.4: Full specification of class Example

7
Evaluation of the tool SpecIT

There are three main aspects of contracts that are interesting to evaluate - soundness, usability and readability. Soundness is a measure of the correctness of contracts, usability measures how useful the contracts are when applied in the real world such as using them for testing, and readability measures how easy the contracts are to read and understand. Soundness is objectively measured whereas both readability and usability are subjective. The evaluation of these aspects can be done either manually or automatically, approaches which have both weaknesses and strengths. As mentioned in the scope we will not focus on the usability of contracts, but we will compare the generated contracts to human-written ones which should give some indication of their usability.

7.1 Automatic analysis

There are two main avenues that we consider interesting in regards to automatic analysis, namely gathering statistics about generated contracts and evaluate their usefulness in automated testing. This will give some information about how useful they are to other tools and give a general idea of our tools strengths and weaknesses, but it will only give some small information regarding the contracts readability and usability.

7.1.1 Gathering statistics about generated contract

When generating contracts, it is possible to collect certain statistics about them such as how many were created versus how many were discarded, in how many methods we encountered loops or recursive functions, how many pre- and post-conditions were created, just to mention a few. By then running the tool on many projects in various domains and comparing the gathered statistics, it is possible to identify for what types of projects syntactic extraction seems the strongest, and where it is lacking. Inspecting the results can give ideas of areas of improvement. What causes us to discard contracts? To what extent are loops present, and how do they affect us? What types of methods generate large, hard to read contracts with many conditions and behaviors?

7.1.2 Evaluate usefulness in automated testing

One possible way of evaluation the soundness and usability of contracts is to use them in automated testing. This could be as simple as running the JML Runtime

Assertion Checker, checking that they are correct and thus useful. It could also involve running some automated testing tool such as JMLUnitNG [8] and comparing the coverage of the generated tests to some other method of automated testing, such as Evosuite [14].

7.2 Manual analysis

Performing manual analysis is just what it sounds like - manually looking at and evaluating generated contracts. Doing this in a vacuum will not tell us much, simply looking at generated contracts will not tell us if we successfully extracted the intent of the code. Instead we propose two methods.

7.2.1 Comparing to existing specification

One such method is comparing generated contracts to existing specification. To do this, a project that has some form of specification is required. SpecIT will then be run on this same project and the generated contracts can be compared to what specification was written before.
The advantage of this approach is the fact that we are able to analyse whether syntactic analysis correctly capture the intent of the code. The already existing specification should ideally convey this, and by comparing differences to the generated contracts, weaknesses in the generated ones can be found. This might also give an idea of what properties are appropriate to specify using syntactic analysis and what properties require more work or are infeasible. Additionally, we can find differences in the style of writing the contracts, which is more readable and what kind of construct do they favor.

7.2.2 Comparing to manually written contracts

Instead of looking for projects that already have some formal specification written, it is possible to manually select a set of interesting methods for which to generate contracts. Some developer, preferably with at least basic knowledge of writing specification, can then write their own contracts for these methods which can then be compared to those that were generated. Ideally, this person should have little to no knowledge of the workings of the tool in order to decrease bias.
This methods allows for testing of many different areas of specification and the strength of our tools can be tested in these. However, it is important that methods that showcase not only strength but also weaknesses be picked, as to show all sides of the tool. Compared to the previously mentioned method, this puts the man-made contracts somewhat on the same playing field as out tool. They have to try to figure the intent of the code through looking at it, however they can also analyze it semantically, something our tool cannot.
As mentioned, this method allows for very focused testing of our tool as we can select methods which showcase interesting properties and carefully analyze every contract generated, which is not feasible when generating contracts for an entire project. However, there is a risk of bias, not to mention the risk that some interesting

property to test is missed. Comparing the generated contracts to those written by hand will give a general idea of readability, usability and soundness, and let us identify areas where improvement is required, not to mention other various strengths and weaknesses.

7.2.3 Dealing with uncertainties

Due to the limitations of SpecIT and its dependencies we encounter situations where we cannot be confident in the correctness of the contract generated. These situations occur when we encounter statements or expressions that SpecIT does not support such as `try-catch`-statements or `lambda`-expressions, or the symbolic solver 6.3.2 is not able to find the method declaration to a matching method call.
Since accuracy of contracts is such an important aspect we have chosen to consider these as failing and will thus discuss them seperatly from the other results of SpecIT. The failing behaviors are not necessarily wrong however and will, along with the reasons for failing them, be discussed in chapter 8 Results.
However, not all failings lead to an incorrect contract. In chapter 8 we discuss how frequently failings of SpecIT lead to incorrect contracts in the projects used for the experimental evaluation.

7.3 Evaluated projects

Finding projects using contracts proved more difficult than anticipated. The plan was initially to use the Java projects featured in the study Contracts in Practice [3], however the websites hosting these projects were unreachable, with the exception of RCC which is a race condition checker for Java. However due to the age and lack of documentation we were not able to get the dependencies of RCC to work, which resulted in large parts of the project to be unspecified; thus we had to drop RCC as well.
The lack of already studied projects forced us to find other projects to use for evaluation. We will now present the projects used for evaluation and how we selected them.

7.3.1 Votail

Votail is an implementation of Ireland's voting system [15]. It has been formally specified using the Java Modeling Language and verified using formal methods. Votail was the only project which had contracts and working dependencies that we were able to find. The specification found in Votail is not something we aim to replicate due to its high level of quality and coverage, however it provides a good basis for comparison with state of the art specification.

7.3.2 JUnit 4 & 5

JUnit is a popular testing framework for Java. It is a large project with 30.000 lines of code, and was available in two versions: JUnit 4 and JUnit 5. This was of interest as JUnit 4 was released before Java 8 and JUnit 5 after. This made evaluating the tools performance on the newer features of Java easier as the projects are still similiar in nature.

7.3.3 libGDX

libGDX is a game development framework for Java. It is a large project with 750.000 lines of code in total of which 260.000 is Java. It was considered interesting to test the tool on since it is in a completely different vein of programs from the others used in this study, containing more low level and non-Java code.

8

Experimental Results

We assess how well SpecIT works in two ways: manual comparison of human-written contracts to generated ones and statistical analysis. The comparison allows us to assess if SpecIT produces contracts of a similar quality to those written by a human, which would show that syntactic analysis can have a significant practical usefulness. It can also give some intuition of the limitations and strengths of SpecIT. Since a manual comparison cannot be done on a larger scale we complement it with detailed statistics of SpecIT's output, to evaluate its performance on entire projects. The statistics that we consider are the number of pre- and postconditions and behaviors. We also look at how many of the preconditions are null-checks and the amount of pre- and postconditions per behavior.

8.1 Human-written vs. SpecIT

In order to reduce the amount of bias when comparing SpecIT generated contracts to human-written ones, a peer without any knowledge of the syntactic analysis or the tool in general was asked to write contracts for eight methods. The peer is a student of the Computer Science programme and has taken the course Software engineering using formal methods. The course covers the topics of specification using formal contracts and specifically using JML.

Our comparison found that SpecIT could in a lot of cases produce contracts similar to the ones produced by a human. Our small study found that contracts generated by SpecIT were in general more thorough than the human-written ones. If SpecIT found that no field was assigned it would specify it whereas the human would often omit this information. The human-written contracts in section 8.1.1 also contained a mistake of omission, something which was not present in any SpecIT generated contract. The readability of the generated contracts is good, with no strange or unnecessarily large behaviors.

SpecIT was less likely to handle method calls due to the symbolsolver not being able to find the declaration of the called method; this can be seen in sections 8.1.3 and 8.1.4. This resulted in several behaviors being considered as failures which definitely hurt the detail of the contracts. In the examples featured in this study no actual errors were identified in the failing behaviors which means that all contracts achieved soundness. However as can be seen in section 8.1.4 several contracts became underspecified due to the issues with method calls.

SpecIT actually performed better when compared to the Votail project than when comparing to the contracts written by the peer which can be seen in section 8.1.2.

The peer wrote more detailed and complete contracts whereas those found in the Votail project were very sparsely written. This is a great result as the contracts found in the Votail project are more likely to be the kind of contracts that a developer writes for their code. This is a promising sign for the usability of the contracts generated.
The inability to handle loops definitely hurts SpecIT, which can be clearly seen in sections 8.1.1 and 8.1.2. This is something which the human had no problems handling for these examples, however it is something humans also struggle with since proving termination for loops is very difficult. Since loops is a very common code pattern it is one of the larger issues with syntactic analysis.

8.1.1 Sum

```
public static int sum(int end) {
    int total = 0;
    for (int i = 1; i <= end; i++) {
        total += i;
    }
    return total;
}
```

Figure 8.1: Method summing numbers from 1 to end.

The first comparison is done on a method which sums numbers from `1` to `end`. Its code can be seen in figure 8.1.

```
/*@
public normal_behavior
requires end > 1;
ensures \result = end*(end+1)/2;

public normal_behavior
requires ends <= 0;
ensures \result = 0;
@*/
```

Figure 8.2: Human-written specification of method sum in figure 8.1

We can see in figure 8.2 that the human-written contract is well specified and covers all possible inputs, divided into two separate behaviors, one when the input is greater than 1 and the other when the input is smaller than or equal to 0. The first behavior specifies the result to be equal to the mathematical definition. This definition is nearly complete and correct if not for the fact that there is no behavior for `end == 1` which should be part of the first behavior.
As can be seen in figure 8.3 the generated contract for this example is non-informative due to the inability to process loops. It does not catch the case of not entering the

```
/*@
//Generated
public normal_behavior
assignable \nothing;
@*/
```

Figure 8.3: Generated specification of method sum in figure 8.1

loop either, which is a case that syntactic analysis could catch, but is not supported by SpecIT. However it correctly captures the fact that the method is pure, that is it does not modify the object's state, which is an aspect that is missing from the human-written contract.

8.1.2 Ballot

```
public class Ballot{
    protected int positionInList;
    protected int numberOfPreferences;
    protected int[] preferenceList;

    public Ballot(final int[] preferences) {
        numberOfPreferences = preferences.length;
        positionInList = 0;
        preferenceList = new int[numberOfPreferences];
        for (int i = 0; i < preferences.length; i++) {
            preferenceList[i] = preferences[i];
        }
    }

    public void transfer() {
        if (positionInList < numberOfPreferences) {
            positionInList++;
        }
    }
}
```

Figure 8.4: Part of the class Ballot from Votail

Figure 8.4 shows a small excerpt from the class Ballot in the Votail project. The class represents a ballot and has a list of preferred candidates as well as a position in that list.
Figure 8.5 shows the generated specification of the Ballot class. We can see that the specification of the constructor captures two out of three field initializations, but it cannot capture the initialization of `preferenceList`. This is in part due to the limitations of JML as `preferenceList == new int[numberOfPreferences]` is

```
/*@
public normal_behavior
requires preferences != null;
ensures this.numberOfPreferences == preferences.length;
ensures this.positionInList == 0;
assignable this.numberOfPreferences, this.preferenceList[0 + 1],
 ↪  this.preferenceList, this.positionInList;

@*/
public Ballot(final int[] preferences){
...
}

/*@
public normal_behavior
requires positionInList < numberOfPreferences;
ensures this.positionInList == \old(positionInList) + 1;
assignable this.positionInList;

also

public normal_behavior
requires !(positionInList < numberOfPreferences);
assignable \nothing;
@*/
public void transfer(){
...
}
```

Figure 8.5: Generated specification for the class Ballot in figure 8.4

not valid JML, and we cannot specify anything else about it due to the for-loop. The specification also captures the precondition of `preferences` not being null as it would throw a `NullPointerException`. For the method `transfer()` we see that the specification captures the two separate behaviors with correct specification for both.
The human-written contract for Ballot can be seen in figure 8.6 and it has strong similarities to the generated contract. The only additional information regarding the constructor is the specification of the values in `preferenceList` which was missing from the generated specification. The specification of `transfer()` is the same as the generated specification, with small differences in style. Whereas the human-written specification relates the two behaviors with each other by specifying the change of `positionInList`, the generated specification cannot make this distinction and simply states that nothing is assigned in the second behavior.
The human-written contract by the creators of the Votail project can be seen in figure 8.7. It can be clearly seen that the focus was not in pointing out what the

```
/*@
    requires preferences != null;
    ensures positionInList = 0;
    ensures numberOfPreferences == preferences.length;
    ensures (\forall i; i >= 0 && i < preferences.length ;
 ↪  preferenceList[i] == preferences[i]);
    assignable positionInList, preferenceList, numberOfPreferences;
@*/
public Ballot(final int[] preferences) {
...
}


/*@
    public normal_behavior
    requires positionInList < numberOfPreferences;
    ensures positionInList == \old(positionInList+1);

    public normal_behavior
    requires positionInList >= numberOfPreferences;
    ensures positionInList == \old(positionInList);
*/
public void transfer() {
...
}
```

Figure 8.6: Human-written specification of the class Ballot in figure 8.4

assigned values was, but rather what variables can be assigned. It is more explicit regarding the assignment of the array `preferenceList`, stating that all indices might be assigned. The Votail version of the constructor uses different syntax to capture that the input array has to be non-null.
The Votail specification does not specify that there are two separate behaviors for `transfer()` nor what the how the value `positionInList` might change.

8.1.3 addError

The method addError in figure 8.8 shows the adding of a throwable `error` to the list `errors`. If `error` is an `AssumptionViolatedException`, an `AssertionError` is added instead. The human-written contract in figure 8.9 captures the different cases well, with exceptional behavior if `error` is null and separating the two behaviors depending on whether `error` is an `AssumptionViolatedException` or not. However, JML is limited when it comes to object creation which made adding detail other than an increase in the size of `errors` difficult to specify. A missed aspect in this contract is that the second normal behavior could specify that `error` was the element added to `errors` by adding the postcondition `ensures`

```
/*@ also public normal_behavior
  @          assignable numberOfPreferences, positionInList,
 ↪  preferenceList[*], preferenceList;
  @*/
public Ballot (final /*@ non_null @*/ int[] preferences) {
...
}

/*@ public normal_behavior
@   assignable positionInList;
@*/
public void transfer () {
    ...
}
```

Figure 8.7: Human-written specification from the Votail project of the class Ballot in figure 8.4

`errors.contains(error)`.
The generated contract in figure 8.10 shows one of the more common issues that SpecIT encounters when generating contracts: the inability to resolve method calls. This results in these behaviors being seen as risky and we lose a lot of certainty of its validity. So a conservative take on the generated contract finds only one behavior which is the exceptional one. It is very similar to the human-written one in figure 8.9, only being more explicit and specifying that the behavior can only throw `NullPointerException` and that no fields are modified.
Looking at the failing behaviors reveals that those behaviors are not incorrect, only underspecified. Since SpecIT could not resolve the method call `errors.add(error)` it cannot say anything about the changes the method call made.

8.1.4 assertEquals

The figure 8.11 shows the method `assertEquals` with accompanying methods from JUnit4. The method takes two objects and a message and compares the equality between the objects. If the objects are not equal an error is thrown using several method calls.
As can be seen in figure 8.12 the human captures the different methods and their behaviors very well. The exceptional behavior of `fail` is correctly propagated through `failNotEquals` to `assertEquals`. The generated contracts can be found in figures 8.13 and 8.14. Here we can see that the failure to resolve method calls once again hurts the contract generation. The non-failing behavior for `assertEquals` in figure 8.13 is identical to the first human-written behavior in figure 8.12. We can also see that the first failing behavior is very close to the second human-written behavior but the inability to resolve `expected.equals(actual)` means that SpecIT cannot be certain of the consequences and thus marks the behavior as failing. The behavior also features a redundant null-check which SpecIT generated from the method call.

```
private List<Throwable> errors = new ArrayList<Throwable>();

public void addError(Throwable error) {
    if (error == null) {
        throw new NullPointerException("Error cannot be null");
    }
    if (error instanceof AssumptionViolatedException) {
        AssertionError e = new AssertionError(error.getMessage());
        e.initCause(error);
        errors.add(e);
    } else {
        errors.add(error);
    }
}
```

Figure 8.8: Method addError in class ErrorCollector from JUnit4

SpecIT captures the behaviors of `format` well, only differing slightly in syntax from the human-written behaviors. It mimics the human-written contract for `fail`, but does not manage to resolve the method call in `failNotEquals` and thus cannot produce any contract for it at all.

8.2 Statistics

In this section the various statistics gathered by SpecIT will be discussed. The discussion focuses on three aspects – some general statistics regarding all projects, the statistics of all successful contracts and the statistics of all failed contracts.

8.2.1 General statistics

In table 8.1 some general statistics gathered by SpecIT can be seen. From these we can see that about one-third of all behaviors are classified as successful, except on JUnit 5 where the ratio of failing behaviors is much higher. This is due to a larger amount of uncovered statements being encountered, something that is caused by the fact that JUnit 5 is more recent and uses newer language-features, such as lambdas.

The leading cause of failure is caused by the symbolic solver being unable to solve some symbol, which means that we can not be confident in the soundness and validity of the contract. However, SpecIT's greatest strength is finding null checks and we have high confidence in the soundness of these regardless of whether the contract is seen as failing or not. We can have this high confidence due to null checks usually being on parameters and fields and are not reliant on the results of method calls.

```
/*@
public exceptional_behavior;
    requires error == null;
    signals NullPointerException;

public normal_behavior
    requires error != null;
    requires error instanceof AssumptionViolatedException;
    ensures errors.size() == \old(errors.size()) + 1;

public normal_behavior
    requires error != null;
    requires !(error instanceof AssumptionViolatedException);
    ensures errors.size() == \old(errors.size()) + 1;
*/
public void addError(Throwable error) {
...
}
```

Figure 8.9: Human-written specification of the method addError in figure 8.8

8.2.2 Successful behaviors

Comparing the statistics in table 8.2, we can see that the greatest outlier among the projects is libGDX. It is by far the largest of the projects with almost tenfold more methods processed, and there are greater differences between the size of the contracts that can be extracted. It has the greatest standard deviation in all categories, as well as in the greatest max values. However, the median values are in the same range as the other projects, which shows that the overall performance of SpecIT is comparable between projects.

Median values are low across all projects, in the range between zero and one, which hints at inconsistent performance by SpecIT, or more precisely, many empty or near-empty behaviors are extracted. Causes for this could be that many methods contain loops, something SpecIT and syntactic analysis does not handle, or that the method might be unfit for contract extraction, and thus all that is found is an empty contract.

However, mean values are higher, especially in Votail and libGDX, and in all projects max values are quite high, showing that is possible to extract contracts of a sensible size. The rather high values of standard deviation seem to reinforce the notion given by the median values, namely that the performance of SpecIT is inconsistent. This is especially evident in libGDX. However, in that project the maximum values are extremely high, which might skew the other statistics considerably.

Comparing these results to a study on human-written contracts [3] again reinforces the notion that SpecIT performs inconsistently, especially compared to human-written contracts. In that study, median values are generally higher and standard deviations are lower than the performance of SpecIT. However, maximum values for

```
/*@
//Generated
public exceptional_behavior
requires error == null;
signals_only NullPointerException;
signals (NullPointerException) (true);
assignable \nothing;
also

// Failing behavior : SymbolSolverException: Method call
public normal_behavior
requires e != null;
requires errors != null;
requires !(error == null);
requires error instanceof AssumptionViolatedException;
assignable \nothing;
also

// Failing behavior : SymbolSolverException: Method call
public normal_behavior
requires errors != null;
requires !(error == null);
requires !(error instanceof AssumptionViolatedException);
assignable \nothing;
@*/

public void addError(Throwable error) {
...
}
```

Figure 8.10: Generated specification of method addError in figure 8.8

SpecIT are higher, which could signify that SpecIT is able to find some contracts that humans are not.
The largest contract found in the libGDX project has 423 postconditions and 1128 preconditions and is a large `toString()`-method. It converts keycodes to string values and is a `switch-case` with 256 cases, which leads to a ridiculously large contract. This is a contract that a developer would most likely not write, but something that SpecIT can competently extract.

8.2.3 Failing behaviors

Comparing the statistics in table 8.3 to those in table 8.2 shows that the worst-case performance in failing behaviors is comparable to the performance in successful behaviors, while best-case performance is much higher. However, looking at the maximum values shows that some of the generated contracts are ridiculously large,

General statistics				
Project name	**Votail**	**JUnit 4**	**JUnit 5**	**libGDX**
Methods processed	113	3736	4094	30268
Total behaviors	156	4203	4579	50843
Successful behaviors	64	1477	905	16502
Failing behaviors	92	2726	3674	34341
Symbol solver failures	81	1951	2439	26544
Uncovered statements	4	407	950	3622
Unresolved parameters	7	368	285	4175
Total postconditions	64	856	644	25662
Total preconditions	46	440	337	24092
Successful null checks	3	210	149	4783
Total null checks	103	3166	5583	66074

Table 8.1: General statistics gathered by SpecIT

and are thus likely not readable. Given the size of the contracts, it is also likely that some of these are not sound.
While some of these contracts are most likely unsound, or unwieldy to use at best, we believe that some of these failing behaviors might still be sound. This is because failure to solve a symbol marks a contract as failing, but this does not mean that it is an incorrect contract. In the same sense, a contract with an excessive amount of behaviors is indicated as failing, but it may still be correct, merely with less readability.

```
    static public void assertEquals(String message, Object expected,
↪   Object actual) {
        if (expected == null && actual == null) {
            return;
        }
        if (expected != null && expected.equals(actual)) {
            return;
        }
        failNotEquals(message, expected, actual);
    }

    static public void failNotEquals(String message, Object
↪   expected, Object actual) {
        fail(format(message, expected, actual));
    }

    public static String format(String message, Object expected,
↪   Object actual) {
        String formatted = "";
        if (message != null && message.length() > 0) {
            formatted = message + " ";
        }
        return formatted + "expected:<" + expected + "> but was:<" +
↪   actual + ">";
    }

    static public void fail(String message) {
        if (message == null) {
            throw new AssertionFailedError();
        }
        throw new AssertionFailedError(message);
    }
```

Figure 8.11: The method assertEquals with accompanying methods from JUnit4

```
/*@
   public normal_behavior
       requires expected == null;
       requires actual == null;

   public normal_behavior
       requires expected != null;
       requires expected.equals(actual);

   public exceptional_behavior
       requires expected != null || actual != null;
       requires expected == null || !expected.equals(actual);
       signals AssertionFailedError;
   @*/
   static public void assertEquals(String message, Object expected, Object
↪  actual) {
       ...
   }

   static public void failNotEquals(String message, Object expected, Object
↪  actual) {
       fail(format(message, expected, actual));
   }

   /*@
   public normal_behavior
       requires message != null && message.length()> 0
       ensures \result == message + "expected:<" + expected + "> but was:<" +
↪  actual + ">";

   public normal_behavioe
       requires message == null || message.length == 0;
       ensures \result == "expected:<" + expected + "> but was:<" + actual +
↪  ">";
   @*/
   public static String format(String message, Object expected, Object actual)
↪  {
       ...
   }

   /*@
   public exceptional_behavior
       requires message == null;
       signals AssertionFailedError;

   public exceptional_behavior
       requires message != null;
       signals AssertionFailedError;
   @*/
   static public void fail(String message) {
       ...
   }
```

Figure 8.12: Human-written specification of method assertEquals and accompanying methods in figure 8.11

```
/*@
//Generated
public normal_behavior
requires expected == null && actual == null;
assignable \nothing;
also

// Failing behavior : SymbolSolverException: Method call
public normal_behavior
requires expected != null;
requires !(expected == null && actual == null);
requires expected != null && expected.equals(actual);
assignable \nothing;
also

// Failing behavior : SymbolSolverException: Method call
public normal_behavior
requires expected != null;
requires !(expected == null && actual == null);
requires !(expected != null && expected.equals(actual));
assignable \nothing;
@*/
    public static void assertEquals(String message, Object expected,
 ↪  Object actual) {
        ...
    }
```

Figure 8.13: Generated specification of method assertEquals in figure 8.11

```
/*@
//Generated
// Failing behavior : SymbolSolverException: Method call
public normal_behavior
assignable \nothing;
@*/
    public static void failNotEquals(String message, Object expected, Object
 ↪  actual) {
        fail(format(message, expected, actual));
    }
/*@
//Generated
public normal_behavior
requires message != null;
requires message != null && message.length() > 0;
ensures \result == (message + " " + "expected:<" + expected + "> but was:<" +
 ↪  actual + ">");
assignable \nothing;

also

public normal_behavior
requires message != null;
requires !(message != null && message.length() > 0);
ensures \result == ("" + "expected:<" + expected + "> but was:<" + actual +
 ↪  ">");
assignable \nothing;
@*/
    public static String format(String message, Object expected, Object actual)
 ↪  {
        ...
    }
/*@
//Generated
public exceptional_behavior
requires message == null;
signals_only AssertionFailedError;
signals (AssertionFailedError) (true);
assignable \nothing;

also

public exceptional_behavior
requires !(message == null);
signals_only AssertionFailedError;
signals (AssertionFailedError) (true);
assignable \nothing;
@*/
public static void fail(String message) {
    ...
}
```

Figure 8.14: Generated specification of methods accompanying assertEquals in figure 8.11

Votail0.0.1b [SUCCESSFUL]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	0.885	0	11	2.036
Postconditions per method	0	1.231	1	4	0.783
Null checks per method	0	0.058	0	1	0.235
Behaviors per method	1	1.231	1	3	0.469
Preconditions per behavior	0	0.359	0	11	1.169
Postconditions per behavior	0	1	1	3	0.713
Null checks per behavior	0	0.047	0	1	0.213

junit4-master [SUCCESSFUL]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	0.316	0	12	1.002
Postconditions per method	0	0.614	0	16	0.917
Null checks per method	0	0.151	0	9	0.580
Behaviors per method	1	1.060	1	8	0.337
Preconditions per behavior	0	0.149	0	4	0.429
Postconditions per behavior	0	0.580	0	5	0.711
Null checks per behavior	0	0.142	0	3	0.437

junit5-master [SUCCESSFUL]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	0.389	0	60	2.250
Postconditions per method	0	0.743	1	8	0.752
Null checks per method	0	0.172	0	16	0.805
Behaviors per method	1	1.044	1	8	0.319
Preconditions per behavior	0	0.186	0	9	0.612
Postconditions per behavior	0	0.712	1	5	0.660
Null checks per behavior	0	0.165	0	4	0.520

libgdx-master [SUCCESSFUL]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	2.402	0	1128	21.590
Postconditions per method	0	2.558	1	423	11.748
Null checks per method	0	0.477	0	222	3.922
Behaviors per method	1	1.645	1	423	6.684
Preconditions per behavior	0	0.730	0	16	1.324
Postconditions per behavior	0	1.555	1	55	2.478
Null checks per behavior	0	0.290	0	8	0.713

Table 8.2: Tables showing the successful behaviors generated for Votail, JUnit4, JUnit5 and libGDX

Votail0.0.1b [FAILING]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	3.061	1	77	10.087
Postconditions per method	0	1.697	1	16	2.443
Null checks per method	0	1.515	1	13	2.476
Behaviors per method	1	1.394	1	11	1.38
Preconditions per behavior	0	1.098	0	11	1.931
Postconditions per behavior	0	1.217	1	5	1.299
Null checks per behavior	0	1.087	1	8	1.419

junit4-master [FAILING]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	1.519	1	93	3.499
Postconditions per method	0	0.770	0	24	1.474
Null checks per method	0	1.216	1	61	2.072
Behaviors per method	1	1.122	1	17	0.651
Preconditions per behavior	0	0.677	0	8	1.004
Postconditions per behavior	0	0.686	0	6	0.876
Null checks per behavior	0	1.084	1	8	1.103

junit5-master [FAILING]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	1.956	1	174	4.771
Postconditions per method	0	0.942	1	20	1.587
Null checks per method	0	1.643	1	110	3.081
Behaviors per method	1	1.111	1	16	0.594
Preconditions per behavior	0	0.881	0	22	1.518
Postconditions per behavior	0	0.848	1	20	1.166
Null checks per behavior	0	1.479	1	22	1.795

libgdx-master [FAILING]					
Statistic measured	**Min**	**Mean**	**Median**	**Max**	σ
Preconditions per method	0	4.669	1	7004	53.321
Postconditions per method	0	3.315	1	1952	28.566
Null checks per method	0	2.803	1	5548	39.550
Behaviors per method	1	1.571	1	192	3.034
Preconditions per behavior	0	1.486	1	46	2.534
Postconditions per behavior	0	2.11	1	63	4.289
Null checks per behavior	0	1.785	1	46	3.033

Table 8.3: Tables showing the failing behaviors generated for Votail, JUnit4, JUnit5 and libGDX

9
Conclusions

In this thesis we used automated syntactic analysis to extract formal contracts from code. We developed a tool called SpecIT to do this analysis and ran it on several large projects such as JUnit, libGDX and Votail. The results show that SpecIT could produce contracts of reasonable quality, sometimes even comparable to that of a human. In general a human will provide more expressive contracts for a more varied range of code, although our study found that a human is prone to introducing small mistakes and omissions. The big advantage of SpecIT is the automation, as creating contracts is a time consuming process; the small sample of functions shown in the human-written comparison took the human about an hour to complete whereas SpecIT is able to process about 1000 lines of code per second for larger projects. The generated contracts were also more detailed than those found in actual projects; which might be a better representation of how contracts written by developers look like. In general, SpecIT performs a bit inconsistently. This is due to internal failures or methods unfit for syntactic analysis, such as those containing loops.
Syntactic analysis is a simpler approach than semantic analysis and one should not expect to get a complete specification using syntactic analysis. Since it is relatively simple it requires a lot less work to implement.
Are the contracts produced by syntactic analysis of any real value? Considering their similarities to human-written contracts they are certainly useful to some extent, such as as oracles for testing. The usages of these contracts is something we consider to be of interest and should be explored further.

9.1 Future work

There are several aspects of SpecIT that we know could be improved. We have a lot of issues with resolving method calls which resulted in a lot of failing behaviors. Resolving this would eliminate most of the failing behaviors as well as increase the level of detail greatly. There is also the possibility to add support for special cases of loops to cover cases such as the one seen in section 8.1.2, however this might require more effort than the benefits it entails. Making use of human-written contracts for the Java standard library is also something that could improve the contracts generated, although this does require a lot of time to implement. Adding support for uncovered statements such as lambdas would also improve the level of detail, however due to the difficulties of implementing these statements it might not be a worthy endeavour.
The value of generated contracts is also something that warrants further research.

Another question of interest is whether using generated contracts can help automated testing – for example, by increasing coverage, by reducing the number of false positives or by finding more bugs. Investigating these issues could help discover areas where syntactic analysis is practically useful.

Bibliography

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

[2] Michael D. Ernst, Alberto Goffi, Alessandra Gorla, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, Saarbrücken, Genmany, July 18–20, 2016.

[3] H-Christian Estler, Carlo A Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *International Symposium on Formal Methods*, pages 230–246. Springer, 2014.

[4] Yi Wei, Carlo A Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 191–200. ACM, 2011.

[5] Nadia Polikarpova, Carlo A Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 262–271. IEEE Press, 2013.

[6] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9), 2009.

[7] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML. Technical report, Citeseer, 2001.

[8] Daniel M Zimmerman and Rinkesh Nagmoti. JMLUnit: The next generation. In *International Conference on Formal Verification of Object-Oriented Software*, pages 183–197. Springer, 2010.

[9] Gary T Leavens and Yoonsik Cheon. Design by contract with JML, 2006.

[10] Curtis Clifton Yoonsik Cheon Clyde Ruby David Cok Peter Müller Joseph Kiniry Patrice Chalin Gary T. Leavens, Erik Poll and Daniel M. Zimmerman. *JML Reference Manual*, May 2013.

[11] Java™ platform, standard edition 7 api specification - class math. `https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`. Accessed: July, 2017.

[12] JavaParser. `http://javaparser.org/`. Accessed: July, 2017.

[13] JavaSymbolSolver. `https://github.com/javaparser/javasymbolsolver`. Accessed: July, 2017.

[14] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[15] Dermot Cochran and Joseph R Kiniry. Vótáil: Pr-stv ballot counting software for irish elections. *Formal Verification of Object-Oriented Software*, page 235.