Static Behavioural Subtyping Checking in Java

Luis Mayorga Martínez
University of Antwerp

February 7, 2014

Abstract

Extending classes that are not behavioural subtypes represent a risk in software applications due to the unexpected behaviour that they can present at runtime. Most specification languages force the specification of subtypes to satisfy the notion of behavioural subtypes. However, by forcing specifications to satisfy behavioural subtyping, contract enforcement tools may obfuscate the problem whenever the corresponding implementation is not a behavioural subtype. The intention of this research is to find a way of performing that kind of check in a static manner and without forcing the subtype to inherit the specification of the superclass. With that purpose, it is presented how to analyze the relationship between the contracts of a hierarchy and compare their restrictiveness using a logic solver. A proof of concept developed to do so with a subset of the Java language is also provided and explained.

1 Introduction

The use of verification and validation techniques is a very extended practice in Software Engineering Processes that helps to reduce the faults that are present in it and ensure that software meets its requirements[1]. A certain degree of commitment from the developer is often required, who needs to write this specification in a concrete notation so that the verification tools can later check the code against it.

The specification or description of each module, consisting of preconditions, postconditions and invariants, commonly referred to as contracts. These contracts are attached to each element and, make possible to develop a system in a modular manner. This is what is known as Design by Contract[2].

These contracts are a reliable and simple resource in traditional imperative programming but in Object Oriented (OO) Software Development some problems arise because of the added complexity and the existence of polymorphism. Some characteristics that are intrinsic to the use of this paradigm such as subtype polymorphism affect how classes and members comply with the contracts.

This can be seen in some of the most simple constraints that can be imposed to a function or method, preconditions and postconditions. In a structured program, if the postcondition of a function is not met, the function itself is to blame, and if the precondition is violated, the calling method was the one that failed to establish the proper context [3]. In an OO environment, instead, due to the aforementioned properties, a method of an extending class can attend different constraints than the ones present in its superclass. For this reason, an adequate environment for the method of a expected class may not be so for the overriding one, causing the call to fail at runtime.

The property that all class hierarchies need to have, consisting of the fact that a property provable for a class must also be so for its subclasses is known as Liskov Substitution Principle (LSP) [4].

There are tools available that provide a way of checking code against contracts but most of them face the problem in a way that has some drawbacks. Some tools substitute the contracts for assertions and raise an error if the conditions are not satisfied at runtime [3]. The checking happens only locally, without taking into account the existence of a hierarchy or an interface [5]. Finally, most of the tools check types in a way in which the child members and methods inherit the restrictions of the superclass so that they are always met[3][6]. This makes them unable to check if the LSP is satisfied.

This work tries to explore the possibility of statically checking if the contracts present within a hierarchy comply with the LSP by turning them into logic formulas and comparing their satisfiability. The main difference with current contract checking tools is that classes are checked to be behavioural subtypes of others and not forced to be so, which allows us to detect the problem rather than obfuscating it. If there exists a run that could make the method call fail, it would be found without having to execute it looking for those values.

This paper consists of six sections. Section 2 provides some necessary concepts to understand how design by contract works and how other tools perform their analysis. Section 3 presents the theoretical basis for the contract checking using an Satisfiability Modulo Theories (SMT) Solver. Section 4 explains the work done with the tool and what is capable of doing. Section 5 presents the way in which the tool could be improved to achieve a complete support for Java. Finally, section 6 presents the conclusions.

2 Background

2.1 Contracts

Within design by contract, the specification of a program is defined by means of preconditions, postconditions and invariants. These conditions or predicates model the desired behaviour of the software, something that between modules can work as an agreement.

Mutual obligations between modules empowers coherence between all parts of a project and serve as modular specification techniques. This definition of interfaces between components turns modular development into something feasible.

Contracts can be used in structured programs as well as in OO programming and there exist a wide range of resources for defining these contracts. In some languages these are available as part of the language, while in others can be added through frameworks. In this document only an OO approach to using contracts will be analyzed. The syntax of our specifications is based on the Java Modeling Language (JML)[6], which is a specification language for Java programs.

Some of the basic contracts that can be defined are the following.

@requires Defines a precondition. That is, a condition that must be met prior to the execution of the method.

@ensures The postcondition must be satisfied after the execution of the method.

@invariant The condition is always met within the class.

In Fig. 1 a module of code that makes use of a simple contract can be seen. It presents the `Particle` class, which defines a particle that can be drawn in a visual application. In line number 3 we can see an `invariant` that affects the `speed` variable. As can be understood from the content of the contract, the class member `speed` must always be greater than zero.

Fig. 2 lists the code of the random generator used in the previous class. The `ensures` annotation present in line number 3 specifies the postcondition of the method. In this case it refers to the return value and limits its to be between 0 and 1.

As seen in the previous examples, the method `setRandomSpeed` from the class `Particle` must comply with the condition that dictates the possible values that the variable `speed` can take. For it, it relies on the method `getRandomNumber` of the class `RandomGenerator`. At the moment of developing `Particle`, it was decided to use `RandomGenerator`, which helps the former to satisfy its imposed conditions. As long as the result returned by the random generator is not significantly altered, and this can be noticed through the change of its contract, it could be refactored being sure that the using class will not have any problems because of it.

```
public class Particle {

    @invariant("speed > 0")
    private double speed;

    public Particle{
        this.setRandomSpeed();
    }

    private void setRandomSpeed(){
        RandomGenerator rg = new RandomGenerator();
        this.speed = rg.getRandomNumber();
    }
}
```

Figure 1: Parcicle class.

```
public class RandomGenerator {

    @ensures("0 <= \result < 1")
    public double getRandomNumber(){
        return Math.random();
    }
}
```

Figure 2: Random number generator.

However, if the `ensures` contract changes, this will affect any clients using the `getRandomNumber` method. This is a good example of how contracts can represent behaviour constraints and what can be expected from some software modules.

2.2 Behavioural Subtypes

This section will introduce the concept of behavioural subtype, and the problems associated with behavioural subtyping, which also regard contracts when used.

In OO programming, a subtype could be any class that extends the functionality or inherits the behaviour of the other class. As Liskov and Wing[4] stated, the extending class can be used according to its apparent (inherited) type with the expectation that if the program performs correctly when the actual type of the object is the inherited class will also work correctly if the actual type of it is the one that inherits. The possibility of being able to seamlessly interchange these two types is known as Liskov Substitution Principle (LSP).

However, there is a problem that might arise. A class can syntactically be a subtype of another class, but this does not ensure that it also is a behavioural subtype. This means that the new class, although defined as a subtype could behave differently from the superclass, causing flaws. Since contracts define the behaviour of these classes, it is possible define a correspondence between the behaviour of the superclass being respected and the relationship existing between the overriding contracts [3][6].

The example involving Fig. 1 and Fig. 2 can be extended with the code listed in Fig. 3. It defines a class that extends `RandomGenerator`, used in the previous code fragments. The main difference with it is that this generator returns a number between -1 and $+1$ as opposed to the former that did so in the range $[0, 1)$. If the code is analyzed, it will be easy to point out the problems that might derive from it.

```java
public class RandomRealGenerator extends RandomGenerator {

    @ensures("-1 <= \result < 1")
    public double getRandomNumber(){
        return -1 + Math.random() * 2;
    }
}
```

Figure 3: Random real number generator.

```java
    private void setRandomSpeed(){
        RandomGenerator rg = new RandomRealGenerator();
        this.speed = rg.getRandomNumber();
    }
```

Figure 4: Particle class fragment using random real number generator.

While Fig. 1 and Fig. 2 are syntactically valid code, the behaviour of `RandomRealGenerator.getRandomNumber()` does not necessarily imply the behaviour of `RandomGenerator.getRandomNumber()`.

Consider the code in Fig. 4. In the call to `getRandomNumber()`, the receiver's static type is `RandomGenerator`. This means that the developer expects a random number in [0.1). However, the dynamic type of the variable is actually `RandomRealGenerator`, so it might return a number in the range $[-1, 1)$, something that is not expected.

Thus, classes that break the LSP should be avoided in order to achieve more robust systems.

As mentioned, classes that do not comply with the LSP produce unexpected behaviour, given that the developer should only rely on the specifications of the receiver's static type whenever a method is called. There are several ways of checking whether a class is able to substitute any of its ancestors.

The simplest way of doing it is as part of the testing process. If there exists a combination of values that makes the program work in a undesired way because of the behaviour difference between classes of a same hierarchy, it can be proven to be incorrect. However, finding this out in a latter state of the development cannot be considered as a practical solution.

Contracts, formally defining the behaviour of the software, enable for the mentioned hierarchy checking as well as checking the implementation against the desired operations to be performed. This protocol must involve the developer, who needs to write the specification in a provided syntax or markup language.

The definition of the behavioural interface allows to strictly compare methods and uncover misleading inheritances. To explain how this is possible, it is necessary to introduce the concept of *refinement* as a binary relation on method specifications. Below is the definition given in [6]:

Let $T' \triangleright spec'$ and $T \triangleright spec$ be specifications of an instance method m, such that T' is a subtype of T. Then $spec'$ refines $spec$ with respect to T', written $spec \sqsupseteq^{T'} spec$, if and only if for all calls of m where the receiver's dynamic type is a subtype of T', every correct implementation of $spec'$ satisfies spec.

We can then consider that a class is a behavioural subtype of another when the method specifications of the subclass refine the ones in the superclass and when the invariant of the subclass implies the invariant of the superclass [6]. This can be summed up in the following rules. Making a subclass a Behavioural Subtype of another class means that within it:

- Method preconditions must not be strengthened.
- Method postconditions cannot be weakened.
- Class invariants should be preserved.

3 Forcing behavioural subtyping

3.1 Available checkers

As mentioned in the previous section, there are several ways of testing code compliance with contracts. JML can be used for annotating code, however, because of the way it has been designed it is not possible its use as a behavioural subtype checker.

The reason behind this is that most contract tools or languages, such as JML, Eiffel or Contract4J, treat contracts of overriding classes as an extension of the specification of the ones overriden and, instead of checking if the contract of the inheriting class is compatible with the one of the superclass, it enforces it to be a behavioural subtype [6].

This specification inheritance mechanism will always include the pre and postconditions of the superclass as described in (1) and (2). By applying these rules, the resulting pre and postconditions of each class will always be a behavioural subtype. Assuming two classes, T and T', where T' is a subtype of T, $T.m$ has precondition pre, and $T'.m$ has precondition pre':

$$pre' \lor pre \tag{1}$$

$$(\backslash old(pre') \implies post') \land (\backslash old(pre) \implies post) \tag{2}$$

As can be seen, the preconditions of the methods in T' are disjointed ($pre' \lor pre$) and this condition cannot ever be stronger than pre. If pre, which would be the precondition of the parent class or the static type, holds, it does not matter if pre' does or not. The specifications will always comply with the LSP, although the implementation might not.

Something similar happens with postconditions. Because the preconditions imply the postconditions, as seen in ($\backslash old(pre') \implies post'$) and ($\backslash old(pre) \implies post$), these only have to hold if the corresponding precondition does so. This entails being only able to check preconditions and postconditions corresponding to the same method but not throughout a hierarchy. Let's assume a scenario in which pre and $post$ hold and pre' and $post'$ do not. In that case, formula (2) would still be satisfied due to the implication existing between pre' and $post'$ that makes $post'$ unnecessary to hold if pre' does not hold either.

The case is similar for invariants. The logic formula in (3) represents how invariants are considered in JML. Joining the invariants with a conjunction means that, apart from the locally specified invariant, the class inherited invariant must also hold. This ensures that the restrictions of the superclass are preserved in the one that extends but in any manner provides a way of checking the LSP, it always forces it and takes behavioural subtyping for granted. The invariant of a subclass could be stronger than the one belonging the superclass and the inherited one could still be held. This could cause the call of one of its methods fail if it respects the restriction that the superclass imposes but not the one of the subclass. An example could be an class that reduces the range of values a member can take. The numbers would still be in the range allowed by the parent type but not in the new one defined by the inheritor.

$$invariant \land invariant' \tag{3}$$

In Fig. 5 a class that represents some kind of communication packet used by an application can be seen. The class in Fig. 6 extends the generic packet class without respecting the LSP. The invariant stays unchanged between the different versions, however, the precondition is more restrictive or stronger in the second class because adds another variable to calculate the space and the range is slimmed down.

The size of the content variable (1600) passed to the method satisfies the precondition of the class `GenericPacket`. However, because `NetworkPacket` needs to assign some space to headers, the actual space for the content is reduced and not enough for the data used in the call. Due to how specification inheritance works with preconditions, as seen in (1), it is only necessary that `GenericPacket`'s preconditions holds. Also, because of (2), only `GenericPacket`'s postcondition is checked, and not `NetworkPacket`'s. The code of `NetworkPacket` is written in a way in which, in case of the precondition

```
public class GenericPacket {

    @invariant("SIZE = 1640")
    final int SIZE = 1640;
    char[] content = new char[SIZE];

    @requires("filling.length <= SIZE")
    @ensures("Arrays.equals(content,filling)")
    protected void setContent(char[] filling){
        System.arraycopy(filling, 0, this.content, 0, SIZE);
    }
}
```

Figure 5: Generic Packet class.

```
public class NetworkPacket extends GenericPacket {

    @invariant("SIZE = 1640")
    final int SIZE = 1640;
    final int HEADERS_SIZE = 180;
    char[] content = new char[SIZE];

    @requires("filling.length <= SIZE - HEADERS_SIZE")
    @ensures("Arrays.equals(Arrays.copyOfRange(" +
            "content, HEADERS_SIZE-1, SIZE-1),filling)")
    protected void setContent(char[] filling){
        System.arraycopy(createHeader(), 0, this.content, 0, HEADERS_SIZE);
        System.arraycopy(filling, 0, this.content, HEADERS_SIZE,
                SIZE-HEADERS_SIZE);
    }
}
```

Figure 6: Network Packet class.

not holding, the postcondition would not do either, so the method call would fail but no errors would be thrown. If the tool is able to check the implementation against the specification, it could alert of the former not complying with the inherited specification. In this case, because the method is supposed to accept a content of a concrete size as specified by `GenericPacket`'s precondition but is trimmed to also fit the headers. The programmer would have to guess that it is not a matter of implementation, but there is specification inheritance going on the back.

This means that even if the inheriting class has a more restrictive precondition, it will not be taken into account and so would be the postcondition related to it. In this case, specification inheritance would force `NetworkPacket` to be a behavioural subclass of `GenericPacket` and because of that, tools that only check for subtyping will not alert of breaking the LSP. Moreover, for tools that check implementation against contracts, the preconditions and postconditions of `GenericPacket` will be taken for `NetworkPacket`, alerting of `NetworkPacket` not complying with them and masking the real problem, that is that class breaking the LSP.

Some efforts provide a contract enforcement algorithm that checks at runtime whether each type is a behavioural subtype[3]. This defines the creation of code guards from code and the generation of code that checks behavioural subtyping for all method calls during the program's execution. However, this has the drawback of having to run the software in order to discover it flaws. This work provides a way of also

```java
public class Main {
    public static void main(String[] args){
        NetworkPacket np = new NetworkPacket();
        char[] content = new char[1600] ;
        np.setContent(content);
    }
}
```

Figure 7: Extending class usage.

checking the LSP avoiding specification inheritance; moreover in a static way.

3.2 Comparing contracts

As mentioned before, most specification languages and tools force all specifications to satisfy behavioural subtyping rules, potentially causing the problems in section 3.1. To solve these problems, behavioural subtyping will be checked, rather than forced. As mentioned before, most of the tools or languages have some disadvantages in its design such as specification inheritance that diminish their usefulness for checking compliance with LSP so it is necessary to delve for other methods to achieve this without turning the subclass into a behavioural subtype. The problems addressed in section 3.1 can be removed using the following methods.

The contracts of each class in an application can be compared to check whether one class is a behavioural subtype of another. That is, comparing each precondition and postcondition of one class to another to check if it is equal, stronger or weaker. This allow to independently check each contract instead of using specification inheritance.

The scheme for comparing if a contract is more restrictive than other is as follows: Both have concrete combinations of input values that makes them evaluate to true but depending of how strong they are, the amount of valid values can differ. In other words, the relationship between the set of values that make contracts true can be mapped to a relationship between their strength or restrictiveness. This can be seen more clearly using the next examples.

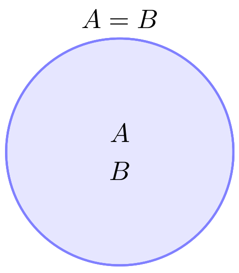

Figure 8: Contracts that evaluate equally.

In figure 8 we can observe the set of values that make two contracts true. A is the set of values that makes the first contract true, while B is the one that corresponds to the second contract. As can be seen, the set of values for each contract are exactly equal so both contracts are equally strong.

Fig. refdia:con2 represents two contracts that are not equivalent and some differences can be seen. This time only some values that make A true also do so for B, so it is clear that B, taking or not other values not contained in A, is more restrictive, because there are some that are not accepted. For example, consider the following two contracts A and B, which could e.g. represent the preconditions of a method $T.m$ and an overriding method $T'.m$:

A: $0 < x < 10$

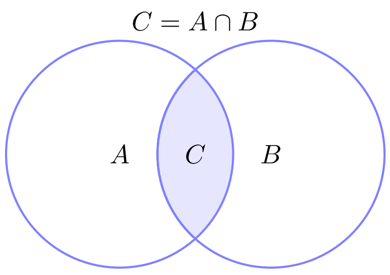

Figure 9: Contracts that differ in some of their accepted values.

B: $5 < x < 15$

If both set of values are attributed to two contracts, *contract* and *contract′* corresponding to two classes, T and T', being the latter a behavioural subtype of the former, as described in the previous sections, it is possible to define each one of the existing sets as follows.

- A is the set corresponding to the values that make *contract* true.
- B is the set of values that makes *contract′* hold.
- C is the subset of values common to A and B, i.e. satisfies both contracts. In this case, $5 < x < 10$.

Through this way of understanding contracts we can conclude that values that belong to the set A but do not so to B are values that prove that both contracts are not equally restrictive. This is represented in Figure 10 and for the previous example would correspond to the range $0 < x < 5$.

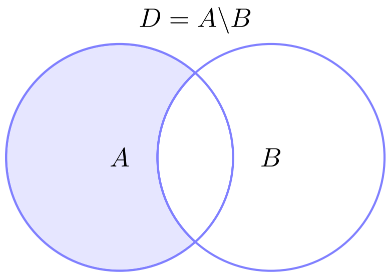

Figure 10: Values that are not respected by the more restrictive contract.

Proving that this set is not empty is enough to know that one contract is more restrictive than other. This problem can be translated into a simple logic formula, (4), and finding it to be satisfiable is a proof of the broken LSP. As stated at the beginning of this subsection, the contract that cannot be more restrictive than its equivalent vary between the defined on the superclass or the one in the subclass depending on its kind. Preconditions and invariants must not be tightened in a subclass, while postconditions must not be weakened.

$$contract \wedge \neg contract' \tag{4}$$

The comparison of contracts could be adapted using the concepts shown. Regarding preconditions, this could be translated into the following formula.

$$pre \wedge \neg pre' \tag{5}$$

As can be deducted from it, satisfiability of (5) means that there is a set of values for the variables used in the contracts in which the precondition holds but the inheriting precondition does not. The existence of this set of values implies that the second precondition, pre', is not a refinement of the first one, pre, and it is stronger, thus breaking the LSP.

The case is similar for postconditions, but in this case they should be checked for not being less restrictive than the ones from the superclass. This can be turned into:

$$\neg post \wedge post' \tag{6}$$

Unlike the method used in JML, postcondition of methods could be checked without taking into account which precondition is held. This would ensure that preconditions and postconditions can be independently proven.

The method for invariants would be similar to the previous ones, as shown in (7).

$$invariant \wedge \neg invariant' \tag{7}$$

4 A Behavioural Subtype Checker for Java

A proof of concept that is available online[1] was developed to test if it was possible to successfully implement the previous concepts. This proof of concept tool statically checks the behavioural subtyping rules of Java classes.

The tool makes possible to retrieve contracts from code and subsequently, process them and feed them to a Satisfiability Modulo Theories (SMT) solver. This solver will check whether the logical formulas referred in section 3.2 are satisfiable and therefore the subclass is not a behavioural subtype and report the results back to the user. Fig. 11 shows an overview of the process.

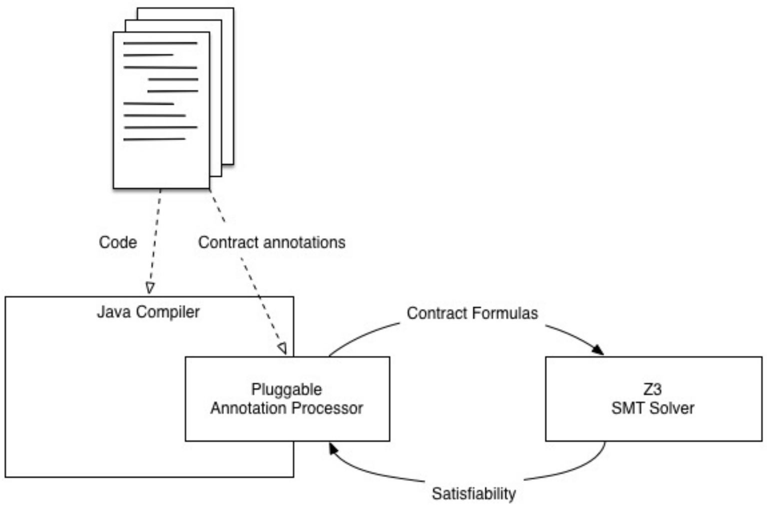

Figure 11: Overview of the process.

Attaching the contracts to the code needed some kind of notation for these to be specified, a task performed by the developer. Due to the Java syntax and philosophy, annotations were chosen as the way of defining contracts. The intention was to replicate how a better known framework such as JML works.

The plan was to work iteratively, adding features as the research progressed. The annotations that have been included in its current state are:

- Requires

[1]http://github.com/rapsioux/BehaviouralSubtyping

- Invariant

The implementation in both is similar because the restraining contract within the hierarchy is the same, as well as most of the code elements where it can be set.

One of the main purposes of the project was to enable this kind of testing statically, which discards the use of reflection, as this is performed at runtime. Taking this into account, I opted to build a tool that is able to process Java annotations statically. I chose to use the JSR 269 API[2],for developing a pluggable annotation processor that is recognized if present during the compiling phase and process the annotations without having to modify the code.

For checking satisfiability of the formulas described in the previous section, (5),(6) and (7), Z3, developed by Microsoft Research[3] was used for being a mature project and having a reasonable community size.

The process for checking the contracts is the following: the annotation processor is called for all the annotations it has been registered for, after that, existence of a superclass is checked and then, the same contract from it is retrieved. After that, both contracts are composed in a logic formula to compare them as stated in section 3.2. The formula is then fed to Z3 and if found to be satisfiable, an error message is printed.

Though it was intended to use Java notation for specifying the contracts within the annotations, that meant a big effort translating Java language to the SMTLib language[4] used by Z3. Finally, it was decided to directly use SMTLib into the annotations. Therefore, only a subset of Java functionality is supported: integer arithmetic, comparisons and boolean operations.

An SMT theory defines the basic predicate and facts regarding a field of application. The implemented theories include the Core Theory except the creation of Bool sort constants and the Ints Theory of SMTLib, respectively defined in sections 4.1.2 and 4.1.3 of the SMTLib tutorial[7]. The first Theory defines most of the logical rules such as equality, boolean constants `true` and `false`, and the operators `not, and, or, xor` and `=>`. Ints Theory adds `+, -, *, mod, div` and `functions`, as well as `<, >, <=, >=` comparison functions. All method parameters and class members are interpreted as belonging to the SMT Integer sort so the tool is only assured to run correctly when provided with this type.

An example can be seen in Fig. 12 and Fig. 13. The first one uses JML, with contracts specified using standard Java notation and the second one corresponds to the same code with annotations already translated to SMTLib, as the tool is able to process.

5 Extension of the work

The work done so far has some limitations but it could be continued and the problems described in the previous section could be suppressed with the implementation of some changes.

Adding a new contract such as 'ensures' would require to alter the formula composition depending on the kind of contract found. Placing the logical negation on the second term for preconditions and invariants and in the first term for postconditions, as shown in (5),(6) and (7) would enable to process also this kind of contract. However, there are keywords and operators that are new in this contract. These keywords are `\return`, that represents the returning value of the method, and `\old`, an operator that permits comparisons of values prior to and after the execution.

To solve that, `\return` can be used as any other variable and `\old` can become a new one through composition with the variable enclosed in it.

Another improvement could be to process other kinds of contracts besides integer arithmetic. Translating Java code into logic formulas is something done by most of the tools that check implementation

[2]https://www.jcp.org/en/jsr/detail?id=269
[3]http://z3.codeplex.com/
[4]http://www.smtlib.org/

```
public class Particle {

    @invariant("speed >= 0")
    private double speed;
    @invariant("x >= 0")
    int x;
    @invariant("y >= 0")
    int y;

    public Particle{
        this.setRandomSpeed();
        this.x = 0;
        this.y = 0;
    }

    private void setRandomSpeed(){
        RandomGenerator rg = new RandomGenerator();
        this.speed = rg.getRandomNumber();
    }

    @requires("x >= 0 && y >= 0")
    private void move(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

Figure 12: Some common annotations in JML's language.

```
public class Particle {

    @invariant("(>= speed 0)")
    private double speed;
    @invariant("(>= x 0)")
    int x;
    @invariant("(>= y 0)")
    int y;

    public Particle{
        this.setRandomSpeed();
        this.x = 0;
        this.y = 0;
    }

    private void setRandomSpeed(){
        RandomGenerator rg = new RandomGenerator();
        this.speed = rg.getRandomNumber();
    }

    @requires("(and (>= x 0) (>= y 0))")
    private void move(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

Figure 13: Annotations as the tool is able to process them.

against contracts[8] and in this case can also be done with the expressions inside annotations. The syntax of boolean operators apart of integer arithmetic, as well as the relationship between classes and their members, can be modelled in new predicates to completely cover Java boolean expressions.

That would allow to logically represent Java objects and method calls within the contracts. However, the use of SMTLib language within the annotations is not very user friendly and another step further could be allowing developers to define the contracts as Java boolean expressions, later translating them with the tool into the logic formulas previously mentioned. These annotations would match the ones used by JML and that would allow using this tool along with others that also use this markup language without having to change anything. This would make possible to verify compliance with the LSP as well as checking the implementation against contracts with the exact same annotations.

The transformation between Java expressions and the SMTLib predicates could be done with a custom parser or using a code transformation library, such as TXL[5].

6 Conclusions

After researching and building a tool as a proof of concept it can be stated that it is possible to check compliance with the Liskov Substitution Principle in a static way. Instead of using specification inheritance, that forces classes to be behavioural subtypes and masks where the actual problem is, it is possible to retrieve the contracts from the source code and compare them with others that belong to different classes and thus check if they break the rules for being a subtype.

These rules include not making preconditions and invariants more difficult to satisfy and having postconditions that guarantee at least the same conditions. Those rules can be modelled as logic formulas that can be proven with a solver and, if the result is not right, the developer can be warned about it.

As future work, using a transformation language and defining predicates to model the entire subset of Java language that can be used within contracts would allow developers to specify the contracts using a programming language they are familiar with.

[5]http://www.txl.ca/index.html

References

[1] D.R. Wallace and R.U. Fujii. Software verification and validation: an overview. *Software, IEEE*, 6(3):10–17, 1989.

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

[3] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.

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

[5] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[6] Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *ICFEM*, pages 2–34, 2006.

[7] David R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial.* GrammaTech, Inc., March 2013.

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