LSINF2335: Programing paradigms: theory, practice and applications

Usage contracts for Ruby

Author:
Noé Phan
Xavier Crochet

Teachers:
Kim Mens
Sebastian Andres Gonzalez
Montesinos

May 16, 2013

Contents

1 Introduction

In this report we will explain the structure of the application designed to support the usage contracts for Ruby. First we present the overview of the applications and the concepts used to implements the usage contracts. The then structure of the DSL is detailed. Finally, a discussion about limitations and improvements that could be done is made.

2 overview

To implement the usage contracts for Ruby we used three concepts of this programming language: meta-programming, reflexivity and usage of gems to parse the code. To have a better understanding of the application's structure, there is the purpose of each files:

- `contractDsl.rb`: contains the DSL definition and methods to analyze structural regularities on the source code.
- `mySexp.rb`: contains en extension of S-Expression definition generated by the parser (`RubyParser`). It defines tools to ease the usage of s-expressions.
- `contracts/*.rb`: contains examples of contracts using the DSL.
- `examples/*.rb`: contains examples of source code to analyse.

To have more detailed informations about the implementation, this overview should be read together with the commented source code in `contractDsl.rb`.

2.1 Meta-programming

To define an internal DSL implementing the usage contracts we used the meta-programming feature of Ruby. More precisely, we used the ability to pass blocks of code to a method and we evaluate them by using `instance_eval()`. Each method implementing a structural condition returns a boolean value such that each evaluated block will return this value to the upper level. At the top level, the method `require` gets this value an inform the user whether the contract is respected or not. Moreover, the benefit of having methods only returning boolean value is that we can combine them using the `and` and `or` keywords of Ruby.

2.2 Reflexivity

The reflexive features of Ruby allowed us to implements structural conditions such as `isImplemented?` and `isOverriden?`. To check whether a specific method is implemented in a specific class we used the reflexive `instance_methods()` function on the desired class.

To check whether a method is overridden or not we used `ancestors()` combined with `instance_methods()` function. Indeed, a function is considered as overridden when it is already defined in one of its ancestors class.

2.3 The parser

Unfortunately, Ruby does not allow the gathering of informations about function bodies. If we only had used the reflexive features or this language to implement the usage contracts, it wouldn't have been as expressive and useful as desired. To overcome this limitation we use a *Gem* named `RubyParser` which allow us to obtain the AST of the classes and methods we analyze.

Using this parser allow us to implement functions such as `calls?`, `returns?`, `assigns?`, `doesSuperSend?` and `doesSelfSend?` which respectively check if the liable method *calls* a specific method, *returns* a specific expression, *assigns* a specific variable or does a *super* or *self* send.

In addition, the functions `beginsWith` and `endsWith` respectively check if the *first* and *last* statement of a method respect a particular condition (passed as a block).

3 DSL specification

```
c1 = ContractDSL.new
c1.define {
    contractName "C1"
    inAClass "ExampleClass"
    definedIn "exampleClass.rb"
    forAMethod :m1

    inCondition {
        isOverriden?
    }

    require {
      beginsWith{doesSuperSend?} and
      (calls? :m2 or calls? :m3)
    }
}
```

4 Improvements

famille de méthodes

toutes les méthodes dans une classes

message pour savoir quelle condition faire foirer le contract autre?..

5 Conclusion

content