Project: From Alloy to C#

Description

This project consists of two parts. For the first part, you will generate C# code annotated with Code Contracts[1] from Alloy models written in a subset of the Alloy language. For the second part, you will generate C# object structures from the instances produced by the Alloy analyzer. These object structures will subsequently be used to test the code generated in the first part.

Note that your implementation must *strictly* follow all guidelines described here to enable its automatic evaluation.

Deadlines and Grading

Initial submission:	Sunday, 6 April 2014, 23:59
Final submission:	Thursday, 17 April 2014, 23:59

These deadlines are *strict*; any submissions received later will not be evaluated.

The purpose of the *initial* submission is to give you feedback on your solution. We will automatically evaluate your submission on a small test suite, which will be made available to you after the deadline. The score of the initial submission will *not* count toward your final grade for the project, but enable the teaching assistant of your exercise session to offer you feedback on your implementation. You will then have time until the *final* deadline to correct and complete your project.

Your *final* submission will be evaluated automatically on a larger test suite, which will include all test cases from the first smaller test suite. The grade for this second submission will be your grade for the project, which is *20%* of your final grade for the course.

[1] `http://research.microsoft.com/en-us/projects/contracts/`

Deliverables

To submit your solution, you must send an email to your teaching assistant with the URL of the Bitbucket[2] Git repository containing your code. Note that the repository should be visible only to you, your team members, and your teaching assistant.

It is *your* responsibility to ensure that the teaching assistant receives the email with the URL before the deadline. Claims of lost emails, emails caught in SPAM filters, etc. will not be considered. You should, therefore, send the email a few days in advance and request a confirmation from the teaching assistant.

For each submission, we will not consider any changes committed after the corresponding deadline.

Guidelines

1. Your implementation for the project must be in Java. Your code should extend the Alloy implementation found at the following Git repository:

   `https://github.com/mariachris/AlloyAnalyzer.git`

   Instructions for building the project can be found in the `README.md` file. After building the project, the `dist` directory contains two `.jar` files. The `alloy4.2.jar` file launches the Alloy user interface, and the `alloy4.2tests.jar` file runs the test suite of your project. Both `.jar` files may be executed with the following command:

   `java -jar` <name of `.jar` file>

   You may find a description of the Alloy API at:

   `http://alloy.mit.edu/alloy/documentation/alloy-api/`

2. A template for your implementation is found under the following directory:

   `edu\mit\csail\sdg\alloy4compiler\generator`

   Note that you must *not* make any changes to this template.

[2] `https://bitbucket.org`

The `CodeGenerator.writeCode` method should generate a single C# file containing the code generated from all signatures, relations, functions, predicates, and expressions found in the input Alloy file. The generated C# code should be annotated with the *strongest* Code Contracts that characterize all types of the input file (that is, the types of all relations defined in signatures and the types of function or predicate parameters and return values). We call this C# file the *code file*.

The `TestGenerator.writeTest` method should generate a single C# file containing a `Main` method. You should assume that the input Alloy file contains at least one assertion for which there exists a `check` command. If the Alloy analyzer finds a counterexample instance for the first checked assertion in the input file, the `Main` method should first create the C# object structures corresponding to this counterexample. The `Main` method should then contain all assertions for which there exists a `check` command in the input file. Note that you should ignore the scopes of `check` commands. We call this C# file the *test file*.

The `CSharpGeneratorTests.java` file implements the way in which your project will be evaluated. You may use this implementation *only* to add more test cases to the test suite of your project. You should also make sure that all of your test cases succeed when run with this implementation, which will be used to evaluate your project. For the evaluation of your project, we will replace this file with a different version of it that runs our own test suite. Note that we will not evaluate the generated source code files, but rather the generated executable files.

You may implement a visitor design pattern[3] in the `Visitor.java` file. In brief, the visitor pattern allows you to define operations to be performed on the elements of an object structure without changing the classes of these elements.

The `tests0.als` file is an example Alloy file. For this Alloy file, your implementation should generate the code in answer files `answer0.cs` and `answer0.tests.cs`, which represent the code and test files, respectively. Note that in the `answer0.cs` file the bodies of methods `Helper.Closure` and `Helper.RClosure` have been removed but your implementation should generate them correctly. You should *strictly* follow *all* naming conventions used in the answer files.

[3]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994

3. You may execute your implementation through the Alloy user interface as follows.

   After opening an Alloy file, you may select Generate C# of the Execute menu to generate the code file. Method `CodeGenerator.writeCode` is called in the following file:

   `edu\mit\csail\sdg\alloy4whole\SimpleGUI.java`

   When clicking on the Execute button, the test file is generated. Method `TestGenerator.writeTest` is called in the following file:

   `edu\mit\csail\sdg\alloy4whole\SimpleReporter.java`

4. Your implementation should *only* support relations of arity less than or equal to three when these are defined in a signature, and relations of arity less than or equal to two when these are passed to or returned from a function or predicate. The C# types generated for these relations should be defined as in files `answer0.cs` and `answer0.tests.cs`.

5. Your implementation need *not* support the following operators:

   - for `ExprBinary`: `ISSEQ_ARROW_LONE`, `JOIN`, `DOMAIN`, `RANGE`, `PLUSPLUS`, `SHL`, `SHA`, `SHR`;
     (Note that you *should* handle the `JOIN` operator *only* when its right operand is a relation defined in a signature.)
   - for `ExprConstant`: `IDEN`, `NEXT`, `EMPTYNESS`;
   - for `ExprList`: `DISJOINT`, `TOTALORDER`;
   - for `ExprUnary`: `EXACTLYOF`, `NO`, `SOME`, `LONE`, `ONE`.

6. Your implementation for the generation of the code file should:

   - generate Code Contracts specifying the strongest properties of all supported types using *only* object invariants, pre-, and postconditions;
   - make all classes, fields, and methods public;
   - support abstract signatures and inheritance (with the `extends` keyword);
     (Note that you are not required to generate any Code Contracts within abstract classes.)

- define all methods in a public, static class `FuncClass` as in files `answer0.cs` and `answer0.tests.cs`;
- define two custom methods for computing the transitive and reflexive transitive closures in a public, static class `Helper` as in files `answer0.cs` and `answer0.tests.cs`;
- *not* support multiplicities of signatures *except* for multiplicity `one`, which should be implemented using the singleton design pattern[4] as in files `answer0.cs` and `answer0.tests.cs`;

  (In brief, the singleton pattern uses static state to ensure that there exists only a single instance of a class.)
- *not* support quantification expressions (`ExprQt`).

7. Your implementation for the generation of the test file should support quantification expressions (`ExprQt`) *except* for operators `SUM` and `COMPREHENSION`.

Questions

We encourage you to ask any general questions you might have through the mailing list of the course (`sae2014@sympa.ethz.ch`). For more specific questions, please contact the teaching assistant of your exercise session.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994