GCD Static Checking Example

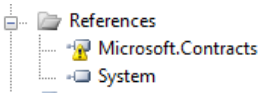
Abstract

This example shows how to use contracts to prove some arithmetic properties of the greatest common denominator computation.

1 Adding the Contract Library Reference

1.1 Visual Studio 2008

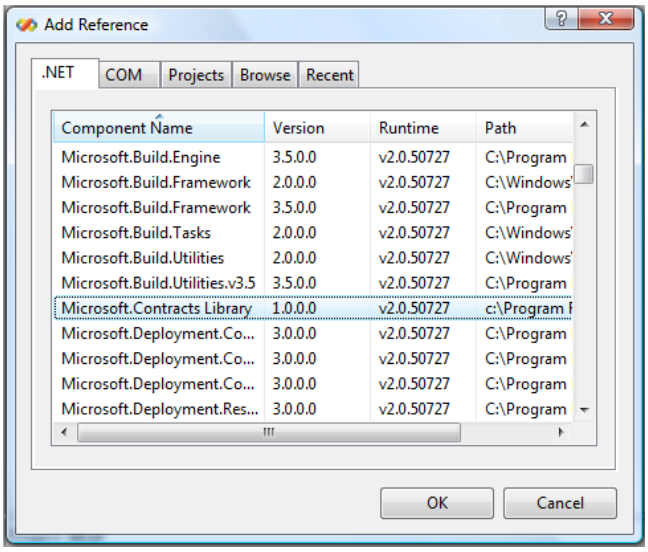
Before getting started with the sample, we need to make sure we have the proper reference for Microsoft.Contracts.dll. Look at the references of project BinarySearch:



If Visual Studio cannot find the Microsoft.Contracts.dll (indicated by the yellow warning), delete the reference and follow the steps below. Otherwise, you are ready to go to Section 2.

For new projects to use contracts, we need to add a reference to the Microsoft.Contracts library to the project.

Assuming you have installed the code contract tools, open the BinarySearch solution and right-click on References in the BinarySearch project and select Add Reference. Find the Microsoft.Contracts library in the .NET tab as shown below and click OK.



1.2 Visual Studio 2010

If you want to try the sample with Visual Studio 2010 you have two choices:

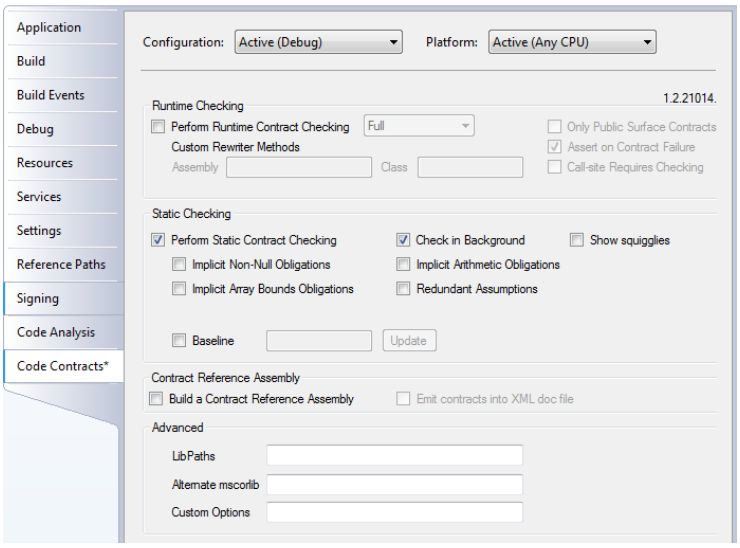
- Target the v3.5 framework (the sample default). In this case, follow the instructions under Visual Studio 2008 as this mode will require the Microsoft.Contracts library to be referenced.
- Target the v4.0 framework. In this case, the contract class is defined in mscorlib and *no* reference to Microsoft.Contracts.dll is necessary.

Thus, to target v4.0 you need to perform the following steps:

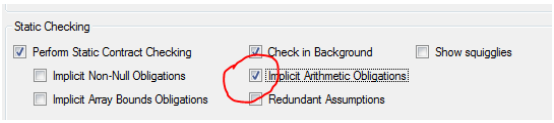
- Delete the Microsoft.Contracts.dll reference from all projects
- Change the target framework of all projects to v4.0.

2 Sample Walkthrough

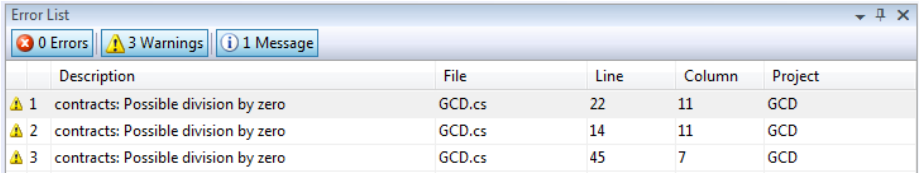
After adding the proper reference, go to the Properties of project GCD, select the Code Contracts pane (at the bottom), and enable static checking by clicking on the checkbox as shown in this screenshot:



Then build the example. There should be no warnings or errors at this point. To get static checking of arithmetic properties, such as division by zero, we need to enable that explicitly by checking the "Implicit Arithmetic Obligations" checkbox as shown in the following screen shot:



Go ahead and add this option, then build again. The warning list should now display three warnings about possible divisions by zero:



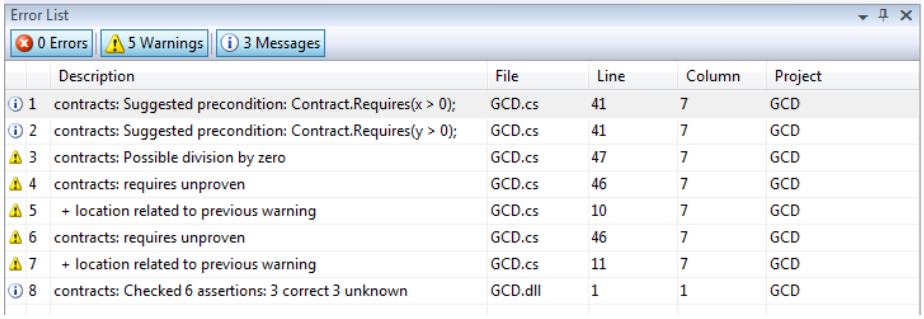
Let's try to write some contracts to make sure we won't run into these division by zero problems. Double click on the first warning. To avoid the division by zero of the code `x %= y`, we can add the following precondition to method `GCD`

```
Contract.Requires(y > 0);
```

The second warning is about the similar division by `x`, so we add a similar precondition for it:

```
Contract.Requires(x > 0);
```

Add those two preconditions and build again. You should now get the following warnings:



	Description	File	Line	Column	Project
1	contracts: Suggested precondition: Contract.Requires(x > 0);	GCD.cs	41	7	GCD
2	contracts: Suggested precondition: Contract.Requires(y > 0);	GCD.cs	41	7	GCD
3	contracts: Possible division by zero	GCD.cs	47	7	GCD
4	contracts: requires unproven	GCD.cs	46	7	GCD
5	+ location related to previous warning	GCD.cs	10	7	GCD
6	contracts: requires unproven	GCD.cs	46	7	GCD
7	+ location related to previous warning	GCD.cs	11	7	GCD
8	contracts: Checked 6 assertions: 3 correct 3 unknown	GCD.dll	1	1	GCD

The possible division by zero remaining is in the NormalizedRational method, when dividing by the gcd value. The GCD should never be zero, and in fact due to our preconditions on the GCD method, our GCD will always be positive. So let's write a postcondition on GCD that makes this explicit. The contracts on GCD should now look as follows:

```
public static int GCD(int x, int y)
{
  Contract.Requires(x > 0);
  Contract.Requires(y > 0);
  Contract.Ensures(Contract.Result<int>() > 0);
```

Write the **Ensures** and rebuild. Now we are left with two warnings that the preconditions of the call in NormalizedRational to GCD are not satisfied. We can easily push these requirements onto the callers of this method by making them explicit preconditions of the normalized NormalizedRational method. In fact, the checker suggests this automatically, as you can see in the informational messages in the Error List. We can weaken the condition on x, as the code explicitly handles the case when x is zero. So we add the following:

```
static public Rational NormalizedRational(bool pos, int x, int y)
{
  Contract.Requires(x >= 0);
  Contract.Requires(y > 0);
```

If we build again now, the checker issues no more warnings.