Rational Runtime Checking Example

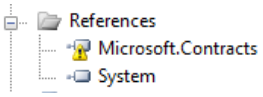
Abstract

This example shows how to enable different levels of runtime checking in your projects.

1 Adding the Contract Library Reference

1.1 Visual Studio 2008

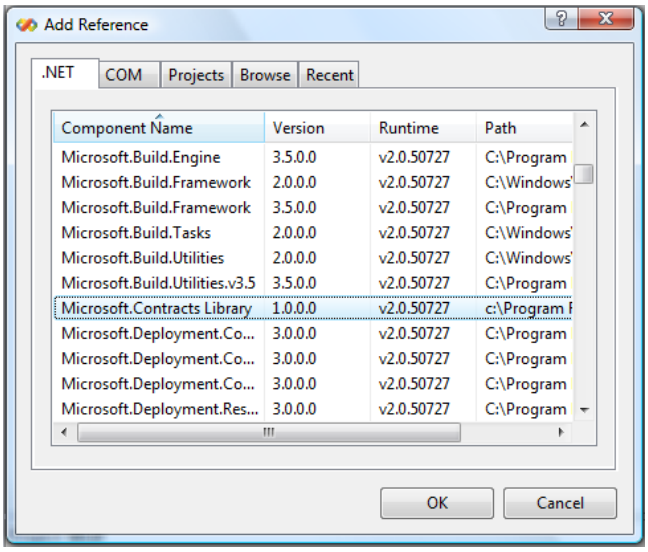
Before getting started with the sample, we need to make sure we have the proper reference for Microsoft.Contracts.dll. Look at the references of project Rational:



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 Rational solution and right-click on References in the Rational 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.

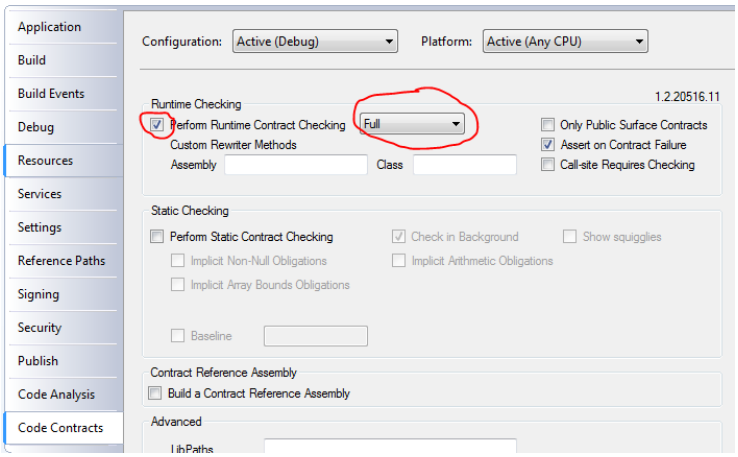
Thus, to target v4.0 you need to perform the following steps:

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

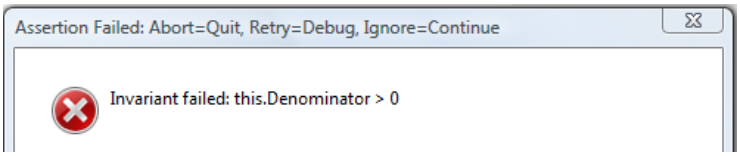
2 Sample Walkthrough

2.1 Full Contract Checking

After adding the proper reference, go to the Properties of project Rational, select the Code Contracts pane (at the bottom), and enable runtime checking. Set the runtime checking level to Full. This will check all contracts at runtime, including preconditions, postconditions, invariants, asserts, and assumes.



Build and run the solution (e.g. by simply hitting F5). You should get the following dialog:



stating that some invariant is violated. To debug this case, click Retry and look at the call stack. We are in the PositiveDenominatorInvariant method, which states that PositiveDenominatorRational expects to always have a positive denominator. Obviously, we somehow failed to maintain this invariant.

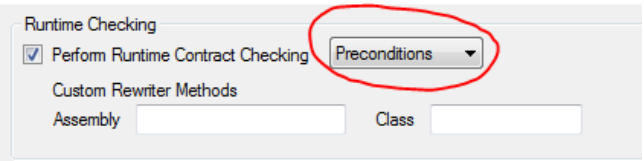
If you look at the call frame below, you notice that we are in the Divide method after the call to **base**.Divide. Since the divisor argument is negative (-5) and the base class simply multiplies the denominator with that value, we ended up with a negative denominator. So the Divide override must reestablish the invariant before returning.

We could fix the code at this point, but that is not the purpose of the sample. So let's stop the debugging session for now, but observe that the bottom of the call stack is located in the Main method at line 18.

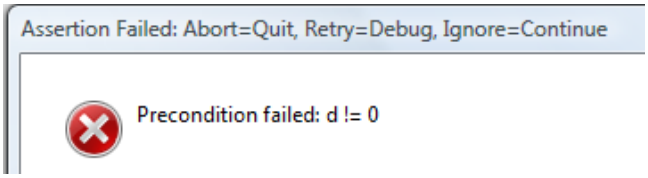
What happened under the hood? The build instructed the compile of the project to include contracts. A post-build step then rewrote the project assembly to perform the checks in the proper positions (e.g., invariants on exit of public methods, post-conditions at method exits), as well as inheritance of all the contracts. Additionally, the rewrite step also introduces strings of the contract conditions so that they are available at runtime when a failure occurs.

2.2 Checking Only Preconditions

Go back to the Code Contracts property pane of the project and change the checking level to Preconditions by using the drop-down menu.



Now let's run the example again (hit F5). This time, we get a different failure:



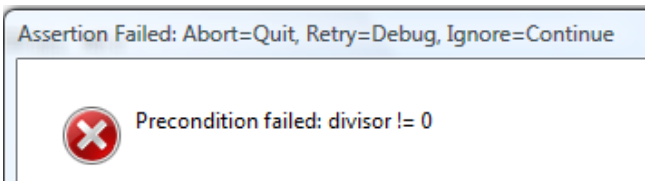
click Retry to enter the debugger and see where we are failing a precondition. We notice that we are in the constructor of PositiveDenominatorRational and parameter d is 0. Looking at the call stack below, we see that we are now at line 20 in the Main method. Thus, by checking only preconditions, we skipped over the invariant check that failed when checking Full contracts.

What happened under the hood? The build instructed the compile of the project to include contract calls. The assembly was then rewritten to retain only precondition checks (including inherited ones) and remove all other contract checks. Additionally, the proper condition strings were inserted so that they are available at runtime.

2.3 Checking Only ReleaseRequires

Stop the debugging session and go back to the Code Contracts property pane and select ReleaseRequires as the checking level in the drop-down menu. This level includes only preconditions of the form **Requires**<E> and legacy-preconditions of the form **if**−then−**throw**[1].

Rebuild and run by hitting F5 again. This time, we hit yet another problem.



Click Retry to enter the debugger and notice that we are now in the Divide method of the NormalizedRational class (look at the call stack). The source line the debugger is stopped on however is the base class Divide method, which specifies the precondition divisor != 0. There are several things to note here:

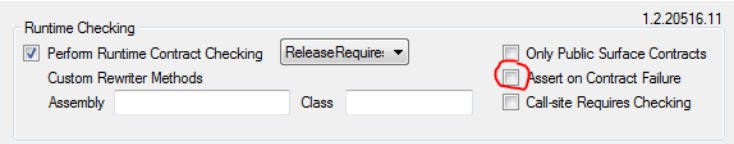
[1]Due to the older contract class in VS2010 Beta1, we also temporarily include RequiresAlways

- The precondition divisor != 0 was inherited from the base method Rational.Divide into the overriding method NormalizedRational.Divide
- The precondition was specified as an **if**−then−**throw**. Due to the presence of the call Contract.EndContractBlock, this "legacy-requires" was recognized by the tools as a proper precondition.
- Because the default behavior for contract failure in this build is set to *assert* (checkbox on Contract Property Pane), even this legacy-requires (if-then-throw) produces an assertion. This is useful to detect contract failures in tests that would otherwise be hidden by catch handlers.

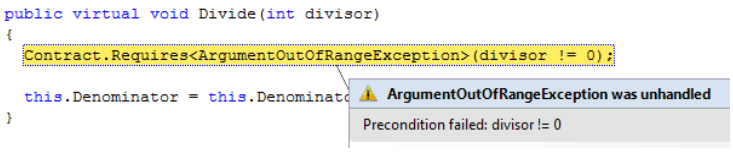
What happened under the hood? The build instructed the compile of the project to include contract calls. The assembly was then rewritten to retain only **Requires**<E>, and legacy-requires, while removing all other contract checks. The rewrite also performed contract inheritance of these preconditions and inserted the proper condition strings so that they are available at runtime. Because the selected runtime failure behavior of contracts was set to *assert*, the instrumented failure behavior was to call System.Debug.Assert instead of throwing exceptions.

2.4 Setting Contract Failure to Throw

Quit the debugger and go back to the Code Contracts property pane again. This time, change the runtime contract failure behavior to throw exceptions by clearing the box "Assert on Contract Failure":



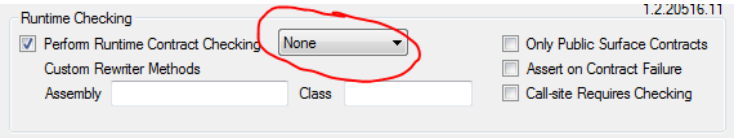
Rebuild and run the project by hitting F5 again. This time, you should see a failure of the following form:



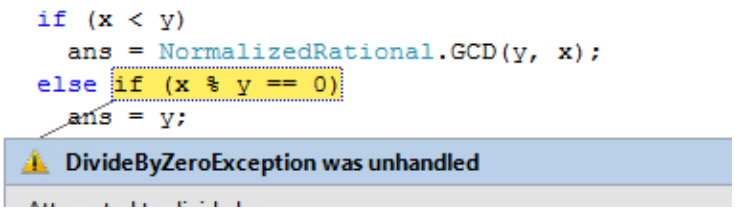
The expected exception is thrown to indicate the contract failure. Because this is a legacy-requires, the failing condition string is not included in the exception message. Other contract failures do throw ContractException and include the failing condition string at runtime.

3 Setting Checking Level to None

Setting the runtime checking level to *None* in the contract pane builds a version of your code where all contract checks are removed, *including* legacy-requires.



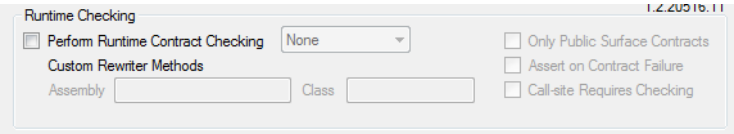
Try it out by setting the level to *None* and hitting F5. In this build and run, there were not enough argument validations, and we fail a division by zero somewhere deep in the code.



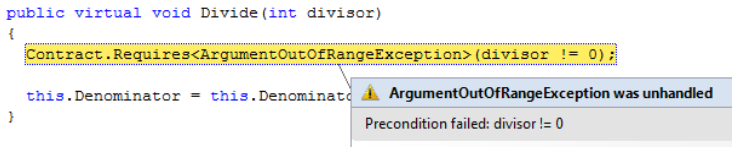
Runtime checking level *None* is therefore not recommended for anything but determining performance of your application without any contracts enabled.

4 Disabling Contract Runtime Checking

Quit the debugger and go back to the Code Contracts property pane again. This time, turn off the runtime contract checking completely:



Rebuild and run the project by hitting F5 again. This time, you should now see the legacy-requires failing as earlier.



A small difference with respect to the earlier version where we enabled runtime checking with level *ReleaseRequires* is that in this build without runtime contract checking turned on, no contract inheritance is performed. Thus, the check fails slightly later, namely when NormalizedRational.Divide calls **base**.Divide. If the

overriding method were not calling the base method at all, the check would be lost in this build.

For release builds where runtime contract checking is disabled, the programmer is responsible for inheriting the argument validations (as in current practice). See Section 5 of the Code Contracts User Manual for more information about how to use Code Contracts effectively in your project.

What happened under the hood? With runtime checking disabled, the build instructed the compile of the project to include no contract calls whatsoever. As a result, only legacy-requires are included in this build.

4.1 Summary

Different levels of runtime checking are available to accomodate particular testing or shipping needs. The more checking that is enabled, the earlier failures are detected.