BinarySearch Static Checking Example

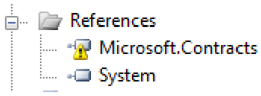
Abstract

This example shows checking of implicit non-null requirements as well as array indexing requirements. The example consists of a binary search routine and a small client of the search that increments a value inside an array. To find the proper place to increment, it searches for the value first.

1 Adding the Contract Library Reference

1.1 Visual Studio 2008

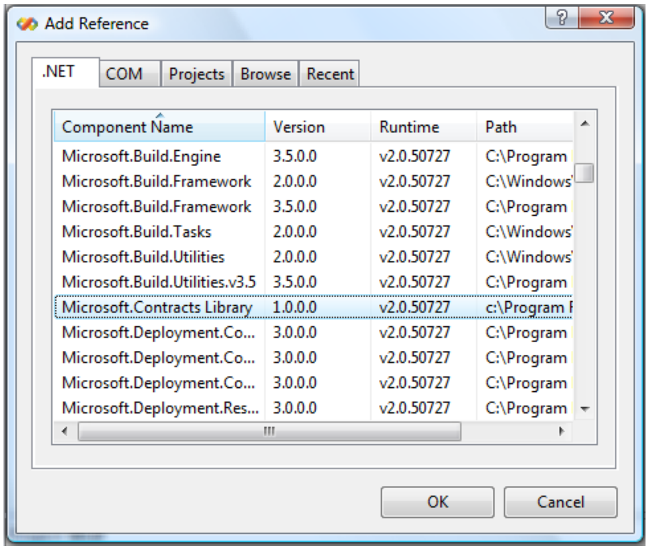
Before getting started with the sample, we need to make sure we have the proper reference for Microsoft.Contracts.dll. Look at the references of project BinarySearch:



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.



Component Name	Version	Runtime	Path
Microsoft.Build.Engine	3.5.0.0	v2.0.50727	C:\Program
Microsoft.Build.Framework	2.0.0.0	v2.0.50727	C:\Windows
Microsoft.Build.Framework	3.5.0.0	v2.0.50727	C:\Program
Microsoft.Build.Tasks	2.0.0.0	v2.0.50727	C:\Windows
Microsoft.Build.Utilities	2.0.0.0	v2.0.50727	C:\Windows
Microsoft.Build.Utilities.v3.5	3.5.0.0	v2.0.50727	C:\Program
Microsoft.Contracts Library	1.0.0.0	v2.0.50727	c:\Program
Microsoft.Deployment.Co...	3.0.0.0	v2.0.50727	C:\Program
Microsoft.Deployment.Co...	3.0.0.0	v2.0.50727	C:\Program
Microsoft.Deployment.Co...	3.0.0.0	v2.0.50727	C:\Program
Microsoft.Deployment.Res...	3.0.0.0	v2.0.50727	C:\Program

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.

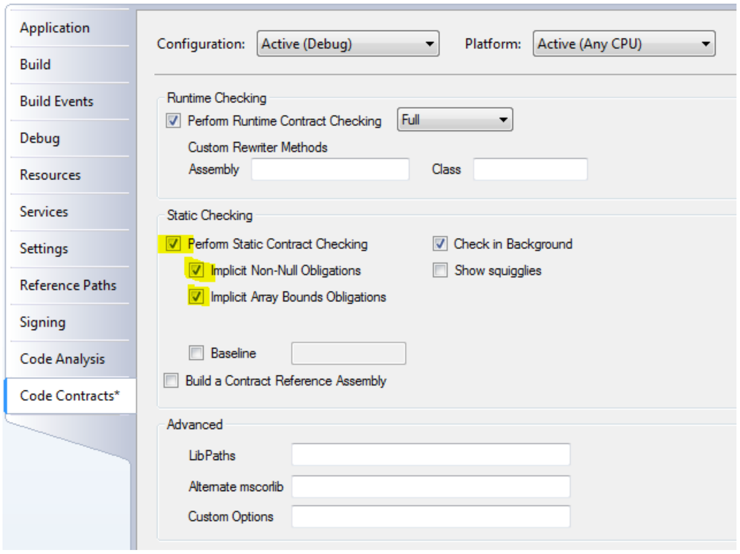
Thus, to target v4.0 you need to perform the following steps:

- 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 BinarySearch, select the Code Contracts pane (at the bottom), and enable static checking by clicking on the static checking box. Also enable implicit non-null and array checks, as shown in this screenshot:



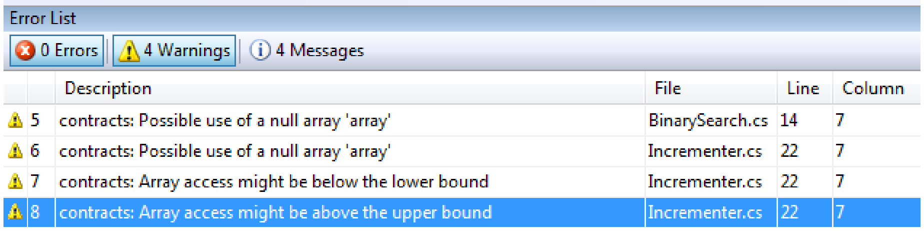
Configuration: Active (Debug)
Platform: Active (Any CPU)

Runtime Checking
☑ Perform Runtime Contract Checking: Full
Custom Rewriter Methods
Assembly:
Class:

Static Checking
☑ Perform Static Contract Checking
☑ Check in Background
☑ Implicit Non-Null Obligations
☐ Show squigglies
☑ Implicit Array Bounds Obligations
☐ Baseline
☐ Build a Contract Reference Assembly

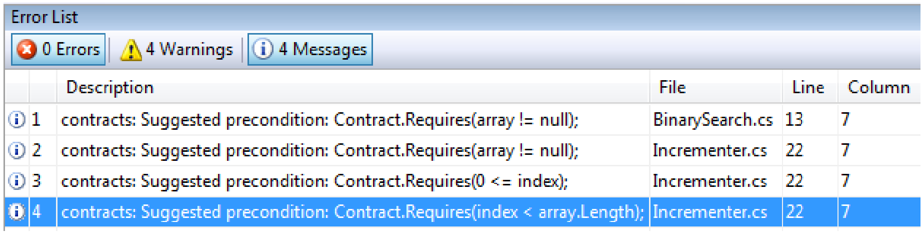
Advanced
LibPaths:
Alternate mscorlib:
Custom Options:

Then build the example. The build should succeed. After a moment[1], the static checker should warn about the following problems:



		Description	File	Line	Column
⚠	5	contracts: Possible use of a null array 'array'	BinarySearch.cs	14	7
⚠	6	contracts: Possible use of a null array 'array'	Incrementer.cs	22	7
⚠	7	contracts: Array access might be below the lower bound	Incrementer.cs	22	7
⚠	8	contracts: Array access might be above the upper bound	Incrementer.cs	22	7

The last three errors are all reported in the IncrementIndex method and have to do with the fact that the method assumes that the array passed as a parameter is non-null. Similarly, it assumes that the index is within the array. The static checker tells us that we should make these assumptions explicit as contracts. If you switch to the Messages tab in Visual Studio, you see that the checker actually suggests the proper preconditions:



		Description	File	Line	Column
ⓘ	1	contracts: Suggested precondition: Contract.Requires(array != null);	BinarySearch.cs	13	7
ⓘ	2	contracts: Suggested precondition: Contract.Requires(array != null);	Incrementer.cs	22	7
ⓘ	3	contracts: Suggested precondition: Contract.Requires(0 <= index);	Incrementer.cs	22	7
ⓘ	4	contracts: Suggested precondition: Contract.Requires(index < array.Length);	Incrementer.cs	22	7

Add those preconditions to the IncrementIndex method (remember to use the cr TAB TAB abbreviation in C# to insert Requires):

---

[1] The static checker runs in the background after the regular build.

```
Contract.Requires(array != null);
Contract.Requires(0 <= index);
Contract.Requires(index < array.Length);
```

If we build again, we get a number of new warnings. Let's fix the one in `BinarySearch` on line 14 first, as it is similar to the ones we just did. The `BinarySearch` method assumes that the passed `array` is non-null and we make that explicit using a precondition:

```
public static int BinarySearch(int[] array, int value)
{
  Contract.Requires(array != null);
```

After rebuilding again, the remaining warnings are all in method `IncrementValue`. Here, the checker cannot ascertain the preconditions of our calls to `BinarySearch` and `IncrementIndex`. Note that it does not warn about the first precondition `array != null` in `IncrementIndex` again, as the checker determines that given the fact that it should be non-null in the call to `BinarySearch`, it would still be non-null in the call to `IncrementIndex`.

The nullness error is again easy to fix by simply adding

```
Contract.Requires(array != null);
```

Because this is such a common precondition, there is a special C# abbreviation for it: `crn TAB TAB`. To get rid of the remaining errors, we have to make it explicit that the `BinarySearch` method actually returns an index into the searched array, so let's add that as a postcondition to `BinarySearch`.

```
Contract.Ensures(Contract.Result<int>() >= 0);
Contract.Ensures(Contract.Result<int>() < array.Length);
```

Note that you can use the C# abbreviation `ce TAB TAB` to get a postcondition, and within that postcondition, use `crr TAB TAB` to get the expression to refer to the method result.

Rebuild the project to see if we got everything right. Lo and behold, there is still an error. The checker should point you at the `return −1` in the `BinarySearch` body. Clearly, our postcondition is too strong in that the method does not always return an index into the array, namely when the value is not found. In that case, it returns -1. So let's weaken the postcondition to the following:

```
public static int BinarySearch(int[] array, int value)
{
  Contract.Requires(array != null);
  Contract.Ensures(Contract.Result<int>() >= −1);
  Contract.Ensures(Contract.Result<int>() < array.Length);
```

and build again. Surely now we must be done ;-) But no, the checker finds yet another problem. It complains that the call to `IncrementIndex` in the `IncrementValue` method does not satisfy the precondition `index >= 0`. Obviously, the `IncrementValue` method is not handling the case where the element is not present in the array. We fix the code by adding a test after the BinarySearch call:

```
public static void IncrementValue(int[] array, int val)
{
  Contract.Requires(array != null);

  int i = Search.BinarySearch(array, val);

  if (i == −1)
  {
    // not found
    return;
  }

  IncrementIndex(array, i);
}
```

Finally, a rebuild will confirm that we now handled the corner cases. One final thing to point out is that the checker figured out the safety of the array indexing operations within the BinarySearch automatically, by inferring the necessary loop invariant.