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.

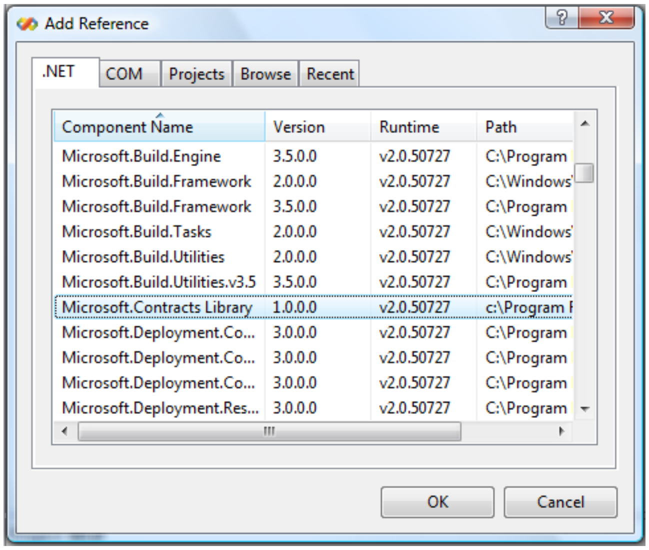
1 Adding the Contract Library Reference

If you are using Visual Studio 2008, or if you for some reason want to target a pre-v4 .NET runtime, then you need to:

- Change the target framework of the project.
- Manually add a reference to Microsoft.Contracts.dll

Otherwise, you may skip this section and go directly the next section!

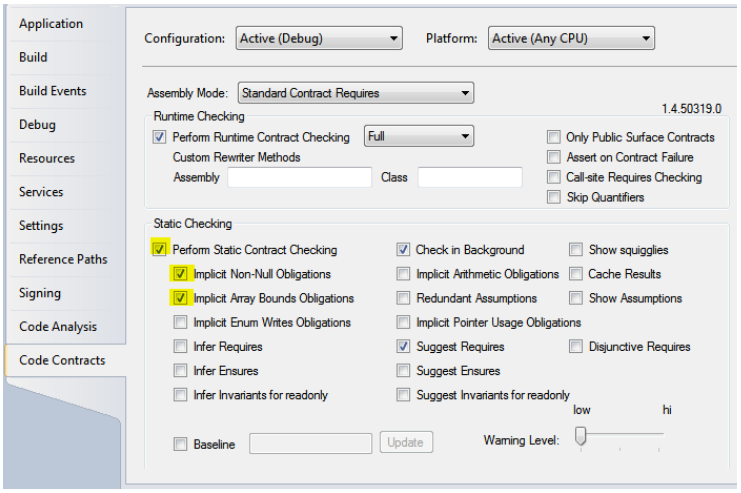
To add the reference, open the BinarySearch solution and right-click on References in the BinarySearch project and select Add Reference. Find the Microsoft.Contracts library in the .NET tab as shown below and click OK.



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

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:

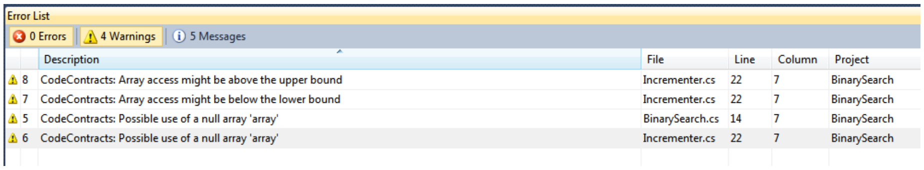


Configuration: Active (Debug)
Platform: Active (Any CPU)
Assembly Mode: Standard Contract Requires
1.4.50319.0

Runtime Checking
☑ Perform Runtime Contract Checking: Full
Custom Rewriter Methods
Assembly
Class
☐ Only Public Surface Contracts
☐ Assert on Contract Failure
☐ Call-site Requires Checking
☐ Skip Quantifiers

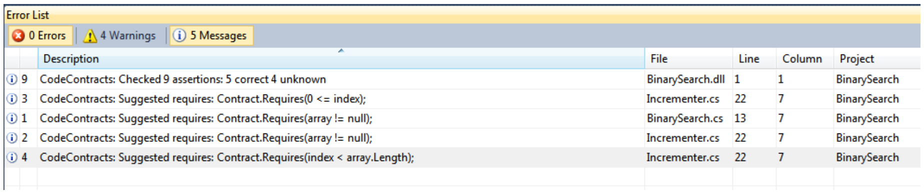
Static Checking
☑ Perform Static Contract Checking
☑ Implicit Non-Null Obligations
☑ Implicit Array Bounds Obligations
☐ Implicit Enum Writes Obligations
☐ Infer Requires
☐ Infer Ensures
☐ Infer Invariants for readonly
☑ Check in Background
☐ Implicit Arithmetic Obligations
☐ Redundant Assumptions
☐ Implicit Pointer Usage Obligations
☑ Suggest Requires
☐ Suggest Ensures
☐ Suggest Invariants for readonly
☐ Show squigglies
☐ Cache Results
☐ Show Assumptions
☐ Disjunctive Requires
☐ Baseline
Warning Level: low

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

The first, second, and fourth 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	Project
ⓘ	9	CodeContracts: Checked 9 assertions: 5 correct 4 unknown	BinarySearch.dll	1	1	BinarySearch
ⓘ	3	CodeContracts: Suggested requires: Contract.Requires(0 <= index);	Incrementer.cs	22	7	BinarySearch
ⓘ	1	CodeContracts: Suggested requires: Contract.Requires(array != null);	BinarySearch.cs	13	7	BinarySearch
ⓘ	2	CodeContracts: Suggested requires: Contract.Requires(array != null);	Incrementer.cs	22	7	BinarySearch
ⓘ	4	CodeContracts: Suggested requires: Contract.Requires(index < array.Length);	Incrementer.cs	22	7	BinarySearch

---

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

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

```
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.