

Leap Year Sample

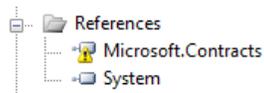
Abstract

This sample shows how to use the static checker to do a form of “abstract debugging” by examining the termination property of a loop computing the year and day given the number of days since 1980. It also exemplifies how the expressiveness of contracts enables this kind of debugging across procedure boundaries by means of communicating established invariants.

1 Adding the Contract Library Reference

1.1 Visual Studio 2008

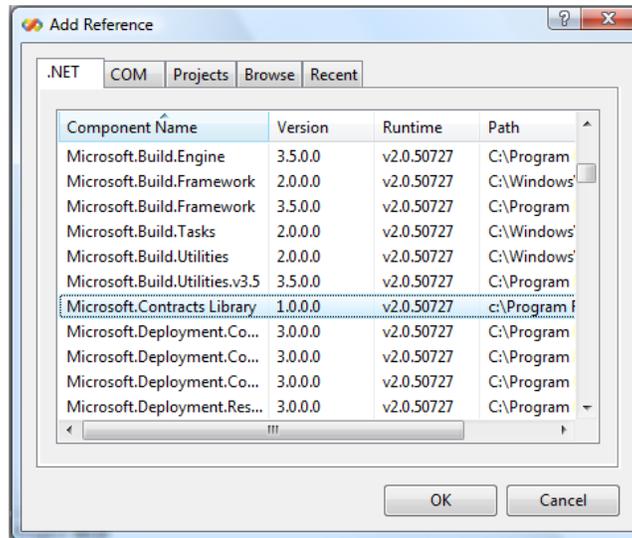
Before getting started with the sample, we need to make sure we have the proper reference for Microsoft.Contracts.dll. Look at the references of project ZuneDate:



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 LeapYear solution and right-click on References in the ZuneDate 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:

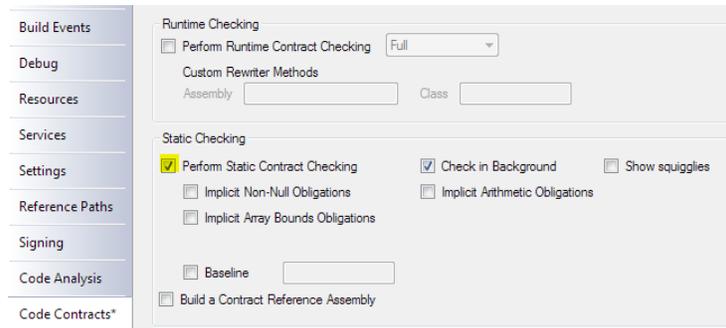
- Target the v3.5 framework (the sample default). In this case, follow the instructions under Visual Studio 2008 as this mode will require the Microsoft.Contracts library to be referenced.
- Target the v4.0 framework. In this case, the contract class is defined in mscorlib and *no* reference to Microsoft.Contracts.dll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

After adding the proper reference, go to the Properties of project ZuneDate, select the Code Contracts pane (at the bottom), and enable static checking by clicking on the static checking box.



3 Overview

On December 31st, 2008, some older Zune models hung during boot. The error turned out to be in some BIOS code that computes the current year from the number of days since 1980. Take a look at the method `YearSince1980` in `ZuneDate.cs`.

The method's intention is to compute the current year and the day within that year, given the number of days since 1980. It does so by repeatedly subtracting 365 days from the days left, until the remaining days fall within a year. Of course, the code has to take care of leap years and subtract 366 days in that case.

4 Proving Termination with Variants

A variant of a loop is the quantity that decreases on each iteration. If a loop has a variant and the loop exit condition triggers when the variant reaches a certain point, then the loop terminates.

In our example, the loop variant should be the number of days left or `daysLeft`. As we can see, the loop exits when we reach `daysLeft <= 365`. To prove termination, all we have to do is show that `daysLeft` decreases on each iteration.

To do so, let's add the following 2 lines of code around the existing loop body:

```

while (daysLeft > 365)
{
    var oldDaysLeft = daysLeft;
    <existing loop>
    Contract.Assert(daysLeft < oldDaysLeft);
}

```

Remember you can use the code snippet `cca TAB TAB` to insert a call to `Contract.Assert`. Build the project. The build should succeed. After a moment¹, the static

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

checker should warn that the assert we just added is unproven (ignore any warnings you might be getting from the Client project at this point).

✖ 0 Errors ⚠ 1 Warning ℹ 1 Message				
	Description	File	Line	Column
⚠ 1	contracts: assert unproven	ZuneDate.cs	32	9
ℹ 2	contracts: Checked 1 assertion: 1 unknown	ZuneDate.dll	1	1

This tells us that indeed, there might be a problem with the termination of this loop. If you look more closely at the loop body, you'll see the inner **if** has no **else**, which is suspicious. Let's test our hypothesis that going through that missing **else** is a problem by adding the following **else**-branch there:

```

    if (daysLeft > 366)
    {
        daysLeft -= 366;
        year += 1;
    }
    else
    {
        Contract.Assert( false );
    }

```

Putting an `Assert(false)` at a particular program point is a way of saying, I expect to never reach this point. If we build again, we get the following warning:

✖ 0 Errors ⚠ 1 Warning ℹ 1 Message				
	Description	File	Line	Column
⚠ 1	contracts: assert is false	ZuneDate.cs	28	13
ℹ 2	contracts: Checked 2 assertions: 1 correct 1 false	ZuneDate.dll	1	1

The `Assert(false)` we just added cannot be proven, meaning the checker thinks that it is reachable. Note however that the assert at the end of the loop asserting our variant is now proven. The checker reasons that since the `Assert(false)` expresses that we should never get to the **else**-branch, it will only consider all the other control flow paths. For the remaining paths, it can prove that the loop will terminate.

5 Fixing the Code

Let's examine what should happen in the problematic **else**-branch. We get into the else branch when we have more than 365 days left, the year is a leap year, but there are not more than 366 days left. Well, this means that `daysLeft` is exactly 366. Let's test this insight with an assertion:

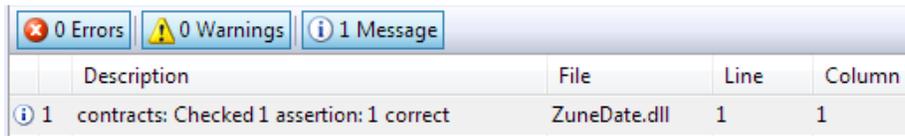
```
else
{
    Contract.Assert(daysLeft == 366);
    Contract.Assert( false );
}
```

Build and look at the warnings. You'll see that it proves this assert correct, as it still complains about the `Assert(false)`. Good, our intuition is correct. Now what should happen in this case? We are in a leap year and we have 366 days left. This means we are on the last day of this leap year, namely December 31st. In this case, the method should return the year and the day in the year should be 366.

Let's fix the code by changing the `else`-branch to the following:

```
else
{
    dayInYear = daysLeft;
    return year;
}
```

We could of course just set `dayInYear` to 366, as we have just shown that it will be the case. Now let's rebuild and see what the checker tells us.



The screenshot shows a status bar with three buttons: '0 Errors' (red X icon), '0 Warnings' (yellow triangle icon), and '1 Message' (blue information icon). Below the buttons is a table with the following content:

	Description	File	Line	Column
1	contracts: Checked 1 assertion: 1 correct	ZuneDate.dll	1	1

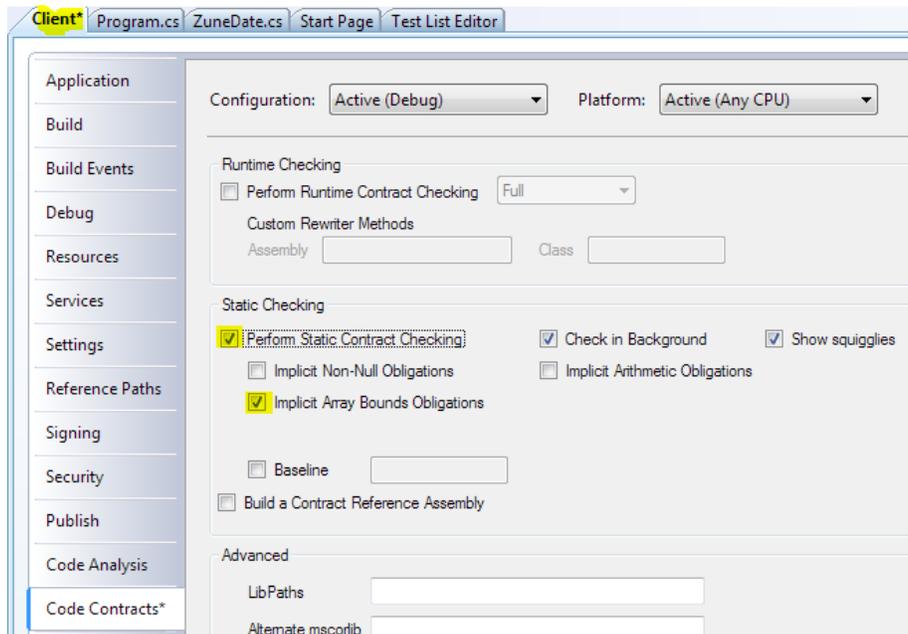
Now the checker proves the assert at the end of the loop that the `daysLeft` decreases each time around the loop. Thus we can be assured that the loop terminates.

So far, we have used the static checker to perform “abstract debugging” without ever running the code. By seeing which program points are reachable and verifying our hypotheses by inserting assertions that the checker then tries to discharge, we discovered the missing corner case and validated our assumption that it happens when the day is the last day of a leap year.

For this simple example, we didn't need any further contracts. In the next sections, we show how contracts can help provide this kind of abstract debugging for callers of code.

6 Checking the Client

Now it is time to look at the client in `Program.cs`. Let's enable static checking on the Client project, including array bound validation:



Build the Client project (or the solution). The checker will report that it cannot prove the array access below:

```
days[dayInYear] = "used";
```

The client code seems to be getting a day since 1980 from the command line, calling `YearSince1980`, allocating an array of days of size 366, and then assigning to the element specified by the returned `dayInYear`.

Why is the static checker not able to prove the array access as safe? After all, we just looked through the `YearSince1980` method and saw that it returns the day in the year.

We can try to narrow down the problem by inserting assertions about these assumptions right after the method call to `YearSince1980`.

```
int year = ZuneDate.YearSince1980(daysSince1980, out dayInYear);
Contract.Assert(dayInYear >= 0);
Contract.Assert(dayInYear <= 366);
```

Let's run the checker again and see what it thinks.

✖ 0 Errors ⚠ 3 Warnings ℹ 1 Message				
	Description	File	Line	Column
⚠ 1	contracts: assert unproven	Program.cs	23	7
⚠ 2	contracts: assert unproven	Program.cs	24	7
⚠ 3	contracts: Array access might be above the upper bound	Program.cs	29	7
ℹ 4	contracts: Checked 10 assertions: 7 correct 3 unknown	Client.exe	1	1

The checker cannot prove either of our assertions. But note that it now complains only about the upper bound of the array. Thus, if the `Assert(dayInYear >=0)` is true, the array bound access later is at least respecting the lower bound of the array. Why isn't it also proving the upper bound?

The answer is of course that arrays are 0 indexed, but days in the year are between 1..366. Our array has valid indices between 0..365. This is a classical "off-by-one" error. Let's correct it:

```
days[dayInYear - 1] = "used";
```

and rebuild the project.

0 Errors 3 Warnings 1 Message				
	Description	File	Line	Column
1	contracts: assert unproven	Program.cs	23	7
2	contracts: assert unproven	Program.cs	24	7
3	contracts: Array access might be below the lower bound	Program.cs	29	7
4	contracts: Checked 10 assertions: 7 correct 3 unknown	Client.exe	1	1

Look at the third warning about the array access: now the checker is complaining that the lower bound may not be respected, but the upper bound is now okay. Looking back at our assertions, we can see that we only asserted `dayInYear >= 0`, where as the code later assumes `dayInYear >= 1`. Let's fix the assert.

```
Contract.Assert(dayInYear >= 1);
```

and rebuild. Now the checker should only warn about the two assertions, but no longer about the array accesses, since the assertions are now enough to imply that the array access will be within bounds.

How can we prove these asserts correct now? The static checker works modularly, one method at a time, using the contracts of called methods to understand what is going on at method call sites. In this case, there are no contracts on `YearSince1980`, and thus the static checker will assume that `dayInYear` in the `Main` method could be any integer. That's why the assertions are not proven. To fix this, we need to go back to `YearSince1980` and add contracts there.

7 Contracts on YearSince1980

The assumption the client is making about the return value of `dayInYear` is that it is between 1 and 366. Let's make this explicit as a post condition of the method:

```
public static int YearSince1980(int daysSince1980, out int dayInYear)
{
    Contract.Ensures(Contract.ValueAtReturn(out dayInYear) >= 1);
    Contract.Ensures(Contract.ValueAtReturn(out dayInYear) <= 366);
}
```

Note how we need to use the method `Contract.ValueAtReturn` to refer to the final value of out-parameters of a method. Currently the tools do not check to make sure that out parameters are initialized properly disregarding their mention in the postcondition. Thus, in the above example, if the code after the contracts uses the value of `dayInYear` before assigning to it, the C# compiler would not issue the error that it should. However, on a build where the `CONTRACTS_FULL` is not defined (such as `Release`), the compiler will issue an error.

Build the `ZuneDate` project and observe the output:

0 Errors 2 Warnings 1 Message				
	Description	File	L.	Column
1	contracts: ensures unproven	ZuneDate.cs	44	7
2	+ location related to previous warning	ZuneDate.cs	13	7
3	contracts: Checked 5 assertions: 4 correct 1 unknown	ZuneDate.dll	1	1

The checker validates the upper bound (366) on the final value of `dayInYear`, but not the lower bound of 1. If you consider the code for a second, you will notice that indeed a negative parameter would cause that problem, as the loop would then never execute.

To avoid this, let's add the necessary pre-condition:

```
public static int YearSince1980(int daysSince1980, out int dayInYear)
{
    Contract.Requires(daysSince1980 >= 1);
}
```

Now rebuild the entire solution. You should see that the checker validates all contracts on the `ZuneDate` project, but that there is now a violated requires in the client.

0 Errors 2 Warnings 2 Messages				
	Description	File	Line	Column
1	contracts: Checked 5 assertions: 5 correct	ZuneDate.dll	1	1
2	contracts: requires unproven: daysSince1980 >= 1	Program.cs	22	7
3	+ location related to previous warning	ZuneDate.cs	13	7
4	contracts: Checked 11 assertions: 10 correct 1 unknown	Client.exe	1	1

The client code does not perform any validation on the parsed `daysSince1980` value before passing it to `YearsSince1980`, thus, the value may not be positive. We can easily fix the client code by writing the extra return below before computing the year:

```
int daysSince1980 = Int32.Parse(args [0]);
if (daysSince1980 <= 0) return;
```

When you rebuild, the checker should now validate the client code as well.

8 More Detailed Contracts

Now suppose the client code programmer were really worried about memory consumption and wants to allocate an array that matches the number of days needed exactly, rather than overprovisioning for leap years. In other words, the client code in `Main` changes as follows:

```
int dayInYear;
int year = ZuneDate.YearSince1980(daysSince1980, out dayInYear);

string [] days;
if (DateTime.IsLeapYear(year))
{
    days = new string[366];
}
else
{
    days = new string[365];
}
days[dayInYear - 1] = "used";
```

As you can see, we now allocate either 366 or 365 elements for the `days` array, depending on the leap year status of the returned year. When you build, the checker will report that it no longer proves the upper bound on the array access.

The reason for this warning is the same as at the beginning of this sample. The contract on method `YearSince1980` is not detailed enough to specify that in case the returned year is not a leap year, then the `dayInYear` is bounded by 365.

Fortunately, if we want to, we can actually specify this and also check that the implementation satisfies this property. Going back to the implementation of `YearSince1980`, add the following postcondition:

```
Contract.Ensures(DateTime.IsLeapYear(Contract.Result<int>()))
    || Contract.ValueAtReturn(out dayInYear) <= 365;
```

The specification states that either the returned year is a leap year, or the final `dayInYear` value is bounded by 365. When you rebuild the solution, you should see that the new postcondition is proven, meaning our date computation is looking pretty solid.