Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract

Mike Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann
Microsoft Research, One Microsoft Way, Redmond, WA, 98052-6399, USA
{mbarnett, maf, jhalleux, logozzo, nikolait}@microsoft.com

Abstract

This demonstration presents two tools, Code Contracts and Pex, that utilize specification constructs for advanced testing, runtime checking, and static checking of object-oriented .NET programs.

1. Introduction

Over the last few years we have been working on influencing the way programmers develop .NET software through two related projects: Code Contracts and Pex. *Code Contracts* provides a specification technique for expressing method pre- and postconditions as well as object invariants. These specifications are then used for runtime checking and static checking. The specifications are also understood by the advanced unit-testing tool, Pex. *Pex* performs a white box code analysis using a constraint solver to determine relevant test inputs. Preconditions allow pruning of irrelevant test inputs, and postconditions guide test generation and allow detecting bugs; object invariants serve both purposes. Furthermore, Pex enables the concept of Parameterized Unit Tests which are essentially parameterized usage scenarios annotated with specifications to state assumptions and assertions. The result of Pex's analysis is a small unit test suite which often achieves high code coverage.

Both tools can be invoked through the commandline, and they integrate with Microsoft Visual Studio through add-ins.

2. Specification Language

We are targeting the specification language and tools at the general developer, not the verification enthusiast. It is thus important to use a single form of specifications that meets three simultaneous goals:

1. Specifications serve as documentation. They must be as readable as possible.
2. Specifications should be executable. This motivates writing specifications for testing and immediate perceived benefit, without consideration of static verification.
3. Specifications simplify static verification.

Our specification approach is language-agnostic: we use idiomatic code written in the developer's source language to express them. Preconditions and postconditions are expressed as calls to the static methods Contract.Requires and Contract.Ensures. Special dummy methods are used to refer to the method's return value as well as referring to the *old* value of an expression, meaning the value of the expression on method entry. The conditions in the example in Figure 1 are written in C# expression syntax.

```
int Increment(int value, string label) {
  Contract.Requires( value > 0 );
  Contract.Requires( label != null );
  Contract.Ensures( Count ==
                    Contract.OldValue(Count) + value );
  Contract.Ensures( Contract.Result<int>()
                    == Contract.OldValue(Count) );
  ...
```

Figure 1. An Increment specification in C#

A language-agnostic approach has many advantages:

- Developers need not learn a new language for specifications. Predicates are boolean conditions expressed in the source language.
- No new front-ends or compilers are required. Standard compilers directly translate contracts into .NET intermediate language (MSIL). As a benefit, compilers check the syntax and typing of contract conditions, thus avoiding errors in specifications, such as unresolved names, that would arise if the specifications were written in comments or attributes.

- Standard development environments help writing specifications in the same way they help writing other code, via highlighting, intellisense, completion, etc.
- The semantics of contracts is defined by that the semantics of the generated MSIL. The compiled code acts as a persisted format of specifications consumable by a variety of tools.

The language independence extends from the specification language to the tools themselves as they consume the generated MSIL rather than on the source.

3. Runtime Contract Checking

In a post-build step, the compiled binaries containing the calls to the contract methods are transformed by having each specification injected at the appropriate program points. For instance, method postconditions are moved to the exit points of each method and calls to Contract.Result are replaced with the return value of the method. At the beginning of each method, arguments to Contract.OldValue are evaluated and stored into locals which then replace those method calls within the postcondition-checking code.

In addition, contracts from supertypes and interfaces are inherited by subtypes and interface implementations. This provides the basis for enforcing behavioral subtyping [2], which is required for modular checking.

4. Static Checking

Our static checker is based on abstract interpretation rather than SMT solvers traditionally used for program verification, in order to automate the generation of loop invariants and strongest postconditions.

Although we use modular verification which, in principle, requires specifications at all method boundaries, we use techniques to infer pre- and postconditions whenever possible.

The existing abstract domains provide an analysis that checks for null dereferences, array indexing, arithmetic overflow, in addition to user-defined general properties.

5. Test Generation

Pex [5] is an automatic white-box test generation tool for .NET. Starting from a designated method, Pex explores the statements of all transitively reachable callees using *dynamic symbolic execution* [1]. Pex uses a constraint solver to compute test inputs that exercise particular program paths. Pex employs heuristic search strategies to select paths that are likely to be feasible and exhibit different program behaviors. As a result, Pex often finds relevant test inputs quickly and fully automatically, while still exercising all feasible execution paths eventually.

When trying to determine relevant execution paths, Pex considers explicit conditional branch statements as well all operations that can cause exceptional control flow changes as a side effect. Furthermore, Pex leverages preconditions, postconditions, and object invariants as follows. The compiled binaries are transformed by a post-build step as described earlier, so that all specifications are injected. This results in additional conditional program branches which distinguish the case when the specification holds from the case where it does not hold. As a result, Pex will try to exercise both cases during its program analysis. When test inputs cause a top-level precondition or object invariant violation, these test inputs are discarded and are not shown to the user. In all other cases of specification violations, the result is a test case that is flagged as a failure.

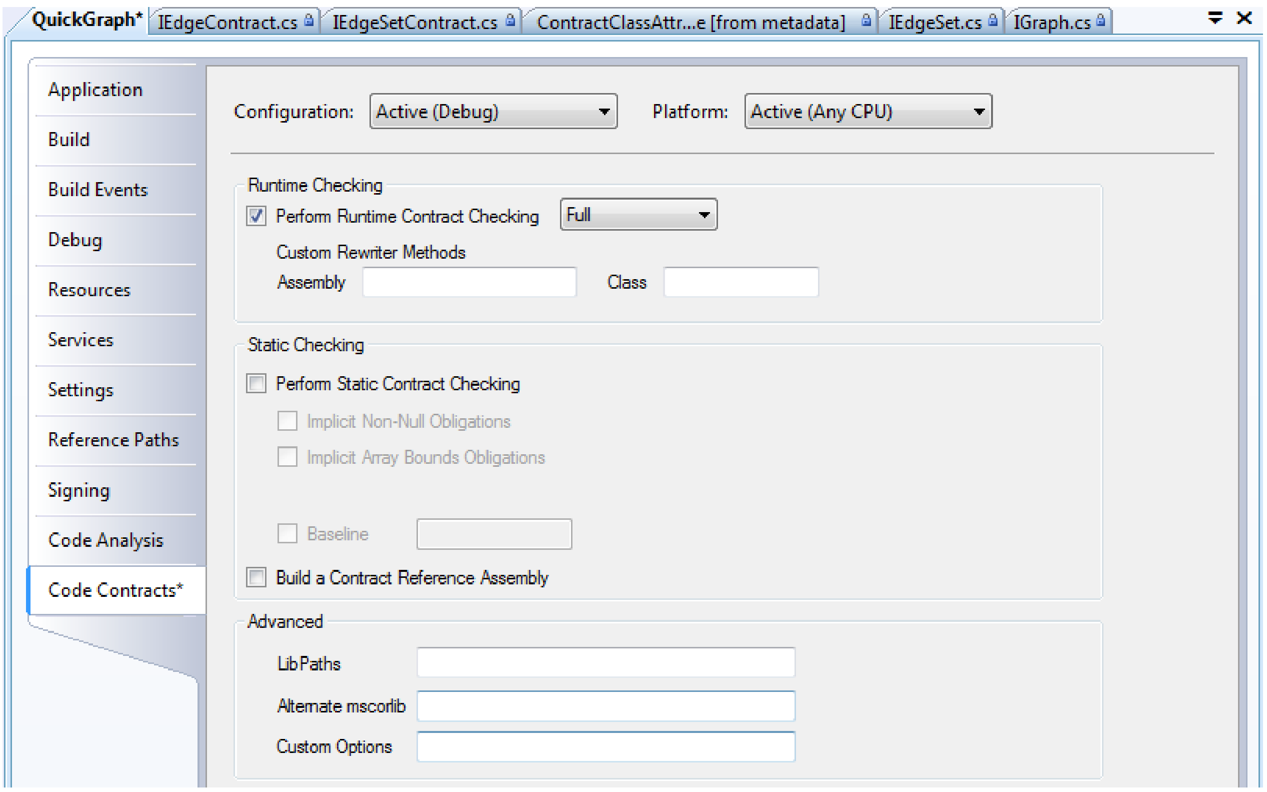
The analysis produces a set of unit tests, saved in a file as source code. Each test sets up particular test input, and then calls the entry point method. Pex can be configured to support the unit test idioms of various unit testing frameworks, including NUnit [3] and MSTest [4].

6. Parameterized Unit Testing

A parameterized unit test (PUT) is simply a method that takes parameters, states assumptions on the arguments, performs a sequence of method calls that exercise the code-under-test, and asserts properties of the code's expected behavior. PUTs are a generalization of traditional unit tests. While preconditions, postconditions, and object invariants can only express specifications that must hold at method boundaries, PUTs allow to express properties that span multiple method calls.

7. Synergies

We already touched upon how runtime checked contracts improve automated test generation by providing more assumptions and test outcomes. A drawback of static checking is the presence of false warnings and the effort to determine whether a warning is warranted or not. Pex often helps alleviate this problem by being able to point out inputs to a method that actually trigger the violation. Pex reports the issues as source code that can be debugged immediately by the developer. Furthermore, Pex can suggest new preconditions at appropriate places to avoid problems it finds,



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
[ ] Implicit Non-Null Obligations
[ ] Implicit Array Bounds Obligations
[ ] Baseline
[ ] Build a Contract Reference Assembly

Advanced
LibPaths:
Alternate mscorlib:
Custom Options:

Figure 2. The Code Contact User Interface

thereby strengthening the specifications and closing the cycle.

8 Demonstration Description

The demonstration will consist of a tag-team talk between a member of the code contract team and a member of the Pex team. During the demonstration, we will live author code and run the static analysis tool, which will point out errors. We will use Pex to find counter examples that exhibit the erroneous behavior. To guard against these errors, we will author more contracts, in particular object invariants. Running our tools again, will find more places where the code is either incomplete or wrong and we will author live fixes. We also show how Pex is used to obtain a series of unit tests that provide very high-coverage of the module being developed.

9 Screenshots

Figure 2 shows the contract user interface in Visual Studio. The UI allows the programmer to enable runtime checking and/or static checking. The integration in the IDE manages all the extra build steps transparently.

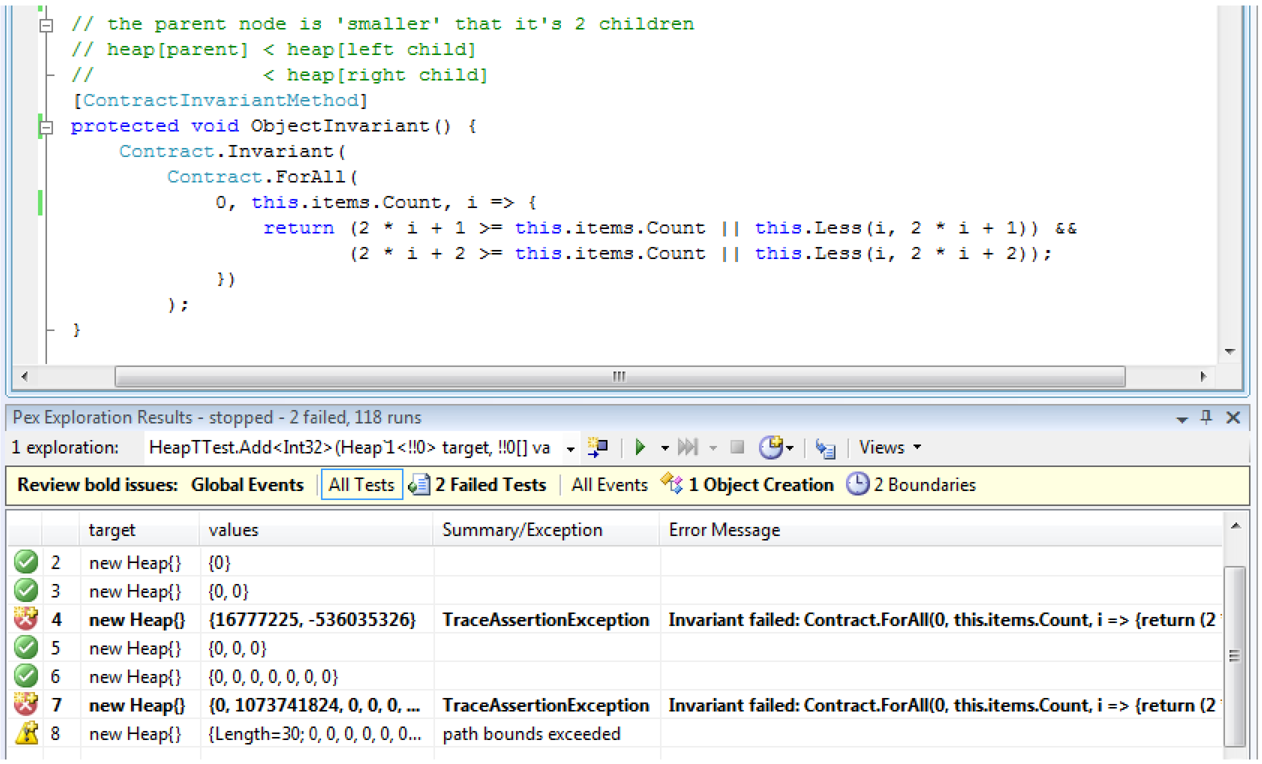
As an example of the language-agnostic approach, the same specification as in Figure 1 can be written in Visual Basic as shown in Figure 4.

```
Function Increment(ByVal value As Integer) As Integer
  Contract.Requires( value > 0 )
  Contract.Requires( label IsNot Nothing )
  Contract.Ensures( Count
                    = Contract.OldValue(Count) + value )
  Contract.Ensures( Contract.Result(Of Integer)()
                    = Contract.OldValue(Count) )
  ...
```

Figure 4. An Increment specification in Visual Basic

Figure 3 shows the output of Pex during test generation. Test entries in the list on the left marked green are succeeding tests, while red ones are failing tests. Test number 4 found an input that violated the invariant of a binary heap.

Figure 5 shows the default contract failure behavior



```
// the parent node is 'smaller' that it's 2 children
// heap[parent] < heap[left child]
//              < heap[right child]
[ContractInvariantMethod]
protected void ObjectInvariant() {
    Contract.Invariant(
        Contract.ForAll(
            0, this.items.Count, i => {
                return (2 * i + 1 >= this.items.Count || this.Less(i, 2 * i + 1)) &&
                       (2 * i + 2 >= this.items.Count || this.Less(i, 2 * i + 2));
            })
        );
}
```

Pex Exploration Results - stopped - 2 failed, 118 runs

1 exploration: HeapTTest.Add<Int32>(Heap`1<!!0> target, !!0[] va

Review bold issues: **Global Events** | All Tests | **2 Failed Tests** | All Events | **1 Object Creation** | 2 Boundaries

		target	values	Summary/Exception	Error Message
✓	2	new Heap{}	{0}		
✓	3	new Heap{}	{0, 0}		
✗	**4**	**new Heap{}**	**{16777225, -536035326}**	**TraceAssertionException**	**Invariant failed: Contract.ForAll(0, this.items.Count, i => {return (2**
✓	5	new Heap{}	{0, 0, 0}		
✓	6	new Heap{}	{0, 0, 0, 0, 0, 0, 0}		
✗	**7**	**new Heap{}**	**{0, 1073741824, 0, 0, ...**	**TraceAssertionException**	**Invariant failed: Contract.ForAll(0, this.items.Count, i => {return (2**
⚠	8	new Heap{}	{Length=30; 0, 0, 0, 0, 0, 0...	path bounds exceeded	

Figure 3. Screenshot of Pex Output

when runtime checking of contracts is enabled. In this case, an invariant was violated.

Figure 6 shows the output produced by a run of the static contract verifier.

References

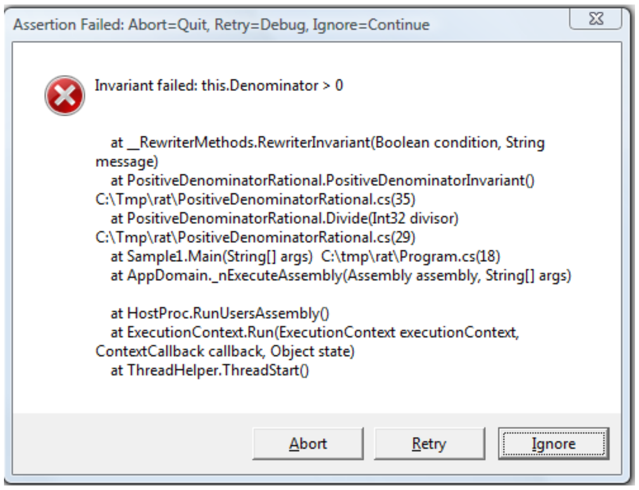
[1] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.

[2] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[3] Michael C. Two, Charlie Poole, Jamie Cansdale, Gary Feldman, James W. Newkirk, Alexei A. Vorontsov and Philip A. Craig. NUnit. http://www.nunit.org/.

[4] Microsoft. Visual Studio Team System, Team Edition for Testers. http://msdn2.microsoft.com/en-us/vsts2008/products/bb933754.aspx.

[5] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.



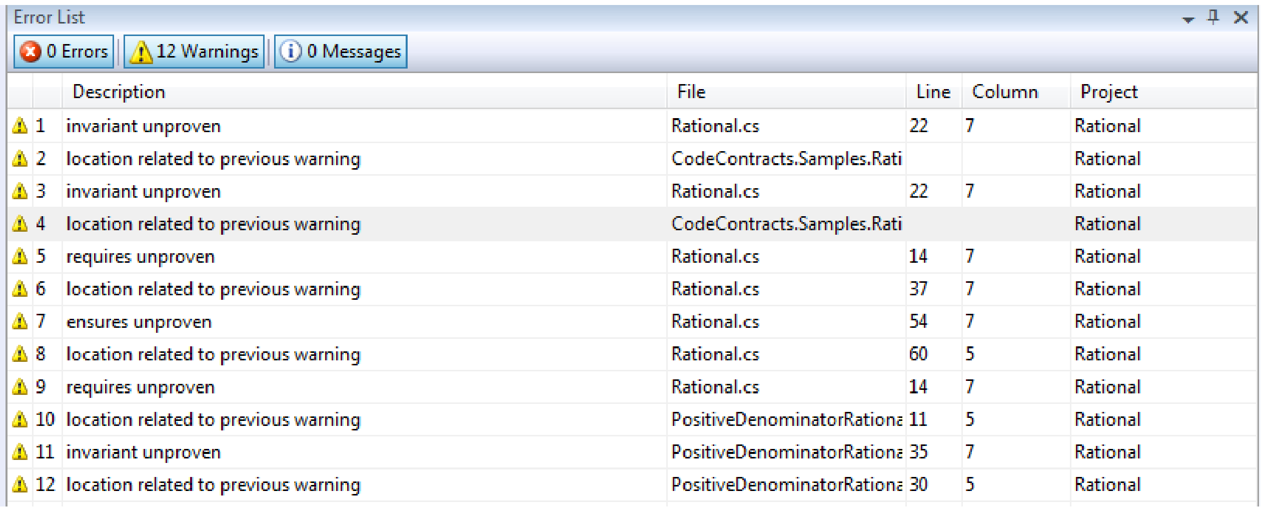
Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue

```
Invariant failed: this.Denominator > 0


   at __RewriterMethods.RewriterInvariant(Boolean condition, String
message)
   at PositiveDenominatorRational.PositiveDenominatorInvariant()
C:\Tmp\rat\PositiveDenominatorRational.cs(35)
   at PositiveDenominatorRational.Divide(Int32 divisor)
C:\Tmp\rat\PositiveDenominatorRational.cs(29)
   at Sample1.Main(String[] args)  C:\tmp\rat\Program.cs(18)
   at AppDomain._nExecuteAssembly(Assembly assembly, String[] args)

   at HostProc.RunUsersAssembly()
   at ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state)
   at ThreadHelper.ThreadStart()
```

Figure 5. Example of Runtime Contract Failure



		Description	File	Line	Column	Project
⚠	1	invariant unproven	Rational.cs	22	7	Rational
⚠	2	location related to previous warning	CodeContracts.Samples.Rati			Rational
⚠	3	invariant unproven	Rational.cs	22	7	Rational
⚠	4	location related to previous warning	CodeContracts.Samples.Rati			Rational
⚠	5	requires unproven	Rational.cs	14	7	Rational
⚠	6	location related to previous warning	Rational.cs	37	7	Rational
⚠	7	ensures unproven	Rational.cs	54	7	Rational
⚠	8	location related to previous warning	Rational.cs	60	5	Rational
⚠	9	requires unproven	Rational.cs	14	7	Rational
⚠	10	location related to previous warning	PositiveDenominatorRationa	11	5	Rational
⚠	11	invariant unproven	PositiveDenominatorRationa	35	7	Rational
⚠	12	location related to previous warning	PositiveDenominatorRationa	30	5	Rational

Figure 6. Example of Static Checker Output