

Designing a NoException Library Using Code Contracts

Erik Dietrich
Anu Pardeshi
John Svitek

Abstract

Many coding projects use exceptions to handle incorrect program states, either caused by bad inputs to functions or runtime failures, and exceptions are the de facto means of handling any error case in the .NET framework. However, exceptions are not always desirable. They place a heavy burden on the calling code to catch and rectify any thrown exceptions. In this paper, we discuss an implementation in C# of some basic types that are guaranteed to never throw an exception, opting instead to always return a sensible default value. To prove that provided classes will not throw an exception, we use a Design by Contract paradigm, further strengthened by exploring the classes with Pex. We also discuss the decisions made that led to this design, and our experiences in learning and utilizing Code Contracts and Pex.

1. Introduction

For our project, we were interested in using the concepts of design-by-contract and static analysis to create code that is guaranteed not to throw exceptions. In object oriented languages, exceptions are mechanisms that break the normal execution flow of the application at run-time, forcing the application (or the OS, if necessary) to handle the error condition. Many times this is desirable, both for exposing errors and forcing them to be handled, but there are times when it is not. Our goal is to provide a utility that makes it easier for developers to express that desire semantically and to reason about exceptions in general.

1.1 Motivation

In object oriented programming (OOP), runtime exceptions are rather demanding of client code, both in terms of reasoning and in terms of developer effort. They require a client developer to understand what exceptions could be thrown by invoking a method and to explicitly handle them with additional code. However, not all client code is created equally. There are many instances where a failure, even an exception-generating one, is considered non-critical by the client in question. If, for instance, a class maintains an array of integers and allows callers to access the array's elements, the class's developer needs to check array bounds before accessing it or exceptions may be thrown. But it is entirely possible that the developer does not want to be bothered with this and prefers the general reasoning that any index for which the integer array has no value should return 0.

There are three generally accepted modes of exception handling: the basic guarantee, the strong guarantee, and the no-throw guarantee. [1] Our interest is in the last guarantee. The no-throw guarantee ensures that no exceptions will be generated at all.

We aim to provide a no-throw solution that allows developers to make this intention clear. By using our code, developers can specify that they do not want an exception to be thrown and would prefer a sensible default return value for accessors or default behavior for mutators. This is ideal for non-critical operations whose failures do not constitute problems and should not interrupt normal execution flow.

1.2 NoException Project

Toward this end, we created the NoException project. We created a series of classes that mirror existing fundamental classes and literals. The difference is that our versions never throw exceptions. Users of them can count on this fact, making it easier to reason about the control flow of their own code, and also making it easier to provide the no-throw guarantee to others, since it is much easier not to throw exceptions when the code with which one works does not, itself, generate exceptions.

We feel that this has the potential to be a significant contribution as it lays the groundwork for a fundamentally different and more explicit way that components can communicate regarding exceptions. With broadly adopted use, we believe that development could be made easier in both the design and debugging phases.

2. Design

Given our stated goals above, we had a variety of options for languages in which to implement the scheme as well as tools to use to help ourselves. Ultimately, we chose to do our implementation in C#[2] with Visual Studio 2010[3], using Code Contracts[4], Pex[5], and Microsoft Test[6] to assist our effort. In this section, we describe why we made each of these decisions and how we thought they would contribute to our stated goal.

2.1. Using C#

Our first decision was to pick a programming language to use. Java, C++ and C# are all major OOP languages that have exception semantics and would be reasonable candidates for our project. We decided against C++ because we felt that a managed language [7] running in a VM would be a more suitable choice than a native language. A managed language provides more potential opportunity for analysis, given that it supports reflection and more degrees of introspection at run time.

Left to choose between Java and C#, we thought that C# was a better choice because of its exception handling paradigm. In Java, checked exceptions force callers to be aware of and handle exceptions. [8] The merits of this behavior are frequently debated, but the nature of this is such that it mandates the coding boilerplate that we seek to eliminate and provides the handling we seek to obviate, so Java is not an optimal choice. Our utility would be more useful to C# developers.

2.2 MS Test

With a language chosen, we considered the available options for a manual testing framework. We felt that manual testing was important, given the bold nature of seeking to provide the no-throw guarantee. Our available options in the .NET framework were NUnit[9], MBUnit[10] and MS Test. We chose MS Test mainly because it integrates tightly with Visual Studio and would thus cause the least amount of setup and integration overhead with the other Microsoft technologies that we had chosen to use. [11]

2.3 Code Contracts

With language and testing in place, the next decision was how to use Code Contracts. Code Contracts is a .NET library that standardizes condition and invariant checking and allows both compile-time (static) and run-time (dynamic) enforcement of the same. By its nature, Design by Contract [12] is a means of making code more explicit and easy to reason about. Using contracts, we have the ability to specify and enforce preconditions, post conditions, and invariants. A careful consideration of our problem domain reduces the issue to one mainly of preconditions and invariants. That is, when some piece of code throws an exception, it is because some precondition is not met or some invariant is not preserved.

As such, it stood to reason that we could use the MS Code Contracts to clearly list exception causing inputs and actions. Once these were enumerated, it would be relatively simple to make a decision as to how No Exception classes would handle these conditions.

2.4 Microsoft Pex

With most of our high-level design in place, and after a bit of proof of concept work, we made the decision also to use Microsoft Pex. Pex is a utility from Microsoft research that allows generation of parameterized unit tests as well as automatic, “white box” generation of test suites that find “interesting” (i.e. corner case) inputs and outputs of methods.

We found Pex to be appealing because of our desire to generate as much code coverage as possible in a relatively short time-frame, thus ensuring as high a degree of accuracy in our guarantee as we could. Pex is a natural compliment to manual unit testing as manual unit testing explicitly documents the intentions of the code authors while test generation tends to find test cases that the author may not have considered.

3. Implementation

With a specific goal in mind and the tools all selected and configured, our next task was the actual implementation. For this phase of development, we chose nine basic classes for which to provide a No Exception guarantee and divided the work of implementation evenly into thirds. We chose basic C# classes: Array, Dictionary, Double, HashTable, Int, List, Math, Stack, and String. These were selected because, in our experience, they are among the most commonly used classes and would thus provide the most utility to prospective users. Another benefit to these choices is that these classes tend to serve as building blocks for other, more in-depth implementations for which there may be a desire, later, to expand the no-throw guarantee.

3.1 Wrapped Classes

With the work divided up, the first major consideration of implementation was how to actually represent the classes in question to users. We considered some approaches such as a series of static methods or classes that inherit from the classes we want to represent. We decided against the former approach because a procedural implementation is somewhat unappealing for representing objects in an object-oriented language, and we decided against the latter because, among other things, some of our targets are (or wrap) C# *value objects*, [13] which do not support inheritance.

We ultimately decided to represent our target objects by wrapping them and using dependency injection. [14] So, for example, List would be represented by the class NEList, which takes a (non-null) List as a constructor parameter and stores a reference to a copy of the list. Making a copy is critical because it prevents users from injecting it and then creating inconsistencies *vis-à-vis* the invariants in the wrapped, NoException class by modifying the wrapped class outside of the scope of the NE class.

For actually implementing the desired functionality, we decided to invoke the underlying class's methods once we were satisfied that the user was not attempting to create conditions that would result in exceptions. The rationale for this is that it provided the most code reuse and fidelity to the user's original intentions.

By creating a copy of an injected object and then providing access to its operations only through the wrapping class, we gain the ability to interpose between user requests and actual operations on the target class. In this fashion, we can examine the user's desired actions and decide whether or not they would create exceptional conditions if applied to the wrapped class.

3.2 Providing Sensible Defaults

Once the scheme of wrapping the target class was in place, the next logical question was what to do when the user supplied a parameter or requested an operation that would result in an exception. Since we are stopping exceptions before they happen, but not, obviously, before the user created a potentially erroneous state, we are left in the position of having no good response to the request.

The route we chose was the notion of sensible defaults. So, for example, if the user has an array of integers and accesses an element outside of the bounds of the array, we decided to return 0. We made similar decisions on a case-by-case basis throughout our implementation. The idea of sensible defaults is clearly rather subjective, but we are offering the user a choice between runtime exceptions and something less invasive to the application at large. In essence, we are allowing them to trade undefined behavior and crashing for potentially incorrect behavior. While there are no good choices here, we believe that users may sometimes favor this one. We did also consider the notion of allowing users to specify the defaults they want in advance, and that is addressed in section 5.3

3.3 Debug Versus Release Modes

One of our more nuanced goals of this project was to allow different behaviors of NE classes in production code versus code in development. Specifically, we felt that our users would most likely want to know about violated invariants and preconditions while developing but be able to turn it off when the code was released.

Visual Studio supports multiple build configurations and they supply two defaults: Debug and Release modes. [15] In Debug mode, the Just-In-Time (JIT) [16] compiler forgoes some inline optimizations to allow developers to faithfully step through their code. In Release mode, this feature is turned off and managed code is optimized as much as possible. As such, these two modes may be thought of as development and production modes.

Visual Studio lets users configure settings differently, depending on the different build modes. Code Contracts follows this convention, and consequently, we were able to configure it differently for Debug and Release. We configured the contracts only to perform runtime checking in Debug mode.

So, we decorated our methods with contract precondition and post-condition statements and our classes with invariants, but we also added Boolean checks for the various conditions as well. In Debug mode, contract violations would throw exceptions, allowing developers to see that they were violating preconditions. In Release mode, the contracts would have no effect, but our Boolean guard conditions would trigger our methods to supply the default return values or behaviors instead of invoking the underlying class and generating an exception. In neither case would the underlying class actually generate the exception. It was only a question of whether we threw it from our contracts or whether we avoided it through our Boolean guard conditions.

3.4 How We Implemented Contracts

To be more specific about the two approaches mentioned in 3.3, all of our methods correspond to methods in the wrapped class. We standardized a very repeatable template for implementing these methods. Any method that provided only accessor functionality was decorated with the [Pure] attribute. At the beginning of each method is a list of Contract.Requires() statements, corresponding to a Boolean condition that must be true in the wrapped method to prevent an exception. Where appropriate, we also supplied Contract.Ensures() methods so that users (and other methods in our classes that might use our method) could more easily and concretely reason about the effects of the method. After all of the Requires and Ensures statements, we declared a Boolean variable and set it equal to the conjunction of all of the preconditions. If the Boolean was satisfied, the method invokes the wrapped method and returns its value or performs its action. If the Boolean was not satisfied, we supply the sensible default return value or behavior.

Finally, each NE class supplies an ObjectInvariant() method that declares all object invariants. These tended to be limited in our implementation as most of the relatively primitive and abstract wrapped classes do a satisfactory job of preserving their own invariants by defensively throwing exceptions.

3.5 How We Used Pex

With our core implementation in place, and basic manual unit tests written to document and verify our intentions, we installed and configured Pex. Pex is feature rich in that it offers a parameterized unit test API [17], a white box testing [18] API, including automated test generation, and integration with both Code Contracts and Microsoft's Moles mocking framework. [19] As a result of all these features and their configuration options, the learning curve is somewhat steep. With this in mind, we sought to keep our usage relatively simple in order to get the most utility in the shortest period of time.

Toward that end, we took advantage of the natural integration with the Code Contracts implementation. Pex processes existing Code Contracts and uses them to direct its test generation. So, we opted to use the white-box test generation feature rather than the manual API for creating parametrized tests. We also opted not to use Moles for mocking as our code is not dependent on any external entities such as files or databases that would be good candidates. To get the most out of Pex for our project, we used

the manual tests that it generated to complement our manual tests – finding edge cases we had not considered and contracts that we had missed.

4. Evaluation

To evaluate our implementation of NoException, we want to consider our goals and how faithfully we were able to recreate them, but also to evaluate Pex and Code Contracts. Our reasoning is that both products are relatively immature and in Beta phases, so our work can reveal as much about their viability for our tasks as it can about the feasibility and benefits of implementing the no-throw guarantee.

4.1 Evaluating Code Contracts

Code Contracts provides a relatively simple API that is quite powerful. It allows specification of method preconditions, object invariants and method post conditions. These can be checked statically at build-time and/or enforced dynamically at run-time. The run-time checking is accomplished via the Contract library, which wraps an exception throwing mechanism that is triggered by the evaluation of passed in Boolean values. The optional static checking is accomplished via an abstract interpretation mechanism. [20].

One of the main advantages of the contract implementation is its simplicity. It replaces the defensive if-not-throw blocks at the beginning of methods with semantically clearer “Requires” statements, along with “Ensures” statements to describe the expected result of the method. This is both easier to read and to reason about as the requirements and guarantees of each method are documented up-front and are enforced.

Another strength of this tool is the static checking. Using the abstract interpretation mechanism, the Visual Studio Contract plugin provides warnings where nulls may be dereferenced, array bounds overrun, and contracts not satisfied. The real power of this is that it moves the focus of a bad method call outside of the method (where it would be if a run-time exception were thrown) and back to the caller. A warning will be shown in the method call where a contract is not or may not be satisfied. This paradigm eliminates the frequent problem in debugging where it is unclear whether the responsibility falls to the caller or the called method to prevent errors. Contracts, and especially the static checking, make this explicit.

Another powerful feature of the static checking is that the checker will actually suggest contracts that do not, but could exist. For someone developing a set of methods with a large number of requirements, this proves invaluable. Our experience is that it becomes substantially easier to miss a contract when many contracts are required and the tool appears as likely to pick up on a missed contract opportunity when there are existing contracts as when there are none.

As much as the static analysis portion of the Contracts implementation is a strength, it is also currently a weakness at times. While we realize that abstract interpretation is an incredibly complex task and that some false positives and negatives are bound to result, these mistakes can be frustrating and have a negative impact on development time.

For instance, we discovered through experimentation that the static checker will not recognize invariants or operations on a member variable outside of the scope of a method, unless the member is accessed only through a method decorated with the “Pure” attribute. [21] This means that a member reference will be flagged as potentially null, even if the reference being non-null is specified as a class invariant.

Another example of the growing pains of Code Contracts is the fact that the `Contract.ForAll()` construct can never be proven. We experimented with this and reduced our implementation to the simplest possible example: a method that allocates a new array filled with positive numbers and returns it, ensuring that for all elements, the array members are non-negative. Contracts is unable to prove this correct.

These shortcomings certainly do not overshadow the usefulness of the tool, but they are somewhat troubling, particularly for developers who prefer the practice of continually keeping the build free of warnings and errors. As the number of warnings starts to increase due to the inadequacies of the tool, the “broken window” effect becomes apparent and the warnings mount, sometimes including warnings that developers would normally fix. Another obvious negative effect of false positives is the time wasted trying to fix them. The effect of false negatives speaks for itself, though false negative impact can be mitigated by the run-time checker and a good unit test suite.

4.2 Evaluating Pex

A good test suite is one of the aims of the Pex suite. The overriding philosophy behind Pex seems to be that of the school of unit testing that believes in automated, randomized test generation. [21] That is, Pex is useful for executing the system under test (SUT) and using an element of randomness to generate and explore “interesting” inputs and parameters. This is accomplished via a “Pex Exploration”, wherein a class or assembly is selected to be explored, and Pex runs a series of executions looking for interesting inputs with a configurable timeout. Once the run is completed, successes, failures and inconclusive results are shown and the user has the option to debug them, 'promote' them to persistent unit tests and many other things. Promoted tests are automatically generated in the form of actual unit tests in the framework of the tester's choosing (MS Test, NUnit, etc). From here, the tests are available to be re-run at the user's convenience. Pex also, in some cases, offers an option to 'fix' the issue automatically, generally by adding a contract.

Clearly, this is quite a powerful tool and the advantages are numerous. The most obvious advantage is the ability to generate a functional unit test suite from scratch, automatically. That capability drastically cuts down on the amount of time required to have functional testing and it also ensures that the test cases are non-trivial, in most situations. In our case, we were able to generate hundreds of additional unit tests in the span of an hour. It is not our aim in this paper to offer any insight on the debate between automated and manual testing, but we will say that, at the very least, this capability of Pex provides a valuable complement to manual tests which are generally more intended to document requirements and intentions than to seek out all conceivable edge cases.

The exploration functionality is another powerful aspect of Pex. Of the hundreds of tests that we were able to generate automatically, many of them found actual issues, some of which would likely only have been exposed in production with an unusual corner case. For example, Pex found a scenario in which

the bounds of an array were overrun in spite of our contract because two large 32 bit integers added together exceeded the maximum value of a 32 bit integer, rolling over to the minimum and thus satisfying the contract incorrectly. The ability of the explorations to find unusual edge cases cements their status as an excellent complement to manual testing.

Like Code Contracts, however, Pex does have its shortcomings in its current incarnation. The learning curve for using Pex is somewhat steep as it offers a great deal of functionality and the documentation for it seems to be rather spread out in various sources. As a result, Pex is rather difficult to approach for a new user and requires a fair amount of trial and error.

A more subtle issue that we noticed is that Pex explorations seem to suffer from diminishing returns. In the first explorations on some of our classes, almost 100% of the explorations marked as failures correlated with actual bugs. As we fixed these however, and continued to run more explorations, the number of false positive steadily increased until eventually the failures were almost entirely false positives. In some cases, these new false positives included the exceptions thrown by our precondition guards, meaning that Pex was supplying inputs that violated our preconditions and reporting this as a failure of the guarded method. This phenomenon creates the appearance that Pex is going to find 'issues' regardless of how stable the SUT is, rather than provide some assurance, by finding no issues, that the code is stable.

One final aspect of Pex that seems as if it could be improved is the code in the generated tests. Pex adds a lot of confusing boilerplate in the form of test method decorations and library calls that makes it somewhat difficult to see what the test setup actually is. In addition to the confusing syntax of the generated code, the tests are also spread out into a parent class and 'child' classes, all of which are linked together as *partial classes*. [23] Generally speaking, unit tests are easiest to reason about when setup is relatively simple and confined to one place so that a developer can see the test inputs and the result. Pex generated tests are neither simple nor confined to one place. This is mitigated somewhat by the ability that Pex provides to step right into the potentially failing test case and skip the setup, but that is not always an adequate substitute, particularly from the perspective of reasoning about the code.

4.3 Evaluating Ourselves

In terms of our own work, we set out with the goal of creating versions of standard C# classes that implement the no-throw guarantee and with secondary goals of proving the concept of using design-by-contract and test generation tools to do it. On balance, we feel that we were successful in this endeavor. For the classes that we chose, we implemented versions of all of their public methods and verified the correctness of their functionality with manual unit tests. The following charts contain some objective metrics to support our contention that the effort was a success.

Manual Unit Tests	Code Coverage	Contracts Defined	Pex Explorations	Successful Automated Pex Tests	Total Tests
357	88.70%	414	27	287	644

As indicated by the metrics, we achieved a significant amount of code coverage with our tests, indicating that basic performance is as expected and normal execution paths do not throw exceptions. The large amount of defined contracts indicates expressiveness of the code that we sought. Finally, the high number of Pex explorations and the number of tests that we promoted from them (only a small subset of the total number of generated Pex tests) indicates that the code is not throwing exceptions where we do not expect them.

Another, less formal metric for the success of the utility is experimental implementations of the NE classes. We set up a simple console application that included NE as a library. From here, we used the classes in a variety of scenarios, both normal and exception-generating and found that, in all cases, the NE classes behaved as expected. While this is not as definitive as the metrics, it does provide some confidence that they serve their purpose in actual usage scenarios.

A final consideration is the behavior of the NE classes in both debug and release modes. By setting our build configuration in the aforementioned console application, we were able to observe the behavior of the same code in both modes. In all cases, sensible defaults were provided in release mode whereas contract violation exceptions were thrown in debug mode. Because our classes do not throw exceptions in release mode, but allow them to be generated while debugging, we feel that we have successfully proved the concept of enforcing no-throw in production code while allowing the expressiveness of exceptions in debugging mode.

5. Future Work

For this project, our time-frame was necessarily somewhat limited, and of the time we did have, we spent a fair percentage of it acclimating ourselves with the tools necessary to accomplish the task. As such, we believe there is a good opportunity for future work on this project. Because of its highly modular nature, we believe that future work on this project is not only possible, but also relatively straightforward and with high potential for added value.

5.1 Adding more classes

The most obvious potential for future work is to expand the number of classes in NE by wrapping additional basic and library classes to provide additional no-throw functionality. There are two main vectors for expanding the functionality: breadth and depth. The breadth component would be achieved by finding other basic and abstract classes and implementing them, such as Float, DateTime, or ArrayList. This could be achieved easily enough by following the blueprint for existing NE classes, both in terms of implementation, contracts, and testing.

The depth component would involve implementing more complex classes that are composed of the sorts of basic and abstract classes that we have been wrapping. For example, with NEString in place, NEStringBuilder could be implemented, making use of NEString instead of String. There is a natural synergy involved in expanding the no-throw functionality, since new NE classes can make use of existing, proven, NE classes, knowing that they will not generate exceptions. As the NE suite grows, this consideration is vital as the number of potentially exception-generating situations could grow enormously.

5.2 Rooting Out More Corner Cases

As with any existing code base, there are naturally some corner cases and conditions under which exceptions may still be generated, even in spite of both the manual and automatically generated Pex tests to prevent them. In any but the simplest of implementations, there will always be bugs. As such, there is future work to be done in expanding the testing suite and proving more and more test cases.

This work could take on a variety of forms from creating more manual tests to creating more Pex explorations to exercising the code in the form of integration tests with actual use. And, since Code Contracts and Pex are both maturing, simply using newer version of them with added, refined or corrected features could also provide additional utility to NoException.

5.3 User-Specified Defaults

A final area in which future work could prove invaluable is the notion of user specified defaults. In its current form, NE classes have a hard-coded default value. For example, if a user of an NEArray of Integers overruns the bounds of the array, 0 is returned. In the case of an array of references, null is returned. It is conceivable and even likely that users would like to be able to specify that default instead of just using whatever the NE class provides. For instance, in cases where 0 or null is a valid value in the aforementioned array, an invalid access returning a valid value may confuse the developer when debugging.

This may be the most interesting piece of future work from an architectural standpoint as the decision as to how to implement the default-setting API is non-trivial. Setting the default at the instance level seems to make sense, but doing that every time an instance of a basic class is created might put back some of the code bloat we sought to eliminate, particularly if the user wanted different defaults at different times for the same instance. Having a static property set for all instances would eliminate some of this potential bloat but might not provide enough granularity and might become problematic in multi-threading or other complex scenarios. Generally speaking, the mechanism for providing defaults was beyond the scope of our work and is a non-trivial and interesting task.

6. Conclusion

In this paper, we have discussed an implementation of several useful basic types in C# that are guaranteed to never throw an exception during runtime. We explained the reasoning behind this sort of design, the way in which we decided to execute the design, and the tools we used to accomplish it. We found that Code Contracts are a viable and easily understood means of accomplishing this design paradigm by shifting the discovery of potential error conditions from runtime to compile time, though they are not without their faults. We also discovered that Pex is well suited for discovering unforeseen corner cases, and for automatically generating unit tests to exercise contracts, though it too has some problematic issues. Even if the problems with these tools were not simply caused by our relative inexperience with using them, we still found their advantages more than outweighed their disadvantages. Using them, we have delivered a useful library that is proven to fulfill its stated goal.

Works Cited

- [1] Wagner, Bill, Effective C#: 50 Specific Ways to Improve Your C#, Second Edition
- [2] <http://msdn.microsoft.com/en-us/vcsharp/aa336706>
- [3] <http://www.microsoft.com/visualstudio/en-us/products/2010-editions>
- [4] <http://research.microsoft.com/en-us/projects/contracts/>
- [5] <http://research.microsoft.com/en-us/projects/pex/>
- [6] <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/test-professional>
- [7] <http://www.developer.com/net/cplus/article.php/2197621>
- [8] http://java.sun.com/docs/books/jls/second_edition/html/exceptions.doc.html
- [9] <http://www.nunit.org/>
- [10] <http://www.mbunit.com/>
- [11] [http://msdn.microsoft.com/en-us/library/ms182409\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ms182409(v=vs.80).aspx)
- [12] Meyer, Bertrand <http://se.ethz.ch/~meyer/publications/computer/contract.pdf>
- [13] <http://www.albahari.com/valuevsreftypes.aspx>
- [14] Fowler, Martin <http://martinfowler.com/articles/injection.html>
- [15] [http://msdn.microsoft.com/en-us/library/wx0123s5\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/wx0123s5(v=vs.71).aspx)
- [16] <http://www.dotnet-guide.com/jit.html>
- [17] <http://research.microsoft.com/en-us/projects/pex/pextutorial.pdf>
- [18] <http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf>
- [19] <http://research.microsoft.com/en-us/projects/holes/>
- [20] Logozzo, <http://research.microsoft.com/pubs/141092/Main.pdf>
- [21] <http://msdn.microsoft.com/en-us/library/system.diagnostics.contracts.pureattribute.aspx>
- [22] Ciupa, Meyer, Oriol, Pretschner <ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/5xx/595.pdf>
- [23] [http://msdn.microsoft.com/en-us/library/wa80x488\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/wa80x488(v=vs.80).aspx)