nContract – Creating Configurable Run-Time Contract Verification for .NET Components

Westley Haggard

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Stephen Edwards, Chair
Dr. Osman Balci
Dr. Calvin Ribbens

March 21, 2005
Blacksburg, VA

Keywords:
Formal Specification, Component Specification,
Design-by-contract, .Net Components, Contract Verification

nContract – Creating Configurable Run-Time Contract Verification for .NET Components

Westley Haggard

Abstract

The use of third-party components is helpful while writing complex software systems, but it can be difficult to debug software that interacts with third-party components. To address this problem, a mechanism for determining if one is using the component correctly would be beneficial. Reading component documentation may or may not help, depending on its clarity and precision. A formally specified contract for that component would be better, and would also allow run-time contract verification via assertions. The client of the component could enable these assertions during development and debugging, and then disable them for the final production release to increase performance.

This thesis presents nContract, a tool that provides configurable run-time contract verification without requiring component recompilation or source code access. nContract allows component developers to formally specify .NET components using attributes. This contract information is retrieved from the compiled component's metadata and a subclass is generated for each formally specified type. All members of the component's interface are overridden and contract assertions are wrapped around calls to the base class. As long as the component client uses a factory to create instances of the component's types, the decision of whether or not to create assertion-checked or unchecked objects can be deferred until run-time.

Table of Contents

Table of Figures

Table of Tables

Chapter 1: Introduction

1.1 Introduction

Every developer at one point or another uses a third-party component. The reason for using third-party components is to try to keep software production costs to a minimum. There are many costs associated with developing a component in-house such as design costs, coding costs, testing costs and maintenance costs. Most of these costs can be eliminated by reusing a third-party component. However, using a third-party component has its own set of problems, such as figuring out how to use its interface and how to integrate it with other components.

Most third-party components come with documentation that informally describes the interface for the component. This informal description is helpful, but an informally written description may not define precisely what the component interface expects or what it produces. As a result, the component client may be unable to determine exactly how to use the interface. This in turn increases the chances of misusing the component and thereby increases the chances of introducing bugs.

To help with these problems, component developers can use a more formal approach to documenting components. A common way to formally specify a component is to use Bertrand Meyer's Design by Contract (DbC) approach [Meyer, 1992]. In DbC, one provides a "contract" between the component developer and the component client.

Using the DbC approach, component developers can precisely and unambiguously specify the component interface by providing pre- and postconditions for each method and invariant conditions for each class. Preconditions formally describe what the component expects to be true on entry to its methods and if these conditions are

not then the client is to blame. Postconditions formally describe what the component client can expect as a result from making a call to the component, and if these conditions are not then the component is to blame. Class invariants formally describe what must hold true about the state of a particular object after construction and before and after every method call.

Providing a contract for a component decreases the chances of component misuse and decreases the number of bugs clients make. Run-time contract verification, if available would help component clients to figure out if they are violating any preconditions more easily, and would assure them that the component is doing what it claims, further decreasing the number of bugs.

During development, DbC contracts can be verified in code by in-lining the conditions as assertions. The ability to verify contracts at run-time is a great asset during development, testing and debugging. In practice, DbC assertions are typically enabled during development and then, unfortunately, disabled during release to eliminate the run-time penalty of executing assertions in the final product.

In the case of component development, developers usually disable assertions in the release versions of their components. Since components are usually distributed in binary form, this eliminates any possibility for the component client to recompile the component with checks re-enabled. Because the developer disabled these assertions, component clients do not get any benefit from the run-time contract verification. On the other hand, if the assertions were enabled, then the component client would have to pay a performance penalty at run-time in the final product.

1.2 Problem

When DbC assertions can only be added or removed by recompilation, the component client has no opportunity to enable or disable them on third-part components. While the component developer could release the source code so that the client could recompile to enable or disable assertions, this option is not viable for commercial components. Instead this thesis explores the following question:

> *How can contract specifications be embedded in .NET components so that the user or purchaser of such a component can enable or disable run-time contract verification without requiring recompilation or access to the source code?*

The ability to enable or disable assertions for a component without source code access or recompilation will allow component developers to protect their intellectual property while allowing clients to take advantage of run-time contract verification features. Developers who provide this added value will also produce more reliable components that are more competitive in the marketplace.

1.3 Proposed Solution

A solution to the proposed problem requires a way of embedding contracts, packaging contract checks, and configuring contract checks. Figure 1.1 shows an overview of how these parts fit together. The component developer needs to provide a formal specification to embed in the binary component. Through the use of the nContract tool, the embedded contract then can be used to generate a subclass that packages the contract checks. The nContract tool also generates an XML configuration file that allows for contract checking behavior to be configured.

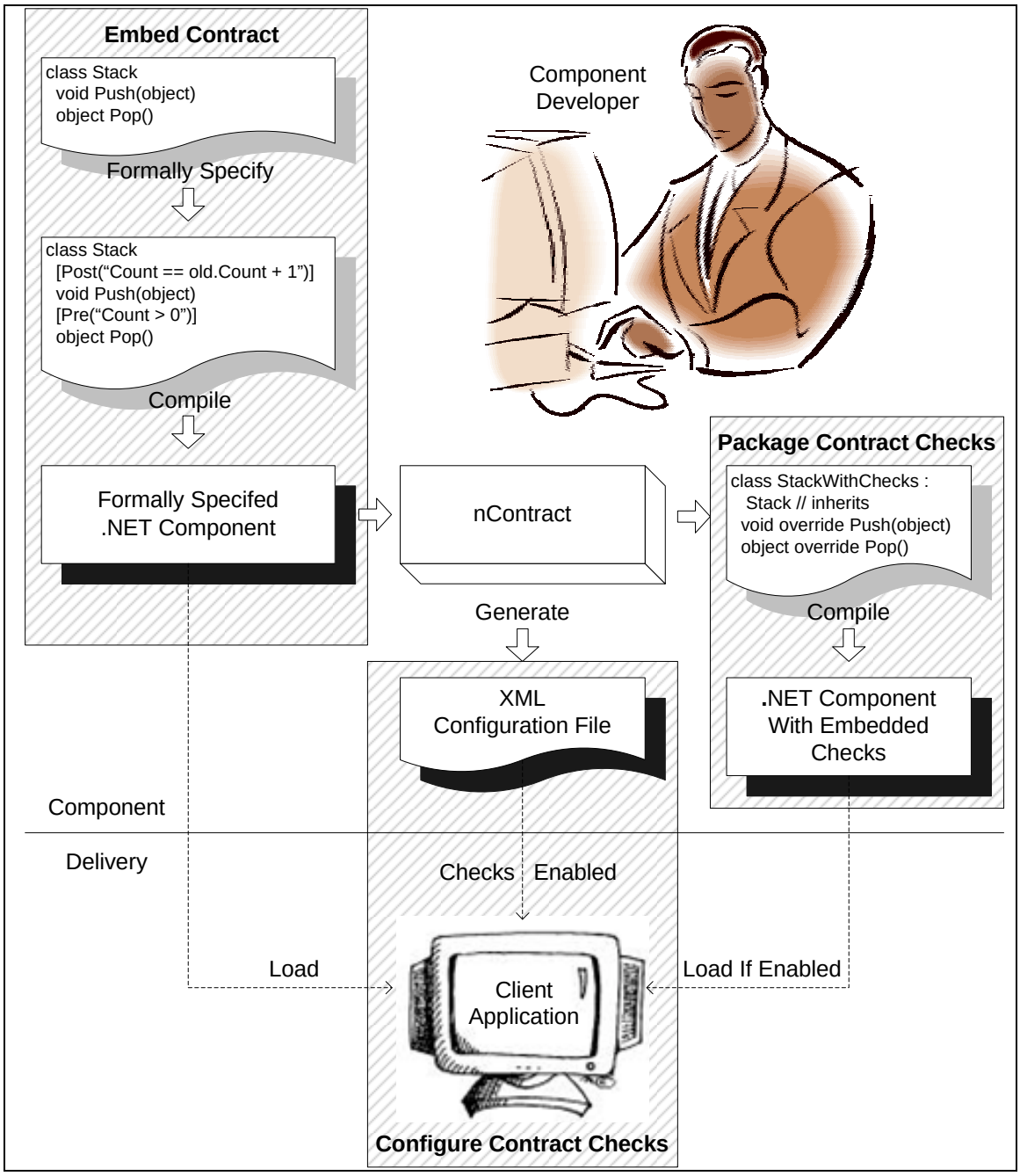

Figure 1.1: General overview of how nContract embeds, packages, and configures contract checks.

Through the use of attributes, .NET allows for custom data to be attached to individual methods and classes. These attributes can be used to store pre- and postconditions for methods and class invariants for classes. By using attributes to store this information, it becomes part of the metadata in the binary component and can be

retrieved easily programmatically. Thus, if the component developer formally specifies the class using attributes, the contract in turn is embedded into the binary component.

One method of packaging checks is to put all the assertions inline in the code. This does not allow for checks to be disabled at run-time without a performance penalty however. Therefore, run-time checking code should be separated from the original code. nContract used a subclass design to achieve this goal. All of the methods in the original class are overridden in the subclass and the assertion checks wrap around a call to the base class method. This design allows for the original class to remain unchanged and therefore have no performance penalty when checks are not needed. Since a subclass is used to contain checking code, a factory method can make a decision at run-time to create either an instance of the original class or an instance of the subclass with checks embedded.

The process of creating these subclasses can be completely automated because the checks are embedded into the binary component. The nContract tool uses the .NET reflection API to take a component with an embedded contract and generate another component which contains the subclasses with inline assertion checks.

For configuration, the nContract tool also generates an XML configuration file. The component developer then distributes three files: the original component, the component with checks, and the XML configuration file. Actually the component developer is only required to distribute the original component with the embedded contract because anyone at anytime can run nContract to generate the other two files. All the client has to do is use a factory method call to create instances of the formally specified classes. The checks for these classes can be enabled or disabled at run-time by

changing settings in the XML configuration file. This approach accomplishes the goal of being able to enable or disable run-time checks without requiring recompilation or source code access.

1.4 Contributions

The contribution of this research is to provide the consumers of .NET components a way to enable or disable run-time assertion checking without recompilation or access to the component source code. This thesis describes a method for embedding formal specifications into the metadata of binary .NET components. This in turn allows for the component developer to deliver not only the binary component but also deliver the formal specification for it in the same file which can add value over and beyond any written documentation provided [Edwards, 2001]. This thesis introduces a technique of packaging run-time checks in subclasses, which can be efficiently configured down to the individual method and check type level. When all the run-time checks are disabled there is very minimal performance penalty. Using this approach to provide run-time contract verification is a step forward in helping .NET component developers and clients develop more reliable software systems [Meyer, 1992].

1.5 Assessment

The assessment of this research focuses on two different aspects. First, a feature by feature breakdown and comparison with the most common tools and techniques used for run-time contract verification is presented. This comparison focuses primarily on how the contracts are provided or packaged and on how the run-time checks are configured. Second, run-time performance testing was conducted to assess the impact of this

approach. Experiments were conducted to test the running times of a component with checks included, both when enabled and disabled, and with checks omitted. This gives some insight of just how big the performance impact will be for production code.

1.6 Outline

Chapter 2 discusses related work and tools that deal with DbC and run-time contract verification. Chapter 3 presents how contracts are embedded in .NET components and how a component developer would use the nContract model. Chapter 4 presents details of how contract checking code is packaged into a subclass. Chapter 5 discusses how a client would use and configure components developed with the nContract model. Chapter 6 is a tutorial-style walkthrough of how to annotate code with contracts and how to use the nContract tool. Chapter 7 displays a comparison of the features of nContract in comparison with other DbC solutions. Chapter 8 presents the performance, comparing running times of code with and with out run-time contract verification embedded and/or enabled. Chapter 9 summarizes contributions and future work.

Chapter 2: Related Work

There have been many different approaches of implementing run-time contract verification for modern programming languages. They range from simple inline assertions to more complex method or class wrapper schemes. However, no matter how complex the scheme is there is some sort of assertion. One common problem that most solutions have is that they only allow enabling or disabling of run-time contract verification at compile time. Also a lot of the solutions only allow for assertions to be enabled or disabled at a global level as opposed to enabling on a pre-, postcondition or invariant level or on a per class or per method level.

2.1 Inline Assertions

An assertion is a boolean expression that is used to ensure a certain condition holds true at the point of the assertion execution. Therefore a failed assertion occurs when the boolean expression evaluates to false.

2.1.1 Simple Assertion Statements

Run-time enforcement at its simplest level can be done by using basic assert statements like the `assert` macro in C / C++. Basic assert statements can get the job done but they lack flexibility. Rosenblum takes basic assertions to the next step with Annotation Pre-Processor (APP) [Rosenblum, 1995], which allows for the checks to be selectively enabled or disabled at run-time through the use of severity levels. APP also allows for custom actions to be executed when an assertion fails.

With these approaches, recompilation is needed to enable or disable assertions. In the face of contracts, these approaches are harder to maintain because the developer has to hard code the assertions for the different types of checks. It can be difficult to determine which assertion statement relates to which part of the contract and it can be very annoying and error prone to update invariant assertions in every method.

2.1.2 Language Constructs

Eiffel, the landmark programming language for implementing DbC [Meyer, 1992], provides language support for formally specifying components through the use of keywords like `requires` for specifying preconditions and `ensures` for specifying postconditions. The Eiffel compiler embeds conditions inline as assertions so they can be enforced at runtime. Support is provided for including or excluding different categories of assertions from the generated code, via compiler switches.

The Eiffel programming language has also been targeted at the .NET platform so one can create .NET Components using Eiffel for .NET [Simon, Stapf, & Meyer, 2002]. Although Eiffel for .NET does not contain all the features of Eiffel, it does contain all the DbC features of Eiffel. Therefore one can provide contracts for .NET components using Eiffel for .NET the same way one would for standard Eiffel components.

Another language that supports DbC through language constructs is Spec# [Barnett, Leino, & Schulte, 2004]. Spec# is a research language being developed by Microsoft Research that is a superset of C#. Like Eiffel it provides language support for contracts via keywords like `requires` and `ensures`. The Spec# compiler embeds the different types of checks as inline assertions in the outputted assembly. Currently there is a compile time switch that will allow for these assertions to be excluded on a per

assembly basis. There is also a way to control whether invariant checks get generated on per class and per method level. At this time they do not support enabling or disabling these assertions at run-time although this has been discussed.

Although providing language support for contracts is more elegant, there are some limitations on the configuration front. There is no easy way to enable or disable the run-time contract verification without recompilation or access to source code.

2.1.3 Preprocessors

One common way of providing contract information is through the use of structured comments. IContract [Kramer, 1998] is a Java preprocessor that interprets its version of specialized comments and then inserts code in-line for the different types of checks. What makes IContract unique is an add-on tool IControl ["iControl"]. IControl is a GUI that allows for fine grain control of what pre-, postcondition or invariant checks should be enabled or disabled. It allows for enabling or disabling checks down to the method level. Although IContract is very configurable, it only works at the source code level and thus needs recompilation to enable or disable checks.

2.1.4 Metadata

Through the use of attributes in .NET, custom metadata can be embedded into a binary component. XC# ["eXtensible C#"], an extension to the C# compiler allows for the concept of compilation attributes. Compilation attributes allow for extensibility of the compiler such that extra IL code can be generated at compile time. XC# has created a requires and ensures compilation attribute that can be used to attach pre- and postconditions to methods. Through the use of these attributes and the use of XC# at

compile time the outputted IL is instrumented with assertions to enforce the contracts at run-time. This is, however, only a compile-time solution and only provides support for pre- and postconditions (although more support could potentially be implemented). There is also no scheme in place to allow assertions to be enabled or disabled at runtime. nContract also uses attributes in a similar manner to store contract information.

2.1.5 Dynamic Instrumentation

JContractor [Karaorman, Holzle, & Bruno, 1999] is a library approach to providing contracts for Java. It uses special naming conventions to add the different types of checks: for example, lets assume there is a method named `m (methodParameters)` then for the precondition checks there would be a method `protected boolean m_PreCondtion (methodParameters)` and similarly for other contract checks. JContractor uses Java's reflection API to inject checks inline dynamically at class load-time. There are two ways to use JContractor: either by using a custom class loader or by using a factory method. There isn't really any way to configure contract checks; they are either all on or all off. One advantage JContractor has over other solutions is that checks can be enabled or disabled at run-time without source code access by either using or not using the custom class loader. JContractor's factory pattern is similar to the nContract's technique except it generates the subclass and instruments it at run-time where as nContract doesn't generate or instrument anything at run-time.

2.2 Wrapper Based Approach

2.2.1 Method Wrappers

The Java Modeling Language (JML) is a specification language for Java that uses concepts from DbC to allow for formal specification of Java code using structured comments. JML has its own compiler, JMLC [Cheon, 2003] which works just like JAVAC with the only difference being that it adds assertion checking code to the Java bytecode it produces. The JMLC uses a method wrapper approach where a wrapper method is generated with the same name as the original method and the assertion checking code is added around a call to the now renamed original method. There is little configuration and recompilation is required to disable the run-time checks.

ContractJava [Findler, Latendresse, & Felleisen, 2001] [Findler & Felleisen, 2001] uses a similar method wrapper technique as JMLC and has similar drawbacks with little configuration and a required recompilation to disable run-time checks. However, ContractJava's primary focus is dealing with behavior sub-typing and assigning proper blame in the hierarchy chain if a contract violation occurs.

A similar technique is used in Handshake [Duncan & Hoelzle, 1998]. Contracts in Handshake are specified in a separate file with special syntax. If Handshake is enabled, then at file load time the Java bytecode is dynamically modified to add contract checks. Similarly to JMLC and ContractJava the original method is renamed and a new wrapper method with the original name is created and the checks are inserted around a call to the now renamed original method. There really is no configuration available other than commenting out or replacing assertions in the contract files. One advantage over the other two approaches is that run-time contract checking can be enabled or disabled globally,

without source code access, by enabling or disabling Handshake as the file load interceptor (done by setting an environment variable).

Even though the method wrapper approaches have some limitations they do add the benefit that the original method code is left unmodified and therefore there is less chance for introducing bugs than when directly injecting assertions into the original code.

2.2.2 Class Wrappers

An approach is to use a wrapper or decorator design pattern [Gamma, Helm, Johnson, & Vlissides, 1995]. Tan's Assertion Wrapper Design for Java [Tan & Edwards, 2003] uses the wrapper design pattern and uses JML to provide contracts. The basic idea is that the externally public interface is put into a Java interface and then there are two classes that implement that interface, the original class and a wrapper class with checks embedded. The wrapper class holds an instance of the original class to be consistent with the wrapper pattern. The pre-, postcondition and invariant checks in each of the wrapper methods surround a call through to the original method in the original class instance. A factory method is used to create instances and based on some configuration settings an object with run-time checks included or excluded can be created. There is also a more complex configuration scheme such that pre-, postcondition or invariant checks can be enabled or disabled on a per class level. Due to Java's reflection API most of this is done automatically by a custom class loader.

A similar design can be used in C++ [Edwards, Sitaraman, Weide, & Hollingsworth, 2004], however C++ does not have the luxury of a custom class loader. In C++, this pattern can be successfully implemented without a recompilation of the source

code but it does require selectively linking object files. So in order to enable checks one can link the object file that contains the wrapped class with checks.

Both of these approaches have drawbacks, but they do have features similar to nContract for Java components, namely configurability and the ability to enable or disable run-time checks without recompilation or source code access.

2.2.3 Proxy Classes

The .NET contract wizard [Arnout & Simon, 2001] is a tool that provides contracts for .NET components by creating Eiffel for .NET proxy classes. It works by reading in a .NET assembly and listing all the types and methods from the assembly and allows for pre-, postconditions and invariant checks to be entered as Eiffel expressions. These pre-, postconditions and invariants are added to the Eiffel proxy class using Eiffel's language constructs as discussed above. The tool then outputs another .NET assembly which contains all the Eiffel proxy classes. If the client wants to have the assertion checks on at run-time the client would use the Eiffel proxy instead of the original class. This is an interesting idea but it still has the same drawbacks as Eiffel.

2.3 Aspects

Aspect oriented programming (AOP) provides for an interesting approach to embedding contract checks. There are a number of possible ways to write an aspect that will instrument the code with assertions for contract checking. One such idea is discussed by Filippo Diotalevi [Diotalevi, 2004]. Another possibly more general solution would be to embed the contracts in the metadata and by doing so the aspects can inspect this metadata and generate assertions accordingly. The biggest problem with an AOP

approach is that most aspect weavers work at compile-time only. This would lead to a similar solution as some of the above where one can enable or disable the checks at compile-time but not run-time thus needing source code access.

Potentially a solution could be developed to work with a dynamic aspect weaver, where at runtime a configuration of some sort could be checked to determine whether or not a type should be woven or not at object instantiation. This again would work best if the contract information was stored in metadata and the aspects could work on that metadata. This was looked at as a potential solution for .NET, unfortunately there are no .NET dynamic aspect weavers that are mature enough to do what is needed, Loom.NET ["LOOM.NET"] was the closest found. Aspect weavers are much more mature in the Java world, although there is no surety about dynamic aspect weavers. With the introduction of metadata in Java version 1.5 ["JSR 175: A Metadata Facility for the Java Programming Language"], there is some potential for a possible Java based approach.

Although AOP doesn’t appear to be able to provide a full DbC solution (at least not at this time) there is still potential to use it for particular pieces of a solution. For example in doing this research, instead of forcing the client to use a factory method to create an object the client could use a standard new call. Then an aspect can be used to replace all new calls with factory method calls. This would prevent the client from having to do anything special and thus allow the client to use the component normally. This seems to be possible through the use of AspectJ ["AspectJ Project"] for Java but at this time there are not any aspect weavers for .NET that are capable of doing this.

Chapter 3: Embedding Contracts

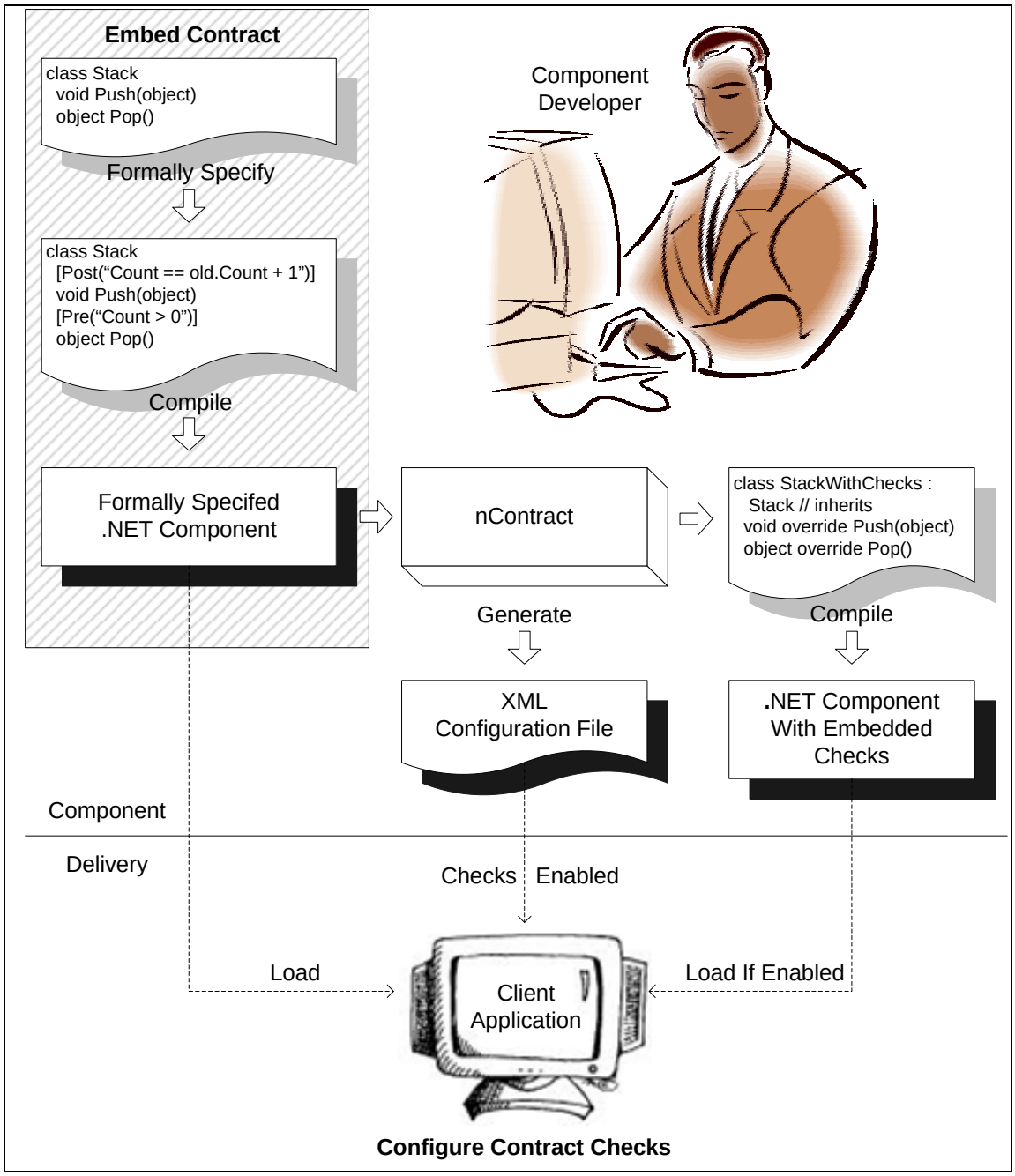

Figure 3.1: Overview of writing and embedding contract descriptions in .NET components.

In order to use the nContract model, a component developer would follow a fairly normal development path. This chapter discusses the changes needed for the nContract

approach to DbC checking such as providing a formal specification and a few design restrictions.

3.1 Design Goals

The following list of design goals guides all parts of the design and implementation presented in this research:

1. Contract information is embedded in the binary version of the component.
2. Run-time Checks can be added or removed without recompilation of either component or client code.
3. Checks can be enabled or disabled at the assembly, class or method level.
4. Checks can be enabled or disabled individually on a pre-, post-, exceptional postcondition and class invariant level.
5. Little or no performance penalty occurs if checks are disabled.
6. Custom actions can be performed on contract violations.

3.2 Providing Formal Specifications

3.2.1 .NET Attributes

.NET components are self-describing; all the information needed to describe the component is stored as metadata in the binary component. On top of this, .NET allows for developers to add custom metadata to the component by using custom attributes. Attributes are tags that can be added to code constructs such as classes, methods, parameters and other constructs where custom information needs stored. Figure 3.2 shows how attributes might be used in C# code.

```
[FormallySpecified]
public class Test {
  [Pre("i > 0")]
  public void DoWork(int i) { }
}
```

Figure 3.2: C# code example of a class attribute (FormallySpecified) and a method attribute (Pre).

.NET provides a reflection API that allows programmatic retrieval of component metadata at run-time. By using attributes to store DbC contract information, the contract can be retrieved from the binary component, eliminating any need for source code access. On top of helping with run-time contract enforcement, having the contract stored in the metadata allows other tools to benefit as well. More accurate and precise written documentation can be generated by pulling the contract information out of binary components.

In order to create attributes that store custom metadata, one creates a class that inherits from the `System.Attribute` class in the Base Class Library (BCL).

3.2.2 Contract Attributes

All the attributes that are used by this research to provide contract information can be found in the `ContractSpecification` component. This `ContractSpecification` component also contains the configuration classes and generic factory methods used to control run-time checking behavior. For a developer to use these features, they must include a reference to *`ContractSpecification.dll`*.

The contract attributes are used to store DbC contract information such as pre- and postconditions for methods. These conditions are expressed as a string parameter given to the attributes. These condition strings are extracted by the nContract tool and

inserted in a code template to generate run-time checks. Since run-time checks will be compiled as C# code these condition strings must be C# boolean expressions[1].

3.2.2.1 `FormallySpecified` Attribute

The `FormallySpecified` attribute is not really part of the contract. It is used to mark a specific class as being formally specified so tools like nContract can search all the types in an assembly and easily determine if they have embedded contracts.

```
[FormallySpecified]
public class Test
```

Figure 3.3: Sample use of the FormallySpecified attribute.

3.2.2.2 `ModelField` Attribute

The `ModelField` attribute is used for defining a public specification and is purely for specification purposes. This is used to prevent clients from tying to a particular implementation of a class. That way if the developer wants to modify that implementation the client code does not break. For example, if the client is typing to a property with the name `firstName` and the developer decides to change that to `fullName` then the client code breaks. To prevent problems like this, the developer can create a model field as a data abstraction and let the client tie to that field. That way if the internal name `firstName` changes to `fullName` the developer can just change the mapping expression without breaking any client code.

The `ModelField` attribute takes the type, name and mapping expression as parameters. There can be any number of `ModelField` attributes on a class. The `ModelField` name can be used in any of the condition expressions in the same class

[1] The expressions are provided as C# expressions because the templates used to generate the code are written in C#. Other .NET languages could be used for the expressions if the templates were changed to that particular language. Also, note that even though these attributes use C# expressions, they can still be used to specify other .NET languages, such as VB.NET.

(i.e. pre-, postcondition or invariant expressions). In fact, the model fields are recommended to be used instead of internal member fields. The mapping expression is a mapping from the representation model to the abstract model. It is expressed as a C# expression that can reference any internal data member of the class to create an instance of the model field type that represents the current value of the class.

```
[ModelField(typeof(string), "Name", "firstName")]
public class Test
```

Figure 3.4: Sample use of the ModelField attribute.

3.2.2.3 `Invariant` Attribute

The `Invariant` attribute is used to specify the class invariant for the given class. It takes only one parameter, a C# boolean expression. The expression can reference any `public` or `protected` members or any model field in the hierarchical chain. The members do include methods but it is highly recommended that only methods marked as pure are used (for more information about the use of methods in expression see Section 3.2.2.8). It is recommended that no specific internal fields are used in the expressions; model fields should be used instead. If an internal invariant is needed use a `RepresentationalInvariant` attribute instead.

```
[Invariant("Name.Length != 0")]
public class Test
```

Figure 3.5: Sample use of the Invariant attribute.

3.2.2.4 `RepresentationalInvariant` Attribute

A `RepresentationalInvariant` attribute is the same as an invariant attribute with an extra layer of abstraction. It should be used for private implementation details that need to be expressed as an invariant. This abstraction is primarily for information hiding so that if tools use the contracts then the

`RepresentationalInvariant` should be ignored by the client and should not be part of the public specification of the class. Other tools like nContract that add run-time verification of the contract can still use this invariant to add an additional assertion. If the invariant is using all `public` members or should be part of the public specification the standard `Invariant` attribute should be used instead.

```
[RepresentationalInvariant("firstName.Length != 0")]
public class Test
```

Figure 3.6: Sample use of the RepresentationalInvariant attribute.

3.2.2.5 `Pre` Attribute

The `Pre` attribute is used for specifying the precondition for a constructor, method or property. A constructor, method or property can only have one `pre` attribute. The precondition expression is a C# boolean expression. For a method or property it can reference any `public` or `protected` members, model fields or parameters. The precondition for a constructor is checked in a static factory method therefore it can only reference static or parameter variables; it cannot access members or model fields.

```
[Pre("newName != null")]
public void ChangeName(string newName)
```

Figure 3.7: Sample use of the Pre attribute.

3.2.2.6 `Post` Attribute

The `Post` attribute is used to specify the postcondition for a constructor, method or property. A constructor, method or property can only have one `Post` attribute. The postcondition expression is a C# boolean expression. For a constructor, method or property it can reference any `public` or `protected` members, model fields, old model fields, parameters or the return value. To access the return value the variable name `result` is used. A pre-state value is the value of the field before the method is executed.

For simplification purposes, the only pre-state values that can be accessed are the model field pre-state values. To access the pre-state value of a model field use the `old` keyword with the pattern `old.<modelfieldname>`.

```
[Post("result > 0 && old.Balance + amount == Balance")]
public int UpdateBalance(int amount)
```

Figure 3.8: Sample use of the Post attribute.

3.2.2.7 `ExceptionPost` Attribute

The `ExceptionalPost` attribute is used to specify any exceptional postconditions for a constructor, method or property. A constructor, method or property can have as many `ExceptionalPost` attributes as needed. The `ExceptionalPost` attribute expects two things, the type of exception expected and a C# boolean expression that must hold true if that exception is thrown. For a method or property it can reference any public or protected members, model fields, parameters or the exception thrown. The exceptional postcondition for a constructor can only reference static or parameter variables; it cannot access members or model fields. If access to the exception thrown is needed the variable name `excep` can be used.

```
[ExceptionalPost(typeof(ArgumentException),
      "newName == string.Empty")]
public int ChangeName(string newName)
```

Figure 3.9: Sample use of the ExceptionalPost attribute.

3.2.2.8 `Pure` Attribute

The `Pure` attribute is used to mark a method or property as having no side-effects. This is not verified by nContract; it is just a way for the developer to declare a method as being side-effect free. There is some potential for future work to create a way for this to

be verified. The motivation for the `Pure` attribute comes from the pure modifier in JML [Cheon, 2003].

If a method or property changes the state of the object it could potentially mask or introduce defects in the component. For that reason it is important to only use pure method or properties in the condition expressions of any of the attributes above.

Essentially there are three different types of methods: methods that are formally specified and marked as pure, methods that are formally specified and methods that are not formally specified (i.e. in a class that is not formally specified). There is run-time enforcement that ensures only pure formally specified methods are used within the conditions expressions (see Section 3.2.3 fore more details about run-time enforcement). However, there is no easy way to detect the use of methods that are not formally specified so they can be used but it is discouraged and up to the developer to use them correctly.

```
[Pure]
public string GetName()
```

Figure 3.10: Sample use of the Pure attribute.

3.2.3 Run-time Method Enforcement

The `ContractSpecification` component also includes support for enforcing that proper methods are called from within an assertion. The `MethodEnforcement` class is used to help make sure that only pure methods are called from within pre-, postcondition and invariant checks. The way the class works is that there are two static methods `EnterChecks` and `ExitChecks` that both take a boolean that represents whether or not the caller is a pure method. When the checks are being executed a call to `EnterChecks` is made, in the overridden methods that have the

checks, to globally mark that checks are being executed. Now if `EnterChecks` is called again with a method that is not pure, before the original method calls `ExitChecks`, then a `NonPureMethodException` is thrown. This allows for some run-time protection of non-pure method use in the pre-, postcondition and invariant checks. This however is not fool proof because as stated in Section 3.2.2.8 there are other methods that could be called that are not formally specified and there is no easy way to detect this at run-time. Perhaps a better way to do this would be to do it statically at compile-time but that is beyond the scope of this thesis. For code examples of how `EnterChecks` and `ExitChecks` are used, see the sample templates in Section 4.3.1.

3.3 Design Restrictions

By using subclasses to package run-time checking code, as detailed in Chapter 4, there are a number of design restrictions that a component developer needs to keep in mind:

1) In order to inherit from a class, the class cannot be `private` nor can it be `sealed` or `static`.
2) All the constructors that are to be checked at run-time must be `public` or `protected`. This prevents run-time checking of any class design that requires a `private` constructor. All constructors should be marked as `protected`, forcing the client to use a factory method to create instances.
3) All methods or properties that are to be checked at run-time must be `public` or `protected` and they must also be overridable. To be overridable the method or property must either have a `virtual` or `override` keyword associated with it.

`Static` and `private` methods or properties are not checked because they cannot be overridden.

4) Fields are not checked. To get around this limitation one could create a `public` or `protected virtual` property that references the field in question. If a field is used in any of the contract conditions it must be at least `protected` otherwise it will not be accessible to the subclass.
5) The `new` operator cannot be used to create objects; a factory method must be used instead (This restriction applies to the use of the class not necessarily to the design)

3.4 Factory Methods

The final design restriction affects client code: the new operator should not be used to create objects with run-time checks; a factory method must be used instead. Using a factory method allows the decision to create an instance of the class with or without embedded checks to be made at run-time. To ensure that the client uses a fatory method instead of the new operator it is recommended that the component developer make all the constructors protected. The component developer has a few choices to make about the factory method used to create instances of a formally specified class. One alternative is to force the client to use a generic factory method provided in the `ContractSpecification` component. As can be seen from Figure 3.11, the factory method creates a checking subclass instance if the checks are enabled for a particular type, and creates an ordinary instance if the checks are not needed.

```
public static T Create(params object[] args) {
  if (ChecksEnabled())
    return CreateChecksEnabledInstance(args);

  return (T)System.Activator.CreateInstance(typeof(T),
    BindingFlags.Public | BindingFlags.NonPublic |
    BindingFlags.Instance | BindingFlags.CreateInstance,
    null, args, null);
}
```

Figure 3.11: Completely generic factory method provided by the ContractSpecification component.

This method requires the least amount of work from the component developer but may require a larger learning curve for the client. One of the problems with making the factory method completely generic is that it uses the dynamic method `Activator.CreateInstance` to create the objects. Creating objects dynamically tends to be slower than creating them using the `new` operator, see Figure 8.2 for running times. Another downfall is that the parameters are not strongly typed so the compiler cannot catch potential type errors with the parameters being passed. Figure 3.12 below shows how a client might actually use the generic factory method to create an instance of a class name `ClassName`.

```
ContractSpecification.Factory<ClassName>.Create(p1, p2);
```

Figure 3.12: Sample use of the completely generic factory method.

Another approach that the component developer may use eliminates the problem of strongly typed parameters: create a custom factory method for each constructor in the class.

```
public static ClassName Create(PT1 p1, PT1 p1) {
  return Factory<ClassName>.Create(p1, p2);
}
```

Figure 3.13: Sample custom factory method with strongly type parameters.

As in Figure 3.13, these custom factory methods can still use the generic factory method to create the objects but provide a strongly typed parameter profile. However,

there is still a potential slow down in object creation from `Activator.CreateInstance`. This can also be improved, when checks are disabled, by customizing the factory methods to mimic the generic factory method but using the `new` operator instead of the generic `Activator.CreateInstance`, as in Figure 3.14.

```
public static ClassName Create(PT1 p1, PT1 p1) {
  if (Factory<ClassName>.ChecksEnabled())
    return Factory<ClassName>.CreateChecksEnabledInstance(p1, p2);
  return new ClassName(p1, p2);
}
```

Figure 3.14: Sample custom factory method using the new operator instead of CreateInstance.

Figure 3.15 shows how the custom factory method would be used to create an instance of the object. Now parameters with incorrect types result in compiler errors.

```
ClassName.Create(p1, p2);
```

Figure 3.15: Sample use of a custom factory method.

The `CreateChecksEnabledInstance` method uses reflection to dynamically call a factory method to create an instance of the subclass with checks. As seen from Section 8.2 this performs slightly slower than the dynamic `CreateInstance`. Since both the generic and custom factories use this same method they both run at the same rate. However, there is an approach which a component developer could employ to dramatically increase this performance for the custom factory. If a component developer creates a custom delegate and a static field of that delegate type, as in Figure 3.16, for each constructor parameter profile then they could use a delegate to create the checks enabled instance.

```
public delegate CharBuffer CreateDelegate1();
public static CreateDelegate1 Create1 = null;
public static CharBuffer Create()
{
  if (Factory<CharBuffer>.ChecksEnabled())
    return Create1();
  return new CharBuffer();
}
```

Figure 3.16: Custom delegate and static field used to create object instance with embedded checks.

Now what happens is when the configuration is loaded for this class the `Create1` field is initialized to point to the appropriate factory call in the generated subclass. In order for this to work properly the static field must start with `Create` and be followed by a number, starting with 1 for the first then incrementing by 1 for each additional one. This is more efficient because the binding only happens once as opposed to every time it is executed. This approach causes more work for the component developer but it gives them the option to increase the performance of object creation in both cases by using the custom factory method.

Chapter 4: Packaging Contract Checks

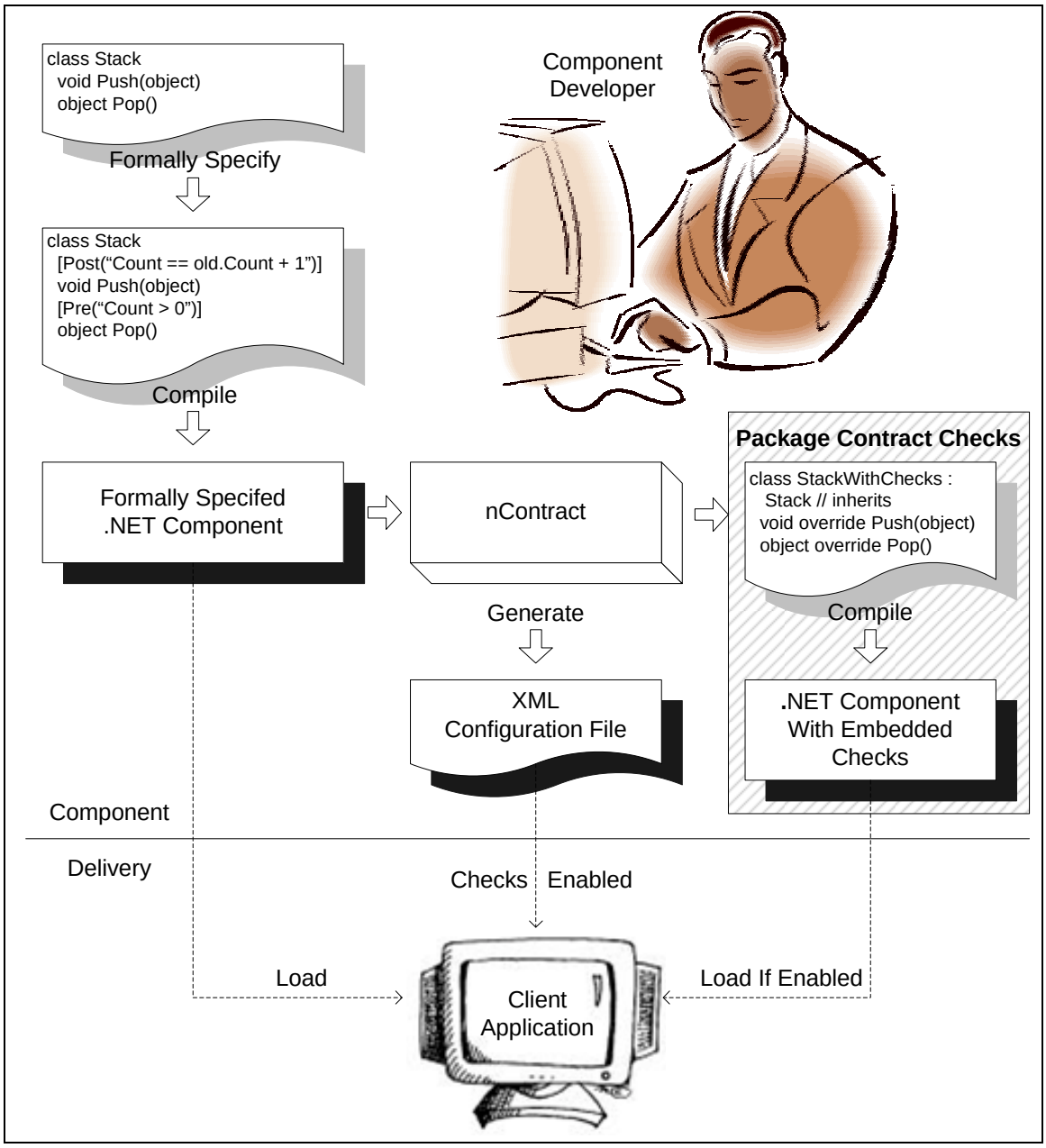

Figure 4.1: Overview of packaging the checking code in subclasses and then creating a .NET component with all these subclasses.

The nContract tool automatically generates a subclass that contains assertion checks to enforce contracts. This chapter discusses the subclass design and how

nContract uses templates to generate the subclasses, which are then compiled into a separate .NET component.

4.1 Packaging Checks in Subclasses

There is a possibility of inadvertently introducing bugs into the original code when tools instrument the checking code inline in the original code. The safest way to ensure no bugs are inadvertently introduced in the original code is to leave it unchanged. If you need to leave the original code unchanged then there needs to be some method of separating the checking code from the original code. Another reason to separate the checking code is to prevent the performance penalty of executing the checking code all the time when checks are not needed.

If the checks are separated from the original code, then there are only two possible ways to have the checks wrap around a call to the original method. One way is to have an instance of the original class as a member field of a “wrapper” class. This can be implemented using a wrapper design pattern [Gamma et al., 1995] as shown by Tan and Edwards [Tan & Edwards, 2003]. One of the problems with this approach is that, in order to create interchangeable objects a common ancestor in the inheritance tree is needed. Tan chose to use a common interface that both the original class and the wrapper class implement. This design has some benefits that are discussed later in this section but it is also more complex and requires more work.

The alternative approach is to put the checks in a subclass. This allows the objects to be interchangeable. This approach is less complex because instead of dealing with two classes and an interface it only deals with the original class and a subclass.

The basic idea behind the subclass design is to override all the `public` and `protected` methods from the original class and insert preconditions, postcondition and invariant checks around a delegating call to the inherited implementation as illustrated in Figure 4.2. A similar approach can be taken for .NET properties because they are just simplified syntax for a pair of get and set methods.

```
protected override void MethodName(int parameter1) {
  // Check Class Invariant
  // Check Preconditions
  base.MethodName(parameter1);
  // Check Postcondtions
  // Check Class Invariant
}
```

Figure 4.2: Simple example of how the checks wrap a call to the base method in the subclass design approach.

Section 3.3 lists some restrictions from this subclass approach. These restrictions are re-listed here with a comparison to Tan's interface approach.

1) In order to inherit from a class, the class cannot be `private`, `sealed` or `static`. This makes some class designs incompatible with this design but the same limitations are present with the interface approach.
2) All the constructors that are to be checked at run-time must be `public` or `protected`. This prevents checking any `private` constructors but this same limitation exists with the interface based approach. Component developers should declare constructors `protected` to force the client to use factory methods to create instances.
3) All methods or properties that are to be checked at run-time must be `public` or `protected` and they must also be overridable. To be overridable the method or property must either have a `virtual` or `override` keyword associated with it. `Static` and `private` methods or properties are not checked because they

cannot be overridden. The interface based approached only works with `public` methods.

4) Fields are not checked. To get around this limitation one could create a `public` or `protected virtual` property that references the field in question. Fields are also a problem for Tan's interface approach, but with .NET one could declare properties in an interface that can be used like fields.
5) The `new` operator cannot be used to create objects; a factory method must be used instead. The same thing is true for the interface approach. (This restriction applies to the use of the class not necessarily to the design)

These limitations can cause some class designs to be altered or unusable but in all with the subclass approach all the restrictions are similar or less restrictive than Tan's interface approach.

4.2 Inheritance of Contract Specifications

A class using this subclass design inherits preconditions, postconditions and invariant conditions from its super-classes and any interfaces that it implements. It supports the notion of behavioral subtyping as stated in the Liskov Substitutability Principle (LSP) [Liskov & Wing, 1994]. A subtype must require no more than its supertype therefore the preconditions must be disjoined. A subtype must promise no less than its supertype therefore the postconditions should be conjoined. The invariants should also be conjoined. In C# terms, this translates into logically ORing all the inherited preconditions and logically ANDing all the inherited postconditions and invariants from all the super-classes and interfaces.

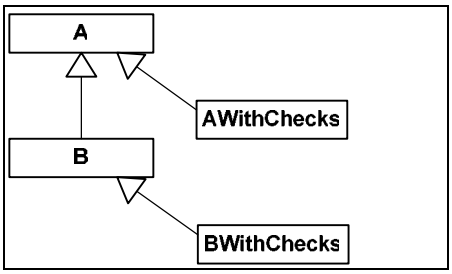

Figure 4.3: Simple inheritance hierarchy showing the subclass design.

By using this subclass approach there is potential for code duplication in the checking code. For example, in Figure 4.3 suppose there are formally specified methods in class `A` that are not overridden in class `B`. These methods are overridden in the both `AWithChecks` and `BWithChecks`, and they also contain exactly the same code. Even if a method from `A` is overridden in `B`, then that same method is overridden in `BWithChecks` and it checks all conditions in `A` as well as the conditions in `B`, thus still producing some overlap and duplication. The biggest problem with code duplication is maintaining the code. Since the checking subclasses are generated automatically, there is not significant problem maintaining the code. If something changes, the class and any subclasses can simply be regenerated.

Another approach that could be taken is to have parallel inheritance chains in the class and the class with the checks. Due to the lack of multiple inheritance in .NET languages and Java, this approach can only be used through the use of an interface. Figure 4.4 shows a simple example of Tan's [Tan & Edwards, 2003] parallel inheritance chains for original and wrapper classes. This removes the code duplication and thus allows for contracts to be updated in super-classes without having to update the wrappers for all the subclasses.

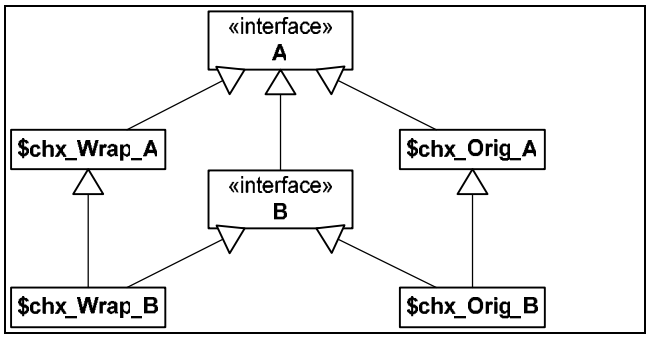

Figure 4.4: Sample of Tan's parallel inheritance chains for original and wrapper classes

This, however, is more complex as discussed in previous sections. The subclass approach taken here has the benefit of being simpler and also completely automated so the duplication problem is manageable. Duplication would be a much bigger problem if the subclasses were being maintained by hand.

4.3 Template Model

One of the unique features of nContract is that it allows for a custom template to be used for generating the subclass that contains the checking code. CodeSmith [Smith, 2002], a freely available code generation tool for .NET, was used to generate subclass code from a template. Therefore the template syntax is from CodeSmith, which is actually very similar to ASP.NET syntax.

4.3.1 Subclass Template

The code template used for generating the subclass consists of four parts: the constructors, methods, properties and model fields. For the purposes of discussion, the template code is only a sample and much of the extraneous detail is removed to make the template pieces shorter and more understandable.

To get an idea of how the run-time method enforcement from Section 3.2.3 comes into play, calls to `EnterChecks` and `ExitChecks` are included in the sample

templates but they are not discussed. For the purpose of these sample templates, the pre-, post-, exceptional postconditions and invariants are simply labeled as such and in reality they would be boolean expressions that are a combination of all such conditions from the class, any base classes and any interfaces. As stated previously the preconditions are logically ORed and the postconditions and invariants are logically ANDed.

Another thing common among the sample templates is the use of unique names for access to the method configurations. This unique name is generated by concatenating the method name with the types of all the parameters, each separated by an underscore character. If there are no parameters then a type of void is appended. This approach is guaranteed to generate a unique name for each constructor, method and property because no two of them can have the same combination of name and parameter list.

4.3.1.1 Constructor Template

nContract generates two items for every constructor in a formally specified class: a constructor and a factory method. A static factory method called `Create` is generated so that the preconditions can be checked before the object is created and so the exceptional postconditions can be checked. The `CreateChecksEnabledInstance` method in the generic factory class actually makes a dynamic call to this `Create` method to create instances of the class with checks instead of dynamically creating the object. Figure 4.5 shows a sample `Create` method. If the precondition is enabled it is checked. The object creation takes place in a `try/catch` block so that if an exception is thrown in the creation then the exceptional postcondition can be checked. As discussed earlier the precondition and exception postcondition for a constructor can access only static members and parameters, this is due to them being checked in this static factory

method. Also keep in mind that it doesn't make sense to check the invariant before the object is created.

```
public static ClassName Create(PT1 p1, PT2 p2)
{
  if(!config.ctor_PT1_PT2.PreDisabled) {
    MethodEnforcement.EnterChecks(IsPure);
    config.CheckPrecondition(Precondition);
    MethodEnforcement.ExitChecks(IsPure);
  }
  ClassNameWithChecks newObject = null;
  try {
    newObject = new ClassNameWithChecks(p1, p2);
  }
  catch(Exception ex) {
    if(!config.ctor_PT1_PT2.ExceptionalPostDisabled) {
      MethodEnforcement.EnterChecks(IsPure);
      if(ex is ExceptionType) {
        ExceptionType excep = ex as ExceptionType;
        config.CheckExceptionalPostcondition
          (ExceptionalPostcondition);
      }
      MethodEnforcement.ExitChecks(IsPure);
    }
    throw;
  }
  return newObject;
}
```

Figure 4.5: Simplified sample template for the generated Create factory method.

Figure 4.6 shows a sample generated constructor. It makes a call to the base constructor which is the constructor for the formally specified type. It also checks the postcondition and invariant. These checks are in the constructor itself instead of in the factory method so they can actually reference members of the class.

```
public ClassNameWithChecks(PT1 p1, PT2 p2)
  : base(p1, p2)
{
  MethodEnforcement.EnterChecks(IsPure);

  if(!config.ctor_PT1_PT2.PostDisabled) {
    MethodEnforcement.EnterChecks(IsPure);
    config.CheckPostcondition(Postcondition);
    MethodEnforcement.ExitChecks(IsPure);
  }
  if(!config.ctor_PT1_PT2.InvariantDisabled) {
    MethodEnforcement.EnterChecks(IsPure);
    config.CheckExitInvariant(InvariantCondition);
    MethodEnforcement.ExitChecks(IsPure);
  }
}
```

Figure 4.6: Simplified sample template for a constructor.

4.3.1.2 Method Template

The method template is similar to the outline in Figure 4.2 but with more detail. One such detail is storing the pre-state of the model fields. As shown in Figure 4.7, in order to store the pre-state values of the model fields being referenced in the postcondition, local variables are used and given the value of the model field before the method is executed. As in the generated factory method in Figure 4.5, a `try/catch` block is used to catch any exceptions thrown from the call to the base method. If the method succeeds then the normal postcondition is checked but if an exception is thrown then in the `catch` block the exceptional postconditions are checked and then the exception is re-thrown. Note that in .NET one uses the keyword `throw` to re-throw the same exception; if `throw ex` is used in this example then all the original stack trace information is replaced with the current stack trace. The invariant is checked in a `finally` block to ensure that it is always checked no matter how the method is terminated.

```
public override ReturnType MethodName(PT1 p1, PT2 p2) {
  if (!config.MethodName_PT1_PT2.PostDisabled) {
    // Generate a local variable to hold the pre-state value
    // of any model field used in the postcondition
    ModelFieldType1 oldModelField1 = ModelField1;
    //...
  }
  if (!config.MethodName_PT1_PT2.InvariantDisabled) {
   MethodEnforcement.EnterChecks(IsPure);
   config.CheckEntryInvariant(InvariantCondition);
   MethodEnforcement.ExitChecks(IsPure);
  }
  if (!config.MethodName_PT1_PT2.PreDisabled) {
    MethodEnforcement.EnterChecks(IsPure);
    config.CheckPrecondition(Precondition);
    MethodEnforcement.ExitChecks(IsPure);
  }
  ReturnType result;
  try
  {
    result = base.MethodName(p1, p2);
    if (!config.MethodName_PT1_PT2.PostDisabled) {
      MethodEnforcement.EnterChecks(IsPure);
      config.CheckPostcondition(Postcondition);
      MethodEnforcement.ExitChecks(IsPure);
    }
  }
  catch (Exception ex)
  {
    if (!config.MethodName_PT1_PT2.ExceptionalPostDisabled) {
      MethodEnforcement.EnterChecks(IsPure);
      // Generate a block like this for each
      // excpetional postcondition
      if (ex is ExceptionType)
      {
        ExcpetionType excep = ex as ExceptionType;
        config.CheckExceptionalPostcondition
          (ExceptionalPostcondtion);
      }
      MethodEnforcement.ExitChecks(IsPure);
    }
    throw;
  }
  finally
  {
    if (!config.MethodName_PT1_PT2.InvariantDisabled) {
      MethodEnforcement.EnterChecks(IsPure);
      config.CheckExitInvariant(InvariantCondition);
      MethodEnforcement.ExitChecks(IsPure);
    }
  }
  return result;
}
```

Figure 4.7: Simplified sample template for a method.

4.3.1.3 Property Template

A property in .NET is simply syntactic sugar for a pair of get and set methods. As shown in Figure 4.8, the property template is just a get part with a method template and a set part with a method template. There are only a couple of small changes, such as instead of calling the base method it calls the base property which has slightly different syntax.

```
public override ReturnType PropertyName {

  get {
    // Method template
  }
  set {
    // Method template
  }
}
```

Figure 4.8: Simplified sample template for a property.

4.3.1.4 Model Field Template

For every model field in the formally specified class, any base classes or any interface implemented, nContract generates a property. The property only has a get method and it returns the mapping expression for the model field. Since model fields are referenced just like fields in the postconditions and properties can be accessed like fields then there is no work to be done to translate the postconditions.

```
public ModelFieldType ModelFieldName
{
  get {
    return ModelFieldMappingExpression;
  }
}
```

Figure 4.9: Sample template for a model field.

4.4 Using the nContract Tool

As seen in Figure 1.1, the nContract tool takes as input a .NET component. The nContract tool works on the formally specified classes contained in this .NET component. It produces another .NET component that contains all the subclasses with the embedded

checks. It also generates an XML configuration file that is used to configure the checks of the outputted component all the way down to the method level.

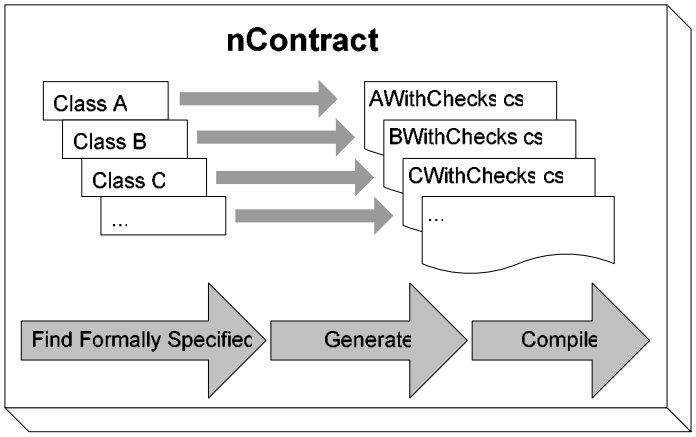

Figure 4.10: Overview of how the nContract tool works.

The tool starts by finding all the classes that are marked as formally specified. For each formally specified class, a template is used to generate a `.cs` file that contains the subclass and the configuration class associated with the formally specified class. For example, if class `A` is formally specified, then a `.cs` file called *`AWithChecks.cs`* is generated that contains the subclass of `A` named `AWithChecks` and a configuration class named `AConfiguration`. Finally, as shown in Figure 4.10, all these `.cs` files are compiled to create the new .NET component. After all the configuration classes are generated, they are used to output an XML configuration file that mirrors the settings in these configuration classes.

4.4.1 Requirements for Running nContract

The nContract tool is written in C# version 2.0 targeted at the .NET Framework. Therefore .Net Framework version 2.0 beta 1 or greater needs to be installed in order to run the nContract tool.

The following is a list of files that are needed to run the nContract tool.

1. *nContract.exe* – This is the actual nContract command line tool.
2. *nContractTemplate.cst* – Contains the default subclass and configuration code template. The *.cst* extension is used for CodeSmith template files. The class templates can be customized by modifying this file.
3. *ContractSpecification.dll* – Contains the supporting classes that are used to implement the nContract design and tool.
4. *CodeSmith.Engine.dll* – This is the CodeSmith [Smith, 2002] engine that is used to generate the code from the template.
5. The last file is the input assembly that contains the formally specified types which the component developer creates.

4.4.2 Running nContract

```
nContract <Input assembly>
```

Figure 4.11: Command line usage for the nContract tool.

The command in Figure 4.11 can be executed in the directory containing all the files listed above. Where *<Input assembly>* is the name of the component which will usually be a *.dll* file. It is recommended that the *.dll* file be copied into the same directory as the tool but it is not required. If the *.dll* is not in the same directory then the location must also be provided.

The tool produces three things. First, it outputs a *.cs* file for every formally specified type in the input assembly. This .cs file has the name <formally specified class>WithChecks.cs and contains the subclass and configuration class that are generated by the template. Second it compiles all those *.cs* files into

a *.dll* file which is outputted with the name *<input assembly name>WithChecks.dll*. Third, it generates a configuration file with the name *<input assembly name>WithChecksConfig.xml*. This configuration file is created by enumerating all the newly generated configuration classes and creating a default instance and serializing them as XML into the configuration file. The configuration file allows for the enabling or disabling of precondition, postcondition and invariant checks down to the method level.

4.4.3 Potential Problems

IF the nContract tool throws an exception during execution, the most likely problem is that one of the required files is missing or the version of the .NET Framework installed is not at least version 2.0 beta 1.

When the command runs but gives an unsuccessful message, the most common problem would be compiler errors in one or more of the generated classes. The compilation errors should be printed as well. One can open up the associated *.cs* file to the line that the error message is pointing to and examine the problem. Most likely it is a syntax problem with one of the assertions. For this case the expression in the appropriate condition attribute needs to be changed to fix the problem.

Chapter 5: Configuring Contract Checks

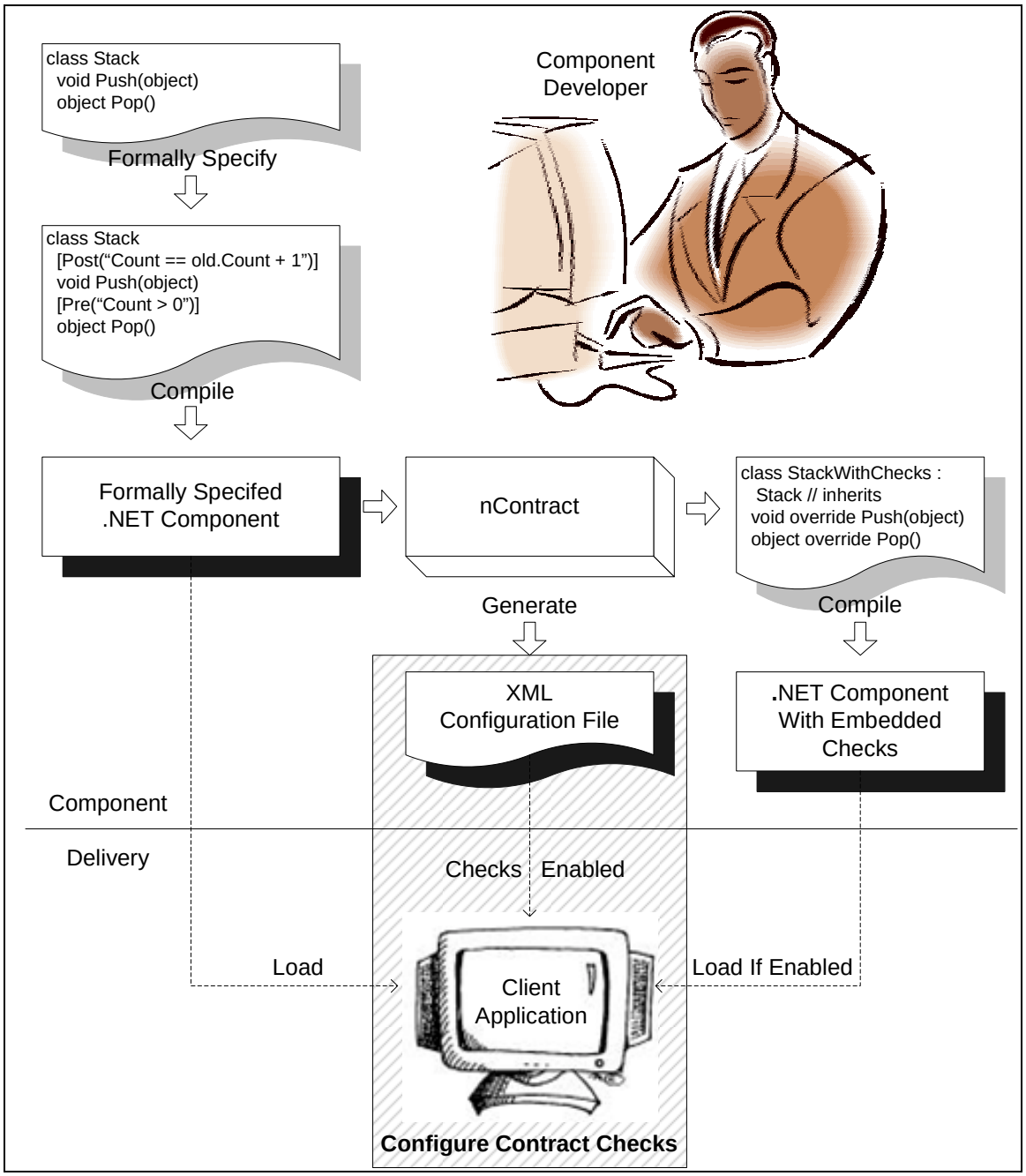

Figure 5.1: Overview of how to configure which contract checking code is executed.

The component developer delivers three files: the original .NET component, the .NET component with checks embedded and an XML configuration file. This chapter

discusses how a client uses this .NET component and also how they can configure the checks using the XML configuration file.

5.1 Configuring Checks

To provide appropriate flexibility to client s, we must give them the ability to individually configure pre-, post-, and exceptional postcondition and invariant checks at the assembly, class and method level. The first approach one might think of to solve this problem is to use a dictionary to store all the configuration settings for every method, with the method name as the key. Another possibility could be to use a dictionary of dictionaries with the first being keyed by the class name and the second by the method name. Although these may be straight forward and easy to implement, they cause a significant performance penalty in all but the simplest components due to the dictionary lookups added to every method call.

A more reasonable approach would be to keep all the settings in a configuration class and store an instance to that class in the subclass with the checks. This gives direct access to the class configuration, but what about the method configurations? It would be beneficial to have direct access to the method configurations as well. This could be accomplished if a custom configuration class is used to store the settings for a particular class and contains a member field for every method that holds its configuration.

In order to get the fine grain level of configuration stated in the design goals, nContract does use a separate custom configuration class that contains class and method configuration settings. The subclass with the checks can then store a reference to that class as illustrated in Figure 5.2. Since all the configurations for a particular class are the same there should only be a single instance of each configuration class. Therefore, a

singleton pattern can be used for the configuration classes. Also, most of the settings for different classes are the same the only difference being the method configurations so the common settings are pulled out into a base class named `ClassConfiguration`.

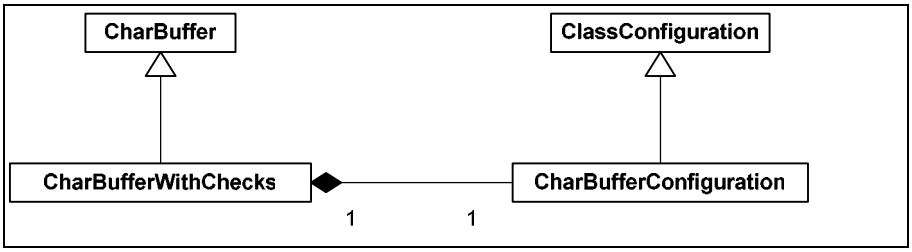

Figure 5.2: Simple class diagram that shows the subclass design in relationship to the configuration classes.

5.2 Configuration Classes

One of the contributions of this research is the configurability of the run-time contract checks. This section goes into the details of the `AssemblyConfiguration` and `ClassConfiguration` which play a strong supporting role in the configuration of these components.

5.2.1 Assembly Configuration

The configuration interface starts with the static `AssemblyConfiguation` class. This class keeps track and loads assemblies dynamically as needed. It also stores single instances of all the individual class configurations. Table 5.1 lists and describes the fields and methods of interest in `AssemblyConfiguration`.

Fields	
`loadedAssemblies`	This is a static dictionary used to map a loaded assembly to its associated assembly which has the checks included.
`loadedConfigurations`	This is a static dictionary used to map a full type name to the singleton instance of the class configuration for that type.
Methods	
`GetConfig`	This is a public method that takes a type as a

	parameter and then returns the associated class configuration for that type. It starts by checking to see if the assembly with checks is loaded for that assembly. If it is not loaded then it loads it along with the configuration for that assembly. If no associated assembly is found on disk the mapping in `loadedConfigurations` is set to null and null is returned instead of the instance of the class configuration.
`LoadAssembly`	This private method takes an assembly as input and then takes the name and location of that assembly and appends the text "WithChecks" to it and sees if an assembly with that name exists and if so loads it dynamically. For example if the assembly name is Example.dll then this method looks for an assembly with the name ExampleWithChecks.dll in the same location and if it is there loads it and sets up the mapping in `loadedAssemblies`. If the assembly is not located the `loadedAssemblies` mapping maps to null. If the assembly is loaded successfully then the `LoadConfig` method is called to load the configuration file associated with this assembly.
`LoadConfig`	This private method takes an assembly as input and then takes the name and location of that assembly and appends the text "Config.xml" to it and checks to see if a file exists with that name and if so it loads it. Whether or not that file exists the configuration settings are setup for all the configuration classes that exist in the passed in assembly. By default if no configuration file is found then the checks are all disabled for all the types.

Table 5.1: Descriptions of all the members in the AssemblyConfiguration class.

5.2.2 Class Configuration

The `ClassConfiguration` class is an abstract base class to be used as the base for all the custom class configuration classes. It contains settings that are general and common to all classes. It also contains the assertion methods. Table 5.2 lists and describes the public members that are of interest in `ClassConfiguration`.

Properties	
`TypeName`	This is a simple string that holds the name of the type that this class configuration is associated with. It is used as the key to setup and lookup configurations in the `AssemblyConfiguration` class.
`Enabled`	This is a boolean value that determines whether or not checks are enabled for the type that this class is representing.
`UseDefaultTraceAssert`	This is a boolean value that determines whether or not to use the standard BCL `Trace.Assert` for failed assertions.
`AssemblyWithChecks`	This is a reference to the assembly that contains the checks associated with this class. It is set when the `LoadAssembly` method is called and all the class configurations are created.
`TypeWithChecks`	This is a type object that provides quick and easy access to the type information for the subclass with checks. It is looked up via the string name once when the `AssemblyWithChecks` property is set and then stored to save that overhead.
Methods	
`Assert`	This method takes three parameters a boolean value, a string that represents which type of assertion it is and a string that represents the boolean expression. Nothing happens if the boolean value is true but if it is false then the `OnAssert` event is fired and if `UseDefaultTraceAssert` is true then `Trace.Assert` is called. Both `OnAssert` and `Trace.Assert` are passed a combined string of the condition type and boolean expression that failed.
`CheckEntryInvariant`	This method takes a boolean value and a string representing that boolean value and then passes it to the Assert method with the extra parameter of “Entry Invariant”.
`CheckPrecondition`	This method takes a boolean value and a string representing that boolean value and then passes it to the Assert method with the extra parameter of “Precondition”.
`CheckPostconditions`	This method takes a boolean value and a string representing that boolean value and then passes it to the Assert method with the extra parameter of “Postcondition”.
`CheckExceptionPostcondtions`	This method takes a boolean value and a string

	representing that boolean value and then passes it to the Assert method with the extra parameter of "Exceptional Postcondition".
`CheckExitInvariant`	This method takes a boolean value and a string representing that boolean value and then passes it to the Assert method with the extra parameter of "Exit Invariant".
Events	
`OnAssert`	This is a static public event that either a component developer or client can subscribe to in order to handle failed assertions however they wish. The only parameter passed to it is a string description of the failed assertion. This could potentially be changed to pass other information but this is just a proof of concept.

Table 5.2: Descriptions of all the members in the ClassConfiguration class.

5.2.3 Method Configuration

The `MethodConfiguration` structure is used to store four boolean values one for each of pre-, post-, exceptional postcondition and invariant checks. This is used in the custom class configuration classes created for each formally specified class. There is a single field of this type for every constructor, method or property in the formally specified class so that each of the four check types can be enabled or disabled individually.

Fields	
`PreDisabled`	This boolean value determines if the precondition is disabled for the particular method this configuration is associated with.
`PostDisabled`	This boolean value determines if the postcondition is disabled for the particular method that this configuration is associated with.
`ExceptionalPostDisabled`	This boolean value determines if the exceptional postcondition is disabled for the particular method that this configuration is associated with.
`InvariantDisabled`	This boolean value determines if the invariant checks are disabled for the particular method that this configuration is associated with.

Table 5.3: Descriptions of the fields in the MethodConfiguration structure.

On a side note these fields are labeled as “Disabled” instead of “Enabled” to simplify the process of initializing the class configurations. In .NET, all boolean values are initialized to false, so instead of having to set all them to true for the “Enabled” label, which could be difficult and time consuming since the class is loaded dynamically, the “Disabled” label was used instead.

5.3 Custom Assert Handler

One of the design goals was to allow for a customizable action to be performed when a contract violation occurs (i.e. assertion fails). `AssertHandler` is a delegate that is used by the `OnAssert` event in the `ClassConfiguration` class. It allows for someone to create a custom assertion handler. For someone to create a custom assertion handler they need to create a method that matches the signature of the `AssertHandler` delegate and then subscribe that method to the static `OnAssert` event in the `ClassConfiguration` class. For an example of how to do this see Section 6.4.4.

```
public delegate void AssertHandler(string assertCond);
```

Figure 5.3: The signature for the AssertHandler delegate.

5.4 Configuration Class Template

Along with generating the subclass as discussed in Section 4.3.1, a configuration class is also generated; a sample can be seen in Figure 5.4. This configuration class inherits from the more general `ClassConfiguration` discussed earlier. It contains a static method for getting the singleton instance stored in the `AssemblyConfiguration` class. The constructor should be private to help support the singleton pattern but in order to serialize and deserialize this class to and from the

XML configuration file the .NET XMLSerializer needs a default public constructor. The primary reason for generating a custom configuration class for every subclass is so the configurations can be controlled down to the method level more efficiently. It is more efficient because there is a `MethodConfiguration` field generated for every constructor, method and property and those field names are hard-coded in the generated subclass. As discussed in the last Section 4.3.1 the field names are unique because the name and parameter types are used to generate the unique name.

```
public class ClassNameConfiguration : ClassConfiguration
{
  public static ClassNameConfiguration GetConfig()
  {
    return Factory<ClassName>.GetConfig()
      as ClasNameConfiguration;
  }
  public ClassNameConfiguration() : base(typeof(ClassName)) { }

  // Create a MethodConfiguration for every constructor
  public MethodConfiguration UniqueConstructorName1;

  // Create a MethodConfiguration for every method
  public MethodConfiguration UniqueMethodName1;

  // Create a MethodConfiguration for every property
  public MethodConfiguration UniquePropertyName1;
}
```

Figure 5.4: Simplified sample template for a configuration class.

5.5 Using the Component

5.5.1 Deploying Component

After running the nContract tool there should be three files that are of interest in deploying the component to the client. The original assembly, the assembly with checks and the configuration file. All three files should be distributed to the client. The client can use all three files during development and configure the checks appropriately. Once the

client is ready to release the application they only need the original assembly and not the assembly with checks or the configuration file.

5.5.2 Creating Instances

As discussed in Section 3.4, in order for the client to be able to create instances of formally specified types with and with out checks a factory method must be used. Just for completeness the two types of factory method calls, generic and custom, are shown in Figure 5.5. The client needs to use one of them to create instances of formally specified types instead of the `new` operator. If a custom factory method is provided by the component developer it is recommended that the client use it instead of the generic method because it is more type safe and potentially more efficient.

```
// Generic Factory Call
ContractSpecification.Factory<ClassName>.Create(p1, p2);

// Custom Factory Call
ClassName.Create(p1, p2);
```

Figure 5.5: Example of both the generic and custom sample factory method calls.

5.5.3 Configuring the Component

By default if the assembly with checks or the configuration file is not in the search path of the executing program, the factory method returns an instance of the original class without checks. To enable the checks for any of the formally specified types in the original assembly, the assembly with checks and the configuration file need to be placed in the search path (usually the same directory as the original assembly). Now when the executable is run again the factory method should return an instance of the class with checks enabled assuming that they are enabled for that specific type in the configuration file. The generated configuration file has all checks enabled by default.

In the configuration file there is a `ClassConfiguration` element for every formally specified type in the associated assembly and an element with the unique names for the constructors, methods and properties. Figure 5.6 shows a simple configuration file with one class named `ClassName`, a single constructor that takes no parameters and a method `MethodName` that takes an integer as a parameter.

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfClassConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ClassConfiguration
    xsi:type="ClassNameConfiguration"
    TypeName="ClassName"
    Enabled="true" UseDefaultTraceAssert="true">
    <ctor_Void PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <MethodName_Int32 PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
  </ClassConfiguration>
</ArrayOfClassConfiguration>
```

Figure 5.6: Sample xml configuration file.

To disable checks for an entire class the `Enabled` attribute of the associated `ClassConfiguration` element should be set to false. To disable individual checks for a particular constructor, method or property, the `PreDisabled`, `PostDisabled`, `ExceptionalPostDisabled` or `InvariantDiabled` attribute should be set to true based on what type of checks need to be disabled. The configuration of checks for a particular assembly could potentially be made easier by using a tool to edit this configuration file but that is beyond the scope of this thesis. One last thing to notice is the `UseDefaultTraceAssert` attribute, which is used to configure whether or not to use the default `Trace.Assert` method when assertions fail for that class.

Chapter 6: nContract: A Walkthrough

The best way to understand how nContract works is to walk through an example. Let's look at a complete walkthrough from what a component developer needs to do all the way through to what the client needs to do.

6.1 Design the Sample Class

6.1.1 Include ContractSpecification Reference

Be sure to include a reference to *`ContractSpecification.dll`*. It is needed for the factory methods, contract attributes and configuration settings.

6.1.2 Design Constraints to Remember

As discussed in previous chapters, the following design constraints must be obeyed when writing a class to use nContract:

1) The class cannot be `private` nor can it be `sealed` or `static`.
2) All constructors that are to be checked at run-time must be `public` or `protected`. It is recommended that all constructors be marked as `protected` to force the client to use a factory method to create an instance.
3) All methods or properties that are to be checked at run-time must be `public` or `protected virtual`. If they are `private` or `static` they are not checked. Keep in mind the main thing is that the method or property is overridable. So an overridden method or property from a base class works as well. If an interface is implemented the methods or properties from the interface need to be marked `virtual` as well.

4) Fields are not checked. To get around this one could create a `public` or `protected virtual` property that references that field.

6.1.3 CharBuffer Sample Class

Keeping all the above ideas in mind Figure 6.1 shows the `CharBuffer` sample class that is used in this walk-through.

```
public class CharBuffer
{
  protected CharBuffer() : this(string.Empty) { }
  protected CharBuffer(string value) {
    stringBuilder = new StringBuilder(value);
    numberOfChars = value.Length;
  }
  public virtual void Append(string value) {
    stringBuilder.Append(value);
    numberOfChars += value.Length;
  }
  public virtual void Insert(int index, string value) {
    stringBuilder.Insert(index, value);
    numberOfChars += value.Length;
  }
  protected virtual void Remove(int startIndex, int length) {
    stringBuilder.Remove(startIndex, length);
    numberOfChars -= length;
  }
  public override string ToString() {
    return stringBuilder.ToString();
  }
  public virtual int Length {
    get { return numberOfChars; }
  }
  protected StringBuilder stringBuilder;
  protected int numberOfChars;
}
```

Figure 6.1: The CharBuffer sample class.

Notice that the methods and properties are either `public` or `protected virtual`, with the exception of the `ToString` method which is overridden but that is ok because it is still overridable which is the key. One other thing to notice is that the constructors are `protected` which forces the clients to use a factory method to create instances.

6.1.4 Include Factory Methods

Add the strongly typed factory methods in Figure 6.2 to the `CharBuffer` class. See Section 3.4 for other options. This approach is more efficient in the cases where no checks are enabled because the `new` operator is used instead of the dynamic `Activator.CreateInstance` method to create the object.

```
public static CharBuffer Create() {
  if (Factory<CharBuffer>.ChecksEnabled())
    return Factory<CharBuffer>.CreateChecksEnabledInstance();
  return new CharBuffer();
}
public static CharBuffer Create(string value) {
  if (Factory<CharBuffer>.ChecksEnabled())
    return Factory<CharBuffer>.CreateChecksEnabledInstance(value);
  return new CharBuffer(value);
}
```

Figure 6.2: The CharBuffer custom factory methods.

6.2 Formally Specify Using Attributes

6.2.1 Mark the Class as Formally Specified

To mark the class as being formally specified, the `FormallySpecified` attribute is attached to the class as in Figure 6.3.

```
[FormallySpecified]
public class CharBuffer
```

Figure 6.3: Marking the CharBuffer class as formally specified.

6.2.2 Include Abstract Model

This class is supposed to represent a list of characters therefore a `ModelField` attribute needs to be added to represent this abstract model. The model field, shown in Figure 6.4, needs to be given the type which is `List<char>`, the name (`Contents` is used) and the abstract to representation mapping expression. In most cases one wants to use model fields in the expressions for the pre-, postcondition and invariant conditions as

opposed to using internal or private fields. This is to help prevent clients from tying to a specific implementation.

```
[FormallySpecified]
[ModelField(typeof(List<char>), "Contents",
      @"new System.Collections.Generic.List<char>
      (this.ToString().ToCharArray())")]
public class CharBuffer
```

Figure 6.4: The ModelField for the CharBuffer class.

6.2.3 Include Class Invariants

There are two possible types of invariants that can be applied: a standard invariant or a representational invariant. There are no standard invariants for this class, only one representational invariant. As shown in Figure 6.5, it is added by using the `RepresentationInvariant` attribute. It is a representational invariant because it references internal data fields, if the invariant referenced only public or model fields then the standard `Invariant` attribute would be used instead.

```
[FormallySpecified]
[ModelField(typeof(List<char>), "Contents",
  @"new System.Collections.Generic.List<char>
    (this.ToString().ToCharArray())")]
[RepresentationalInvariant(
  "numberOfChars == stringBuilder.Length")]
public class CharBuffer
```

Figure 6.5: The RepresentationalInvariant for the CharBuffer class.

6.2.4 Include Method Contracts

The method contracts include three things the method pre-, post- and exceptional postconditions. The `Pre` attribute is used for the preconditions, the `Post` attribute is used for the postconditions and the `ExceptionalPost` attribute is used for exceptional postconditions. These method contracts are demonstrated only on the `Insert` method, for the method contracts of the other methods see the complete code in

the Appendix. Remember model fields are used instead of private fields in the C# condition expressions.

6.2.4.1 Preconditions

The precondition for the `Insert` method can have references to static, member or parameter variables. In Figure 6.6 the `Pre` attribute is used to provide the precondition for the `Insert` method. Notice that the only variables references are parameters and the model field `Contents`. This precondition simply checks bounds and makes sure the string to be inserted has a valid reference.

```
[Pre(@"index >= 0 &&
       index <= Contents.Count &&
       value != null")]
public virtual void Insert(int index, string value)
```

Figure 6.6: The precondition for Insert method of the CharBuffer class.

6.2.4.2 Postconditions

The postcondition for the `Insert` method can reference static, member or parameter variables. In addition to those variables the return value can be accessed through the variable name `result` and the old (i.e. value before method was executed) value of any model field can be accessed via `old.<modelFieldName>`. In Figure 6.7, the `Post` attribute is used to provide the postcondition for the `Insert` method. This postcondition ensures that the new count of characters is the same as the old count plus the length of the inserted string.

```
[Pre(@"index >= 0 &&
       index <= Contents.Count &&
       value != null")]
[Post(@"Contents.Count ==
        old.Contents.Count + value.Length")]
public virtual void Insert(int index, string value)
```

Figure 6.7: The postcondition for Insert method of the CharBuffer class.

6.2.4.3 Exceptional Postconditions

An exceptional postcondition for the `Insert` method can reference static, member or parameter variables. In addition it can reference the exception thrown through the variable name `excep`. It can have as many `ExceptionalPost` attributes as there are potential exceptions but in this case there is only one. Remember that the `ExceptionalPost` attribute expects two things, the exception type expected and an expression that must hold true if that exception is thrown. In Figure 6.8 the `ExceptionalPost` attribute is used to provide a postcondition for the `Insert` method that checks states the index must be out of bounds if the `ArgumentOutOfRangeException` is thrown.

```
[Pre(@"index >= 0 &&
       index <= Contents.Count &&
       value != null")]
[Post(@"Contents.Count ==
        old.Contents.Count + value.Length")]
[ExceptionalPost(typeof(ArgumentOutOfRangeException),
  "index < 0 || index > Contents.Count")]
public virtual void Insert(int index, string value)
```

Figure 6.8: The exceptional postcondition for Insert method of the CharBuffer class.

6.3 Generate a Subclass with Checks

Once the class is designed, specified and compiled into binary form then it is time to generate an assembly that contains the classes with the embedded checks.

6.3.1 Requirements for Running nContract

The nContract tool is written in C# targeted at the .NET Framework version 2.0 beta 1.

The following files are needed to execute nContract:

- *nContract.exe*
- *nContractTemplate.cst*
- *ContractSpecification.dll*

- *CodeSmith.Engine.dll*

Of course on top of those files the input assembly is needed in this sample case the assembly that the `CharBuffer` class is compiled into is *ExampleComponent.dll*.

6.3.2 Running nContract

Place all the files listed above in the same directory and run the command in Figure 6.9 from the command line in that directory.

```
nContract ExampleComponent.dll
```

Figure 6.9: The nContract command for CharBuffer sample.

6.3.3 nContract Output

For this example if the command is successful the following files are created:

- *CharBufferWithChecks.cs*
- *ExampleComponentWithChecks.dll*
- *ExampleComponentWithChecksConfig.xml*

Notice that there are three things that are created the first being one *.cs* file for every class in *ExampleComponent.dll* that is marked as formally specified. For this sample the only formally specified type is `CharBuffer` so the only *.cs* file generated is *CharBufferWithChecks.cs*. The next thing created is the assembly *ExampleComponentWithChecks.dll* which is the assembly that results from compiling all the generated *.cs* files. The final thing created is a configuration file *ExampleComponentWithChecksConfig.xml* which is used to configure the checks down. Note that the *.cs* files are only for outputted to help debug any syntax errors in the generated classes and they are not deployed with the component.

6.3.4 A Closer Look at the Generated Class

The generated classes in question here is `CharBufferWithCheck` along with a configuration class `CharBufferConfiguration` both are in the file *`CharBufferWithChecks.cs`*.

6.3.4.1 Generated Constructor

For each constructor in the original class, there is a constructor with matching parameters generated. There is also a static `Create` method generated that acts as a factory method that checks the preconditions before the constructor is called and exceptional postconditions. Figure 6.10 shows the generated constructor that corresponds to the second constructor in the `CharBuffer` class.

```
protected CharBufferWithChecks(System.String value)
  : base(value) {
  if (config.ctor_String.PostDisabled == false) {
    MethodEnforcement.EnterChecks(false);
    config.CheckPostcondition(Contents.Count ==
      value.Length, @"Contents.Count == value.Length");
    MethodEnforcement.ExitChecks(false);
  }
  if (config.ctor_String.InvariantDisabled == false) {
    MethodEnforcement.EnterChecks(false);
    config.CheckExitInvariant(numberOfChars ==
      stringBuilder.Length, @"numberOfChars ==
      stringBuilder.Length");
    MethodEnforcement.ExitChecks(false);
  }
}
```

Figure 6.10: The generated constructer for CharBuffWithChecks class.

Notice that only the postcondition and the exit invariant is checked inside the constructor. The invariant cannot be checked before the class is created and the precondition cannot be checked inside this constructor because of the call to the base constructor happens before the body is executed. To deal with the precondition checks a factory method is created for this class.

Instead of actually calling the constructor to create instances of the class `CharBufferWithChecks`, the factory method shown in Figure 6.11 is used. That way the precondition can be checked before the object is actually created. This does however limit the variable access in the precondition to only static variables and parameters. Also notice that if the constructor had any exceptional postconditions they would be checked between the `EnterChecks` and `ExitChecks` method calls in the `catch` block.

```
public static CharBuffer Create(System.String value)
{
  if(config.ctor_String.PreDisabled == false) {
    MethodEnforcement.EnterChecks(false);
    config.CheckPrecondition(value != null,
            @"value != null");
    MethodEnforcement.ExitChecks(false);
  }
  CharBufferWithChecks newObject = null;
  try
  {
    newObject = new CharBufferWithChecks(value);
  }
  catch(Exception ex)
  {
    throw;
  }
  return newObject;
}
```

Figure 6.11: The generated Create factory method for CharBufferWithChecks class.

6.3.4.2 Generated Method

```
public override void Insert(int index, string value)
{
  Generic.List<System.Char> oldContents = null;
  if(config.Insert_Int32_String.PostDisabled == false) {
    oldContents = Contents;
  }
  if(config.Insert_Int32_String.InvariantDisabled == false) {
    MethodEnforcement.EnterChecks(false);
    config.CheckEntryInvariant(numberOfChars ==
      stringBuilder.Length, @"numberOfChars ==
      stringBuilder.Length");
    MethodEnforcement.ExitChecks(false);
  }
  if(config.Insert_Int32_String.PreDisabled == false) {
    MethodEnforcement.EnterChecks(false);
    config.CheckPrecondition(index >= 0 && index <=
      Contents.Count && value != null,
      @"index >= 0 && index <= Contents.Count &&
      value != null");
    MethodEnforcement.ExitChecks(false);
  }
  try {
    base.Insert(index, value);
    if(config.Insert_Int32_String.PostDisabled == false) {
      MethodEnforcement.EnterChecks(false);
      config.CheckPostcondition(Contents.Count ==
            oldContents.Count + value.Length,
            @"Contents.Count == oldContents.Count + value.Length");
      MethodEnforcement.ExitChecks(false);
    }
  }
  catch(Exception ex) {
    if(config.Insert_Int32_String.ExceptionalPostDisabled == false) {
      MethodEnforcement.EnterChecks(false);
      if(ex is System.ArgumentOutOfRangeException) {
        ArgumentOutOfRangeException excep =
            ex as ArgumentOutOfRangeException;
        config.CheckExceptionalPostcondition(index < 0 ||
            index > Contents.Count,
            @"index < 0 || index > Contents.Count");
      }
      MethodEnforcement.ExitChecks(false);
    }
    throw;
  }
  finally {
    if(config.Insert_Int32_String.InvariantDisabled == false) {
      MethodEnforcement.EnterChecks(false);
      config.CheckExitInvariant(numberOfChars ==
            stringBuilder.Length,
            @"numberOfChars == stringBuilder.Length");
      MethodEnforcement.ExitChecks(false);
    }
  }
}
```

Figure 6.12: The generated version of Insert method for the CharBufferWithChecks class.

To get an idea of what a generated method looks like take a look at Figure 6.12 which shows the generated version of the `Insert` method. Notice that first the invariant is checked then the precondition. Following that is a `try/catch` block that wraps the call to the base `Insert` method which is followed by the postcondition. In the `catch` block the exceptional postconditions are handled and lastly the invariant is checked in the `finally` block. One other thing to notice is the pre-state version of the model field `Contents` is stored before the execution of the method and the postcondition value `old.Contents` is changed to `oldContents` to reference the local variable.

6.3.4.3 Generated Configuration Class

To get an idea of how the checks are configurable down to the method level, take a look at Figure 6.13 which is the configuration class that is generated.

```
public class CharBufferConfiguration :
      ContractSpecification.ClassConfiguration
{
  public static CharBufferConfiguration GetConfig() {
      return Factory<CharBuffer>.GetConfig() as
      CharBufferConfiguration;
  }
  public CharBufferConfiguration() :
      base(typeof(ExampleComponent.CharBuffer)) { }

  public MethodConfiguration ctor_Void;
  public MethodConfiguration ctor_String;
  public MethodConfiguration Append_String;
  public MethodConfiguration Insert_Int32_String;
  public MethodConfiguration Remove_Int32_Int32;
  public MethodConfiguration ToString_Void;
  public MethodConfiguration Equals_Object;
  public MethodConfiguration GetHashCode_Void;
  public MethodConfiguration Length_Int32;
}
```

Figure 6.13: The generated version of the CharBuffer configuration class, CharBufferConfiguration.

The `CharBufferWithChecks` class contains a member field `config` that holds a reference to the singleton instance of this `CharBufferConfiguration` class. The base `ClassConfiguration` class contains information that is general for all

classes as a whole such as whether or not the checks are enabled for this class or not. The generated class contains class specific information like a member to hold method configurations for every constructor, method and property of the original class. The method configuration consists of three boolean values one for the pre-, postcondition and invariant check. So this `CharBufferConfiguration` class contains a `MethodConfiguration` field for every constructor, method or property in the `CharBuffer` class.

6.4 Sample Client Use of Component

To wrap up this `CharBuffer` sample lets look at how a client actually creates instances and configures the checks for this class.

6.4.1 Create Instances of CharBuffer

Now as discussed in Section 3.4 there are multiple ways to create an instance of this class however for this sample the factory method included in the `CharBuffer` class is used. Figure 6.14 shows a sample object initialization.

```
CharBuffer cb = CharBuffer.Create("abc");
```

Figure 6.14: How a client creates an instance of the CharBuffer class.

By default if the *`CharBufferWithChecks.dll`* and *`CharBufferWithChecksConfig.xml`* files are not located in the search path of the executable all checks are disabled. Therefore by using the *`ExampleComponent.dll`* just like any other assembly a standard `CharBuffer` class is created by this factory method.

6.4.2 Enable Checks for CharBuffer

To enable the checks for the `CharBuffer` class just place the *`CharBufferWithChecks.dll`* and *`CharBufferWithChecksConfig.xml`* files in the search path of the executable (easiest if placed in the same directory). Now when the executable is run again and the factory method is called an object of type `CharBufferWithChecks` is created. This can be changed by modifying the configuration file but the default configuration file that is generated has everything enabled by default.

6.4.3 Configure Checks for CharBuffer

Take a look at Figure 6.15 which is the *`CharBufferWithChecksConfig.xml`* file that is generated. There is a `ClassConfiguration` section for every formally specified class in the assembly and for every constructor, method or property in that given class there is a `MethodConfiguration` element. Notice that the enabled attribute for the `CharBuffer ClassConfiguration` section is set to true. This is the default when the configuration file is generated that is why when the component with checks and the configuration file are found the checks are enabled.

Now if one wants to disable only certain checks for certain methods then the associated attribute for the appropriate method should be set to true. For example to disable invariant checks for the `Insert` method the `InvariantDisabled` attribute on the `Insert_Int32_String` element would be set to true.

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfClassConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ClassConfiguration
    xsi:type="CharBufferConfiguration"
    TypeName="ExampleComponent.CharBuffer"
    Enabled="true" UseDefaultTraceAssert="true">
    <ctor_Void PreDisabled="false"
      PostDisabled="false" InvariantDisabled="false" />
    <ctor_String PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <Append_String PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <Insert_Int32_String PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <Remove_Int32_Int32 PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <ToString_Void PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <Equals_Object PreDisabled="false"  PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <GetHashCode_Void PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    <Length_Int32 PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
  </ClassConfiguration>
</ArrayOfClassConfiguration>
```

Figure 6.15: The generated configuration file for the CharBuffer class.

6.4.4 Creating a Custom Assertion Handler

In the `ClassConfiguration` class there is a `static event OnAssert` that fires when an assertion fails. This allows the client to create a custom assertion handler and subscribe it to this event to handle the failed assertions however they feel fit. By default the only thing that happens when an assertion fails is the standard .NET assertion dialog pops up from the call to `Trace.Assert`. That can be turned off on a per class basis by setting the `UseDefaultTraceAssert` setting to false. Note that the client needs to add a reference to the ContractSpecification component in order to setup a custom assertion handler.

Figure 6.16 shows a sample custom assertion handler that writes the assertion condition that failed out to the console.

```
static void ClassConfiguration_OnAssert(string assertCond) {
      Console.WriteLine(assertCond);
}
```

Figure 6.16: Sample AssertHandler method.

Figure 6.17 shows how to subscribe the above handler to the `OnAssert` event.

```
ClassConfiguration.OnAssert +=
      new AssertHandler(ClassConfiguration_OnAssert);
```

Figure 6.17: How a developer would subscribe a custom AssertHandler to the OnAssert event.

Chapter 7: Feature Comparison

There are a number of key features that are important for this thesis and similar work. Some of these features are used in the coming sections to compare this research against other techniques and tools in the area.

7.1 List of Contract Tools Used in the Comparisons

The list of tools that are used in the comparisons have already been discussed in Chapter 2 but are listed here for completeness.

- Eiffel [Meyer, 1992]
- Spec# [Barnett et al., 2004]
- XC# ["eXtensible C#"]
- JContractor [Karaorman et al., 1999]
- IContract [Kramer, 1998]
- JMLC [Cheon, 2003]
- Handshake [Duncan & Hoelzle, 1998]
- ContractJava [Findler et al., 2001]
- Tan-JML [Tan & Edwards, 2003]
- C++ Wrappers [Edwards et al., 2004]

The names in the list above are used as the names in the comparisons.

7.2 Contract Storage

Contract Storage	**Tools**
Structured Comments	JMLC, IContract, Tan-JML, C++ Wrappers
Language Construct	Eiffel, Spec#
Separate File	Handshake
Naming Convention	JContractor
Metadata	XC#, nContract

Table 7.1: Contract storage types for different tools.

Storing the contract information is an important aspect of providing a run-time contract verification tool. As seen from Table 7.1 one of the most common ways to

provide contract information among the listed tools is to provide it via structured comments. The biggest disadvantage to this approach is that the contract information is only available through the source code. So either the developers have to give source code access or somehow generate written documentation that contains this information in order for the client to have access to the contract information. However, the developer doesn't want to give source code access and generated written documentation can get outdated very quickly. Perhaps a better approach would be to provide the contact information in such way that it gets integrated with the binary component. One such way to integrate this information is by storing the contract in the metadata of the component as is done in this research. This way any tool can easily retrieve the contract therefore the client can always have the most updated contract information, without the component developer doing any extra work.

7.3 Assertion Design Approach

Design Approach	**Tools**
Inline Assertions	Eiffel, Spec#, IContract, XC#, JContractor
Separate Methods	JMLC, ContractJava, Handshake
Separate Classes	Tan-JML, C++ Wrappers, nContract

Table 7.2: Design approaches for including assertions for different tools.

The common approaches of in-lining assertions or putting assertions in separate or "wrapper" methods has the potential of inadvertently introducing bugs into the component. Anytime one modifies existing code there is this possibility. Granted, there is less possibility with separate methods but still potential because they usually do require renaming the original method and also add methods to the original class. The safest way to include assertion checks is to do so without modifying the original code. The usual

way to include insertions without modifying the original code is to create separate classes to house the checks.

On the other hand one advantage of in-lining assertions in the original code is that objects can still be created by using the `new` operator. As seen by this research when checks are placed in a separate class one is forced to use object-oriented designs (i.e. interfaces or subclasses) and when doing this a factory method is required to create objects. Therefore one trade-off is to either use a factory method or potentially introduce bugs. Since the primary reason for design-by-contract is to produce more reliable software [Meyer, 1992] (i.e. software with less bugs) it doesn't make since to use an approach that potentially introduces bugs.

7.4 Configurability

One of the most useful parts of this research is providing checks that are configurable. An example would be if a client is debugging and has narrowed down a problem to a single class or even a single method it is nice to be able to disable everything except what is needed. This in turn could save time and money in tracking down and fixing problems.

7.4.1 Compile-time vs. Run-time

Time	**Tools**
Compile-time	Eiffel, Spec#, IContract, XC#, JMLC, ContractJava
Link-time	C++ Wrappers
Run-time	JContractor, Handshake, Tan-JML, nContract

Table 7.3: Decision time to include checks or not for different tools.

Most tools either enable or disable checks at compile-time or run-time as seen in Table 7.3 however Edwards et al. [Edwards et al., 2004] found a unique way to include or

exclude checks at link-time by selectively linking particular object files. Compile-time exclusion or inclusion of checks is a common approach. Usually this is the case because the tool is either a compiler or a preprocessor that deals directly with the source code. In order to be run-time based the environment usually needs to support some sort of dynamic loading or instrumentation, such as with the .NET CLR (Common Language Runtime) and the JVM (Java Virtual Machine). The common run-time approaches for Java usually deal with using custom class loader to dynamically instrument classes as they are loaded. There is no custom class loader available in .NET but assemblies can be loaded dynamically, which is what happens in this research. One of the primary benefits of a run-time approach is that a component client can enable or disable checks without source code access or recompilation.

7.4.2 Configuration Levels

Tool	Package/Assembly	Class	Method
Eiffel	Pre/Post/Inv	No	No
Spec#	Pre/Post/Inv	Inv	Inv
IContract	Pre/Post/Inv	Pre/Post/Inv	Pre/Post/Inv
Tan-JML	Pre/Post/Inv	Pre/Post/Inv	No
C++ Wrappers	Pre/Post/Inv	Pre/Post/Inv	No
nContract	Pre/Post/Except/Inv	Pre/Post/Except/Inv	Pre/Post/Except/Inv

Table 7.4: Summary of the configurability of different tools.

Table 7.4 shows the configurability of the tools and to what degree they are configurable (i.e. which checks are configurable where Pre = Precondition, Post = Postcondition, Except = Exceptional Postcondition, Inv = Class Invariant). The tools that are not listed in Table 7.4 either do not provide any method of configurability or they only have the globally all on or off setting.

Keep in mind that even though Eiffel, Spec# and IContract are configurable it still requires a recompilation for the new configuration to take place. Tan-JML and C++ Wrappers are both configurable at run-time down to the class level. Therefore, they do have some fine grain control of the checks but for efficiency reasons they didn't provide settings at the individual method level. The method nContract uses for providing configuration down to the method level is efficient because of the generated custom class configuration which contains direct references to the configuration settings for each method. IContract is the only other tool that provides such a fine grained configuration however it is still only compile-time configuration.

7.5 Custom Assertion Actions

Having the ability to execute customizable actions on failed assertions can be a big advantage to the developer. During development and debugging, if a contract violation occurs different developers want different actions to be taken. Some may want the violation to be logged, throw an exception, start the debugger or simply display a message. Allowing for customizable actions gives the developer more freedom to handle the violation in their own way.

By taking advantage of the .NET event model nContract is easily able to provide support for customizable actions during contract violations. A developer can create any number of custom assert handlers that can be subscribed to the fail assertion event thus allowing complete freedom to the developer, whether they are the component or client developer. APP [Rosenblum, 1995] is the only other known tool that provides such customizable actions for failed assertions. Most other tools usually hardcode the assertion action with the most common action being to throw an exception of some sort.

Chapter 8: Performance Evaluation

This chapter discusses some of the run-time performance implications that are associated with this research. Experiments to test the performance of virtual methods, object creation and components with and without inline checks are the primary focus. Remember one of the primary ideas of nContract is to allow for checks to be disabled in the release version of a product with little or no performance penalty. This chapter discusses these performance issues and why it is beneficial to be able to eliminate them in release code.

8.1 Test Setup

All the performance tests that are run for this chapter are run on a machine with a P4 3 GHz (Hyper-threaded) processor and 1 GB of memory. All the test setups require some code to be timed and for this the Stopwatch class from the .NET Framework 2.0 is used. All the tests are run at least 1 million times and run multiple times to help equalize any noise in the results. The execution times used in the charts are calculated by taking the average of all the execution times. Most of these tests use the `CharBuffer` class but some require a customized version of it.

8.2 Virtual Methods

For running this test, two versions of the same class were created, one with all virtual methods and the other with no virtual methods. Both classes contain an empty method and non-empty method (the `Append` method from the `CharBuffer` class). This test was run 10 times with each run consisting of 100 million calls to the particular method.

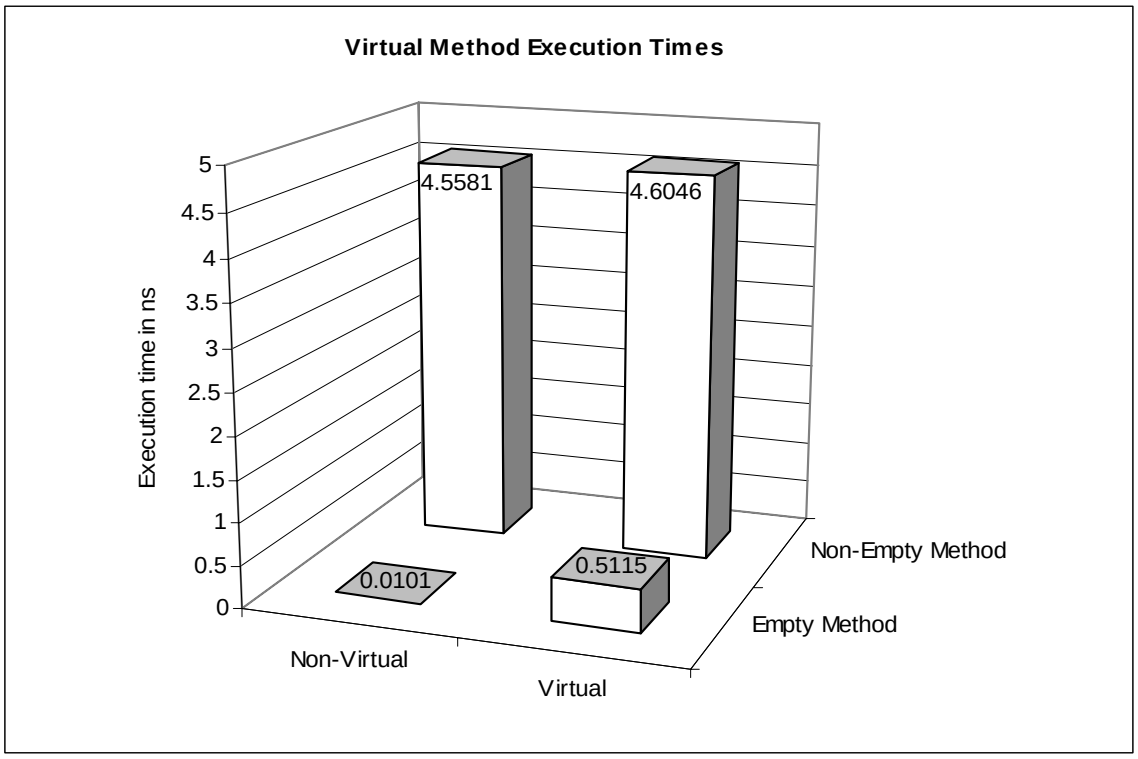

Figure 8.1: The times of virtual vs. non-virtual and Empty vs. Non-Empty method calls.

The difference in execution time for the non-empty method calls is less than 1%. On the other hand, for the empty method calls the virtual call is about 50 times slower than the non-virtual call. Keep in mind that the difference in execution time between the empty method calls is only about 0.5 ns. Therefore the time to do a virtual table lookup is about 0.5 ns. This is negligible in comparison to the actual execution time of a non-empty method.

The subclass model relies on the use of virtual methods. This test was used to determine the performance implications of using virtual methods versus non-virtual methods. As Figure 8.1 points out the time overhead to perform the virtual table lookup is minor compared to the execution time of a non-empty method. Therefore any reasonable method that does some amount work will not suffer much from being a virtual method.

8.3 Object Creation

Objects can be created statically by using the `new` operator or dynamically by using the `Activator.CreateInstance` method. The nContract technique uses the following three different methods of object creation.

- New – Creates object using the traditional static `new` operator.
- Create Instance – This method is used by the generic factory method to create instances of objects without embedded checks. It creates objects dynamically by passing the parameters and the static type (i.e. type provided by the `typeof` operator) to `Activator.CreateInstance`.
- Dynamic Factory Call – This method is used to create objects dynamically with checks embedded. It is used instead of the Create Instance approach so that preconditions can be checked before object creation (see Section 4.3.1.1 for more details). A reference to the dynamically loaded type is gathered by doing a dynamic type lookup based on a string literal. This reference is cached and reused for subsequent object creations. To actually create the object a string literal is used to dynamically invoke a static factory method. This static factory method uses the new operator to actually create the object.
- Delegate Factory Call – This method can be used by a component developer in a custom factory method to create objects with checks embedded. A delegate is initialized once when the class configuration is loaded to the appropriate factory method generated in the subclass. Then to create the object the delegate is invoked.

Looking at Figure 8.2 it can be seen that `new` operator performs about 31 times faster than the dynamic create instance method, about 38 times faster than the dynamic factory call and about 1.5 times faster than the delegate factory call. The `new` operator is faster because it is static and at compile-time the compiler can hardwire a direct reference to the correct constructor to invoke; whereas with the create instance and dynamic factory call methods these references need to be retrieved at run-time. The delegate call is comparable to the new call because it only needs an initial reference lookup and from that point onward it just invokes a method by direct reference.

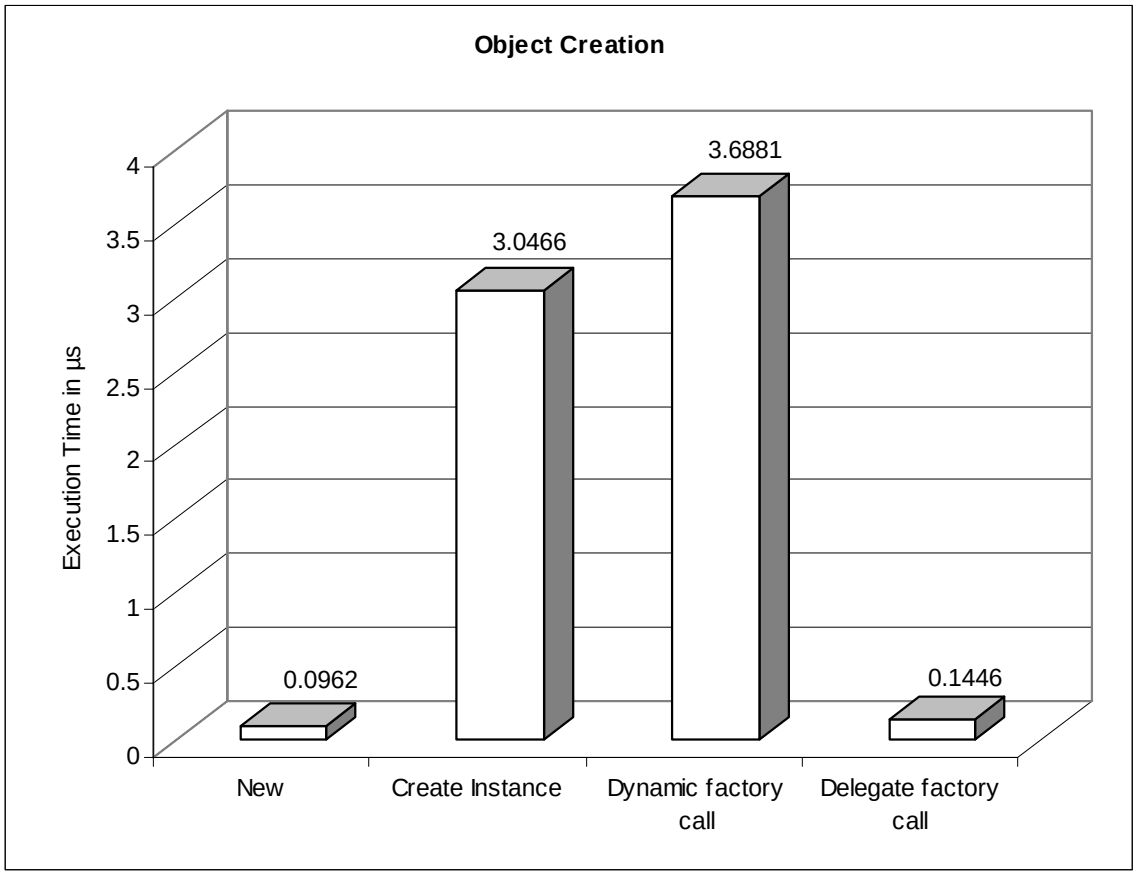

Figure 8.2: The amount of time it takes to create an instance of an object using different creation methods

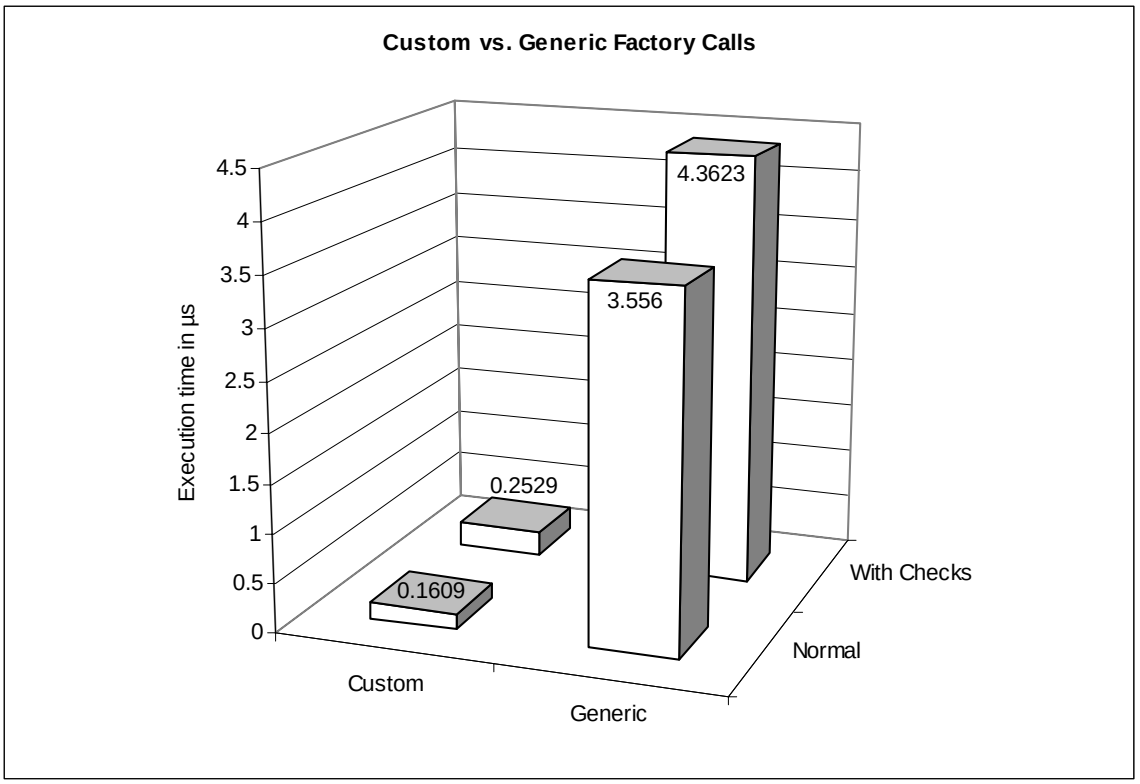

Figure 8.3: Amount of time it takes to create objects using a custom and a generic factory call.

From Figure 8.3 the normal (i.e. without embedded checks) factory calls are proportional to the execution times of the new operator and the create instance method respectively. There is a little more overhead with the normal factory calls versus the new and create instance methods due to checking the configuration to determine if an instance with checks embedded needs to be created. Similarly the execution times of the factory calls that create an instance with checks embedded are proportional the delegate factory method call and the dynamic factory method call from Figure 8.2.

The execution times in Figure 8.3 demonstrate why a component developer is recommended to create a custom factory method. By creating a custom factory method the normal case will execute about 1.6 times slower than using a typical new operator as opposed to the 37 times slower if the generic factory is used. Also if the component developer decided to use the delegate approach the execution time of the case with

checks included will be only 2.6 times slower than a typical new operator compared to the 45 times slower if they used the dynamic factory call like the generic factory.

8.4 nContract Performance

Now is the time to see what performance gain developers get by omitting checks in the final release of a component. This test consists of executing methods from the original class, the class that includes all checks enabled and the class that includes checks but with all disabled. Each test includes 1 million executions of a call to the `Insert` and `Remove` methods and two calls to the `Length` property of the `CharBuffer` class. This test is run 10 times and then all the execution times are averaged to come up with the execution times for Figure 8.4.

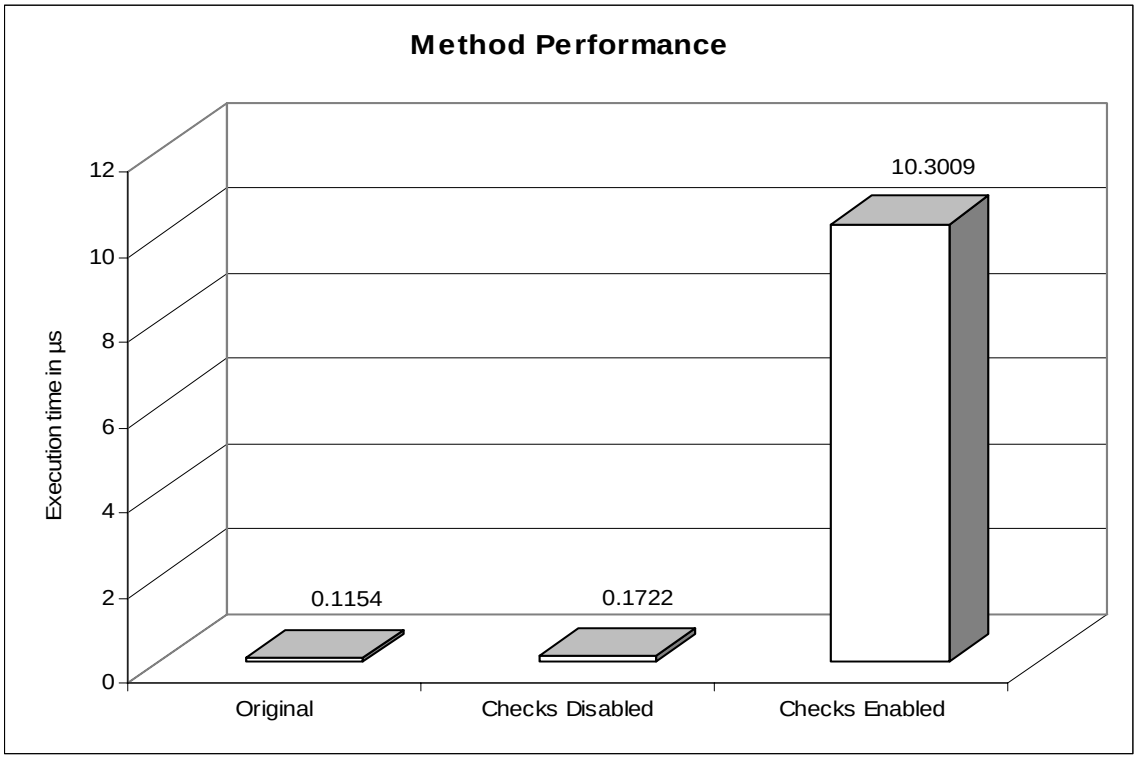

Figure 8.4: The execution times of some sample methods for the original component and the component with checks enabled and disabled.

When all checks are enabled, the execution time of the component is about 89 times slower than the original component with no inline assertions. However the version with checks embedded but disabled is only about 1.5 times slower. This small 0.05 µs slow down is primarily caused by four boolean expressions that are executed to determine if particular assertions are enabled. The execution time of the version with checks enabled is so much slower because of the execution of the assertions and all the supporting code. A large portion of time in the checks enabled version is due to executing the mapping expressions for the model fields and also storing the pre-state values for those model fields.

These times are for one particular class and they will vary from class to class depending on the assertion conditions and model field expressions. Howeve,r the pattern is pretty clear that if the checks are separated from the original code then in the general case when the checks are not included there is an obvious performance gain.

Chapter 9: Conclusion

9.1 Summary

The primary focus of this research is to provide a solution to the problem of enabling or disabling run-time contract verification for .NET components without requiring recompilation or access to the source code. Remember the three phases of accomplishing this goal are: embedding contract information, packaging contract checking code, and configuring contract checking code.

.NET attributes allow for additional declarative information to be added to a .NET binary component's metadata. Therefore a set of contract attributes were created to allow a component developer to formally specify a component. In turn this contract information is stored into the metadata of the binary component.

Once the component developer creates a .NET component with formally specified types they can run the nContract tool on it. The nContract tool then retrieves the contract information from the binary component and packages it as assertion checks in a subclass of the original class. Since a subclass is used to package the checks a decision can be made at run-time by a factory method to create an instance of the original class without checks or a subclass with checks. That way if no checks are enabled then the original class is created and there is no performance penalty imposed by the assertion checks. There are two minor performance penalties incurred by this design. One penalty is due to requiring all the methods to be virtual. The experimental evaluation in Section 8.2 shows this penalty is very minor compared to the amount of work a method does, however. The other penalty is due to the objects being created dynamically, using reflection. Section 8.3

shows that this penalty can be eliminated when all checks are disabled if the component developer creates custom factory methods that use the `new` operator instead of reflection. Even though there are two small performance penalties for using this approach, Section 8.4 shows their impact is negligible compared to the times of executing the assertions themselves.

On top of generating subclasses for each formally specified class the nContract tool also generates a custom configuration class. This class allows for efficient configuration of the checks down to method and check type level. The nContract tool also generates an XML configuration file and based on the settings in this file it is decided which checks are enabled or disabled at run-time.

Now when all the above is put together it forms the nContract model that allows for enabling or disabling run-time contract verification without requiring recompilation or access to source code.

9.2 Limitations

Even though this research covers the general object-oriented designs there are still some cases that cannot be dealt with. For example, this design approach cannot provide checks for static methods nor can it for sealed (final) classes. The `new` operator cannot be used to create instances of the classes a factory method must be used instead. These are a couple of the limitations that prevent some object-oriented designs from working with this approach. Overall this research supports most object-oriented design patterns.

Another area of limitation is the level of formal specification. Although all the basic formal specification constructs can be expressed there are some that aren’t provided. For example quantifiers are not supported. There is only limited support of model

specifications. The only pre-state values that can be accessed are model field pre-state values. These are all limitations that could potentially be added with future work. It would also be nice if the conditions were not strings that way a compile error could be given during the development of the component. As it stands right now if there is a syntax error in the condition expressions it is not reported until the sub-class is generated and then compiled by the nContract tool.

9.3 Future Work

As stated in the last section there is room for improvement and additions to the level of formal specification discussed in this research. Another thing that could help improve the formal specification part is the addition of static analysis. For example with static analysis it could be verified at compile-time that only pure method calls are used in the condition expressions.

As aspect weavers mature in the .NET world it could become possible to do some of this work using aspects as discussed in Section 2.4. Even if a complete solution cannot be created at least maybe an aspect could replace new calls with factory method calls. This would allow the client to use the component like any other by using the `new` operator. Other future work could potentially implement a similar approach for Java since it supports metadata in version 1.5 ["JSR 175: A Metadata Facility for the Java Programming Language"]. Also since AOP is more mature in Java it could potentially be used more with a Java approach.

Even though this research provides a rich, fine-grained configuration model it has to be configured manually through the use of an xml file. It would be nice to have a simple GUI tool that would allow for easy enabling or disabling of individual checks at

the assembly, namespace, class and/or method levels. This would make configuration easier for the component developer and client.

9.4 Conclusion

Having configurable run-time contract verification can provide a great tool for component developers and clients. It adds an extra layer of verification that allows component developers to ensure their component does what it is supposed to do given the correct input. For component clients it helps alleviate the problem of not knowing the correct input and expected output for a particular component. By having this verification when the client uses a component they are more likely to produce more reliable software systems with that component. Using the nContract design and tool allows for component developers and clients to get the benefit of run-time contract verification with the added power to disable it without performance penalty.

Appendix A: Source Code

Complete Formally Specified CharBuffer class:

```
[FormallySpecified]
[ModelField(typeof(List<char>), "Contents",
  @"new System.Collections.Generic.List<char>
    (this.ToString().ToCharArray())")]
[RepresentationalInvariant(
  "numberOfChars == stringBuilder.Length")]
public class CharBuffer {
  [Post("Contents.Count == 0")]
  protected CharBuffer() : this(string.Empty) { }
  [Pre("value != null")]
  [Post("Contents.Count == value.Length")]
  protected CharBuffer(string value) { }
  [Pre("value != null")]
  [Post(@"Contents.Count ==
          old.Contents.Count + value.Length")]
  public virtual void Append(string value) { }
  [Pre(@"index >= 0 &&
         index <= Contents.Count &&
         value != null")]
  [Post(@"Contents.Count ==
          old.Contents.Count + value.Length")]
  [ExceptionalPost(typeof(ArgumentOutOfRangeException),
    "index < 0 || index > Contents.Count")]
  public virtual void Insert(int index, string value) { }
  [Pre(@"startIndex >= 0 &&
         startIndex + length <= Contents.Count")]
  [Post("Contents.Count == old.Content.Count - length")]
  [ExceptionalPost(typeof(ArgumentOutOfRangeException),
    @"startIndex < 0 || length < 0 ||
      index + length > Contents.Count")]
  protected virtual void Remove(int startIndex, int length) { }
  [Post(@"result != null &&
          result.Length == Contents.Count")]
  public override string ToString() { }
  [Post("result == Contents.Count")]
  public virtual int Length { }
}
```

For the complete source code of this project look for a zip file at the same location you obtained the electronic version of this thesis.

References

["AspectJ Project"] AspectJ Project. Retrieved 1/2005, from http://aspectj.org

["eXtensible C#"] eXtensible C#. Retrieved 1/2005, from http://www.resolvecorp.com/Products.aspx

["iControl"] iControl. Retrieved 1/2005, from http://icplus.sourceforge.net/iControl.html

["JSR 175: A Metadata Facility for the Java Programming Language"] JSR 175: A Metadata Facility for the Java Programming Language. Retrieved 1/2005, from http://jcp.org/en/jsr/detail?id=175

["LOOM.NET"] LOOM.NET. Retrieved 1/2005, from http://www.dcl.hpi.uni-potsdam.de/research/loom/

[Arnout & Simon, 2001] Karine Arnout and Raphaël Simon. (2001). The.NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel. 14.

[Barnett et al., 2004] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte. (2004, 10/2004). The Spec# Programming System: An Overview. Retrieved 1/2005, from http://research.microsoft.com/SpecSharp/papers/krml136.pdf

[Cheon, 2003] Yoonsik Cheon. (2003). *A Runtime Assertion Checker for the Java Modeling Language* (No. #03-09): Department of Computer Science, Iowa State University.

[Diotalevi, 2004] Filippo Diotalevi. (2004, 7/2004). Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ. Retrieved 1/2005, from http://www-106.ibm.com/developerworks/library/j-ceaop/?ca=dnt-528

[Duncan & Hoelzle, 1998] Andrew Duncan and Urs Hoelzle. (1998). *Adding Contracts to Java with Handshake* (No. TRCS98-32): University of California at Santa Barbara.

[Edwards, 2001] Stephen H. Edwards. (2001, 10/2001). *Toward reflective metadata wrappers for formally specified software components.* Paper presented at the Specification and Verification of Component Based Systems, held in conjunction with OOPSLA 2001.

[Edwards et al., 2004] Stephen H. Edwards, Murali Sitaraman, Bruce W. Weide and Joseph Hollingsworth. (2004). Contract-Checking Wrappers for C++ Classes. *IEEE Trans. Softw. Eng., 30*(11), 794-810.

[Findler & Felleisen, 2001] Robert Bruce Findler and Matthias Felleisen. (2001). *Contract Soundness for object-oriented languages.*

[Findler et al., 2001] Robert Bruce Findler, Mario Latendresse and Matthias Felleisen. (2001). *Behavioral contractsbehavioral subtyping*.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*:Addison-Wesley.

[Karaorman et al., 1999] Murat Karaorman, Urs Holzle and John Bruno. (1999). jContractor: A Reflective Java Library to Support Design by Contract. 175-196.

[Kramer, 1998] Reto Kramer. (1998). iContract - The Java(tm) Design by Contract(tm) Tool. *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, 295.

[Liskov & Wing, 1994] Barbara H. Liskov and Jeannette M. Wing. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst., 16*(6), 1811-1841.

[Meyer, 1992] Bertrand Meyer. (1992). Applying "Design by Contract". *Computer, 25*(10), 40-51.

[Rosenblum, 1995] David S. Rosenblum. (1995). A Practical Approach to Programming With Assertions. *IEEE Trans. Softw. Eng., 21*(1), 19-31.

[Simon et al., 2002] Raphael Simon, Emmanuel Stapf and Bertrand Meyer. (2002, July 2002). Full Eiffel on the.NET Framework. *MSDN* Retrieved 1/2005, from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp

[Smith, 2002] Eric Smith. (2002). CodeSmith. Retrieved 1/2005, from http://www.ericjsmith.net/codesmith/

[Tan & Edwards, 2003] Roy Patrick Tan and Stephen H. Edwards. (2003). *An Assertion Checking Wrapper Design for Java* (No. #03-11). Ames, IA: Department of Computer Science, Iowa State University.

Westley Haggard

Vita

Education

Northern Kentucky University, Highland Heights, KY

8/1998 – 5/2002
Bachelors of Science in Mathematics and Computer Science
Cum. GPA 4.0

Virginia Polytechnic Institute and State University, Blacksburg, VA

8/2003 – 5/2005
Masters of Science in Computer Science
Cum. GPA 3.95

Work Experience

Navaro Medical Solutions

2/2002 – 12/2004
- Software Developer, working primarily with Microsoft .Net technologies.
- Developing a N-Tier Medical Billing Software utilizing C# and the Microsoft .Net Framework with a SQL Server 2000 backend.

Northern Kentucky University

6/1998 – 2/2002
- Full Time - Computer Lab Technician
- Created and implemented a process that allowed IT department to setup and maintain 20 times more computers around campus with the same staff.

Military: Kentucky Army National Guard

7/1996 – 7/2002
- Distinguished Honor Graduate: Radio Technician (Secret Security Clearance)

Honors / Awards

Undergraduate Research Fellowship – Cryptography Research project
CINSAM Scholarship and Poster Competition winner
Putnam and Virginia Tech Math Exam participant
ACM / TopCoder Programming Competition participant
Outstanding Senior in Mathematics and Computer Science