International Journal of Software Engineering and Knowledge Engineering

A FLEXIBLE STRATEGY FOR EMBEDDING AND CONFIGURING RUN-TIME CONTRACT CHECKS IN .NET COMPONENTS

STEPHEN H. EDWARDS

Department of Computer Science, Virginia Tech, 660 McBryde Hall (0106),
Blacksburg, Virginia 24061, United States of America
edwards@cs.vt.edu
http://people.cs.vt.edu/~edwards/

WESTLEY HAGGARD

Department of Computer Science, Virginia Tech, 660 McBryde Hall (0106),
Blacksburg, Virginia 24061, United States of America
wes@puzzleware.net
http://www.puzzleware.net/

Received (1 November 2006)
Revised (9 February 2007)
Accepted (9 February 2007)

In component-based systems, there are several obstacles to using Design by Contract (DbC), particularly with respect to third-party components. Contracts are particularly valuable when debugging or testing composite software structures that include third-party components. However, existing approaches have critical weaknesses. First, existing approaches typically require a component's source code to be available if you wish to strip (or re-insert) checks. Second, documentation of the contract is either distributed separately from the component or embedded in the component's source code. Third, enabling and disabling specific kinds of checks on separate components from independent vendors can be a significant challenge. This paper describes an approach to representing contracts for .NET components using attributes. This contract information can be retrieved from the compiled component's metadata and used for many purposes. The paper also describes nContract, a tool that automatically generates run-time checks from embedded contracts. Such run-time checks can be generated and added to a system without requiring source code access or recompilation. Further, when checks for a given component are excluded, they impose no run-time overhead. Finally, a highly expressive, fine-grained mechanism for controlling user preferences about which specific checks are enabled or disabled is presented.

Keywords: Design by Contract; assertion checkers; dynamic verification; component-based software; binary components; preconditions; postconditions; invariants; coding techniques; debugging aids; specification.

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, including costs for design, coding, testing and maintenance. These costs often can be reduced or

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 of introducing bugs.

To reduce the impact of these problems, component developers can use a more formal approach to documenting components. Bertrand Meyer's Design by Contract (DbC) approach[1,2] is a popular technique that seems to fit this problem naturally. DbC lays out a clear division of responsibilities between a component implementation and client code that uses it. A contract delineates what each party may assume and what each party is obligated to ensure.

Using the DbC approach, component developers can precisely and unambiguously specify the component interface by providing pre- and postconditions for each method and by providing invariant conditions for each class. Preconditions formally describe what the component expects to be true on entry to its methods—if these conditions are not met, 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—if these conditions are not met, then the component is to blame. Class invariants formally describe what must hold true about the state of a particular object after initial construction, as well as before and after every (public) 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 determine more easily if they are violating any preconditions, and would assure them that the component is doing what it claims, further decreasing the number of bugs.

1.1. *The Problem*

Most DbC approaches allow one to check conformance with a contract at run-time, usually through some form of assertion checking. Such run-time verification is a great tool during development, testing, and debugging, since it can help spot places where one component is calling another improperly, where glue code contains bugs, or even where the client has misunderstood the intended behavior of a component. In practice, DbC assertions are typically enabled during development and then disabled during release to eliminate the run-time penalty of executing assertions in the final product.

However, while the benefits of DbC for developers are well-understood, component-based development introduces new challenges. While the original developer of a component may wish to use DbC internally during his or her development activities, the

client who reuses a component is also a developer, working on an even larger system. Indeed, the primary limitations of most existing approaches to run-time contract checking impact not component developers but *component clients* when a component is distributed in compiled form only. Component developers understandably wish to maintain control over their source code. However, when assertion checks are embedded directly within the component being checked, as in most existing techniques, either the client must relinquish the benefits of assertion checks, or the client must pay some run-time penalty for the checks because there is no option to recompile without checks to eliminate this overhead.

Edwards *et. al* describe a set of requirements for any contract-checking strategy targeted at component-based software.[3] In short, these requirements suggest that to be viable in a component-based setting, any contract-checking approach should:

- Allow checking code to be inserted or removed without editing source code.
- Allow run-time checks to be selectively enabled or disabled for individual components or features.
- Avoid requiring recompilation of a component or the client's code to control run-time checking features.
- Allow the client to control which action(s) are taken in response to detected contract violations.
- Avoid requiring the client to use the same development tools used by the component developer in order to take advantage of the contracts.

The key problem addressed in this paper is how to obtain these goals for a given component framework. Because different component technologies use different methods to interconnect components, solutions to this problem may vary from one technology to another. Here, we focus on .NET components.

1.2. *Goals for a Solution*

The .NET framework provides both unique capabilities and unique restrictions that come into play in solving the problem of expressing component contracts. We describe a strategy and associated tool, called nContract, which provides configurable run-time contract verification without requiring component recompilation or source code access. More specifically, nContract addresses the problem through the following key features:

(1) Contract information is embedded in the binary version of the .NET component as metadata.
(2) Run-time checks can be added or removed without recompilation of either the component or the client's code.
(3) All checks can be enabled or disabled at the level of a .NET assembly, a class or even an individual method.

(4) Different classes of checks—preconditions, postconditions, exceptional postconditions, and class invariants—can be enabled or disabled individually at all levels of granularity.
(5) Little or no performance penalty occurs if checks are disabled.
(6) Custom actions can be performed when any contract violation is detected.

nContract allows component developers to formally specify .NET components using attributes. Figure 1 illustrates an overview of the key aspects of this strategy. The

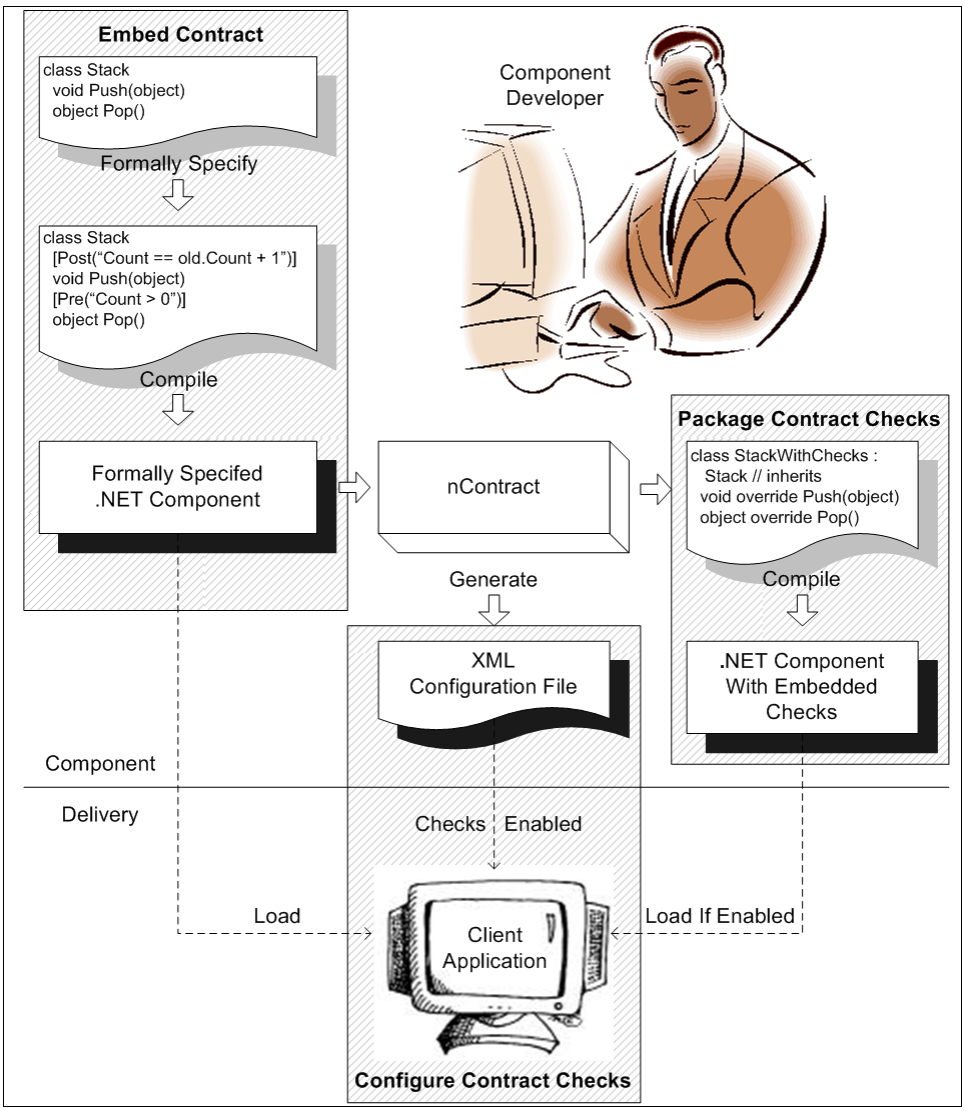

Fig. 1. General overview of how nContract embeds, packages, and configures contract checks.

embedded contract information is retrieved from the compiled component's metadata and a subclass can be generated for each type with a contract. This subclass is used as a container to package run-time checks for contract conformance. All members of the component's interface are overridden in the subclass 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.

1.3. *Organization of this Paper*

Section 2 describes related efforts to support run-time contract checking features. Section 3 explains our approach to embedding DbC contract descriptions directly in binary .NET components using metadata. This strategy allows contracts to be carried along in compiled components, ready for inspection or use through a standard API by a variety of tools. Section 4 lays out the nContract strategy for packaging run-time contract-checking code so that it can be added or removed in a design without requiring recompilation, even in situations where only the binary version of the component is available. Section 5 describes an innovative way to selectively enable or disable run-time checks at a fine-grained level, without imposing any additional performance overhead in performing the checks themselves. Section 6 provides an evaluation of the approach by comparing it to existing strategies and quantitatively assessing its performance impact. Finally, Section 7 summarizes our conclusions.

2. Summary of Related Work

A great deal of past work has been conducted on run-time assertion checking in general, and DbC-style contract checking in particular. Because a complete review is beyond the scope of this paper, here we briefly discuss the most relevant projects.

Eiffel[2] is one of the oldest and most well-known tools to support DbC. Eiffel allows contracts to be expressed directly in the language via relevant language constructs, and also can generate in-lined run-time checks of contract conditions within each class method. Removing checking code requires recompilation. More recently, Spec#[4] is a research language that adapts the same techniques to C#. Like Eiffel, it provides specific programming language constructs for describing contracts, supports generation of in-lined run-time checks inside methods, and requires recompilation to remove checking code from a binary component.

A number of other tools are "add-ons" to existing programming languages that allow developers to describe contracts using structured comments right in a component's source code. JML[5] and IContract[6] typify this strategy. JML is a behavioral specification language for Java, with an associated compiler that can generate run-time executable checks for (most) behavioral specifications in the resulting bytecode file[7]. The resulting class is one where the original method implementations have been renamed, helper methods implementing each assertion have been added, and the original method name is

used to define a "wrapper" method that calls checking helpers as well as the now-renamed underlying method. ContractJava[8] uses a similar strategy with "wrapper" methods, but focuses more on behavioral subtyping rules and assigning proper blame in the hierarchy chain if a contract violation occurs. IContract is a preprocessor for Java that can insert contract checks in-line in methods during compilation. All three tools require recompilation to remove checking code.

In addition to using embedded comments, some tools use alternate contract representations. XC#[9] is an extension to the C# compiler that supports compile-time attributes. It provides attributes for expressing preconditions and postconditions, and XC# can in-line the corresponding run-time checks in methods as part of the compilation process. Recompilation is necessary to remove checks.

Instead of inserting checking code during compilation, others have also investigated dynamically inserting checks. JContractor[10] provides a custom class loader for Java programs. It uses special naming conventions to identify regular methods that represent executable contract checks. When loading classes, if it identifies any such methods via its naming conventions, it dynamically rewrites the class bytecode to insert calls to the checks in the desired method bodies. Unlike compilation-based approaches, this allows one to insert or remove checks by choosing whether or not to use the custom class loader, without requiring recompilation. Handshake[11] also uses a custom class loader to dynamically insert checks into Java classes at load time. Unlike JContractor, however, Handshake represents contracts in a separate file with a special syntax. It then dynamically modifies class bytecode at load time to insert checks, if desired. Similar to JML's approach, Handshake renames the original method and then generates a replacement that includes run-time checks surrounding a call to the now-renamed original.

Aspect oriented programming (AOP) provides an alternative approach to adding contract checks to components. There are a number of possible ways to write an aspect that will instrument the code with assertions for contract checking. Unfortunately, most tools for implementing AOP, or "aspect weavers," work at compile-time only. Compile-time-only aspect weaving typically leads to a solution where one can enable or disable the checks at compile-time but not at run-time, usually with a requirement for source code access. Alternatively, more recent aspect weaving tools are beginning to employ dynamic weaving techniques that do not require source code access or recompilation. Such an approach would be a viable alternative to the strategy described in this paper. Unfortunately, although some weavers such as Loom.NET[12] hold promise, there are no dynamic aspect weavers for .NET that are mature enough to support the techniques needed for true source-less control over embedded contracts at this time.

The notion of storing contracts directly in components is also related to the notion of proof-carrying code.[13,14] However, with proof-carrying code, the idea is to attach a complete, machine-verifiable proof along with a component. Although techniques similar to those we propose could be used to embed proofs in components, here we are focusing purely on embedding behavioral contracts (or behavioral specifications).

For .NET components, the .NET contract wizard provides contracts for .NET components by creating Eiffel for .NET proxy classes.[15] It works by reading in a .NET assembly and listing all the types and methods from the assembly and allows for preconditions, postconditions and invariant checks to be entered as Eiffel expressions. These preconditions, postconditions and invariants are added to the Eiffel proxy class using Eiffel's language constructs. The tool then produces another .NET assembly containing all the Eiffel proxy classes. To enable assertion checks at run-time, the client would use the Eiffel proxy instead of the original class.

Two prior efforts by the author have focused on the same goals described here, but for different languages. Tan and Edwards[16] describe an approach to packaging JML-style run-time behavioral checks in separate wrapper classes, rather than including them in the bytecode of the original class. Their approach relies on a custom class loader to transform calls to `new` into factory method invocations at load time, and to integrate the wrapper class into the program's class hierarchy to ensure subtype substitutability. This approach is the one most closely related to the work reported here. However, due to the way .NET implements components, the bytecode editing and class loading techniques described by Tan and Edwards are not applicable in a .NET environment. Further, their work does not address enabling/disabling assertions efficiently down to the level of individual methods. Finally, their approach for Java relies on behavioral specifications embedded in the source code in the form of structured comments, and does not allow contracts to be retrieved from compiled assets. In C++, Edwards *et. al* describe how wrapper classes can be used to support insertion or removal of run-time checking code at link-time.[3] The C++ technique requires clients to use factories for object creation, but allows the client to either "link in" or omit supporting wrapper classes that contain contract checks. When wrapper classes are linked into the final application, they are automatically detected by the corresponding factory methods. When they are omitted from the linkage phase, factories produce unwrapped (unchecked) object instances. However, the C++ wrappers share many of the same limitations as the JML wrapper classes described by Tan and Edwards.

3. Embedding Behavioral Contract Descriptions in .NET Components

.NET components are self-describing; all the information needed to describe the component is stored as metadata in the compiled version. Further, .NET allows developers to add custom metadata to the component by defining their own custom attributes. Attributes are tags that can be added to code constructs such as classes, methods, parameters and other constructs where one might want to store custom information. Here is a simple C# code example that has a *FormallySpecified* attribute attached to the class and a *Pre* attribute attached to the method (attributes have been italicized for emphasis):

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

.NET provides a reflection API that allows programmatic retrieval of component metadata at run-time. By using attributes to store the DbC contract information, the contract can be retrieved from the binary component, eliminating any need for source code access. Furthermore, precise written documentation can be generated by pulling the contract information directly out of binary components.

nContract provides a set of attributes that are used to store the DbC contract information, such as preconditions and postconditions for methods. These conditions are expressed in the form of a string parameter to the corresponding attribute. 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 take the form of C# boolean expressions (other .NET languages could be used for the expressions if the templates were changed to that particular language). Figure 2 illustrates a small C# component that contains a contract expressed in this form, a `CharBuffer`.

nContract uses the following contract attributes (complete details and a full example are provided by Haggard[17]):

- **`FormallySpecified`** marks a class as being formally specified.
- **`Pre`** provides a condition which needs to hold true on the entry of a method.
- **`Post`** provides a condition which needs to hold true on the exit of a method.
- **`ExceptionalPost`** provides a condition which needs to hold true whenever the specified exception is thrown from a method.
- **`Invariant`** provides a condition which needs to hold true after an object's creation and before and after every public method call for that object.
- **`RepresentationalInvariant`** is the same as invariant except it is not part of the public specification. Its purpose is for a developer to provide invariants which reference internal data and can be verified at run-time by nContract.
- **`ModelField`** indicates an abstract field a developer can use for public specification which allows them to change implementation details later without breaking the public specification.
- **`Pure`** marks a method as having no side effects. Any method that is used in any of the condition expressions needs to be marked as pure.

Note that operator `@` appearing in some assertions in Figure 2 is a C# operator that alters the parsing of string literals to allow multi-line strings and to ignore escape

```
[FormallySpecified]
[ModelField( typeof(List<char>), "Contents",
    "new List<char>(this.ToString().ToCharArray())" )]
[RepresentationalInvariant(
    "numberOfChars == stringBuilder.Length" )]
public class CharBuffer
{
    [Pre( "value != null" )]
    [Post( "Contents.Count == value.Length" )]
    protected CharBuffer( 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 )
    {...}

    ...

    // Member fields
    protected StringBuilder stringBuilder;
    protected int numberOfChars;
}
```

Fig. 2. An example C# class containing an embedded contract description.

sequences. It is used here purely to allow assertions—which are expressed as string literals—to span multiple lines.

At this point, it is also important to note that the purpose of designing nContract was to explore ways of embedding contracts in binary components, and using this embedded information to overcome the need for recompilation or source code access. Other researchers have investigated alternative ways of specifying behavior and alternative languages for expressing such specifications. While the choice of C# expressions as the means to write assertions in nContract is expedient, it is by no means the only choice. It would be equally feasible for one to choose a completely different contract specification language, together with a corresponding run-time checking code generator, to achieve the same basic result.

The only notable aspect of nContract's approach to describing assertions is that it provides explicit support for *model-based specification* of contracts. While many DbC approaches, beginning with Eiffel, require one to describe a component's contract directly in terms of its attributes or methods, such a strategy may require contracts to be too implementation-specific, and may require the client to understand the implementation in order to understand the contract. With a model-based specification[18] however, one can define an abstract model of an object's state, and define the contract in terms of the abstract model rather than the concrete implementation. nContract's `ModelField` attribute is used to introduce an abstract model of a component's state, together with an abstraction function that describes how to compute the abstract state from the component's concrete implementation. Figure 2 illustrates an abstract state model, where a model field called `Contents` is introduced at the start of the class declaration. This model field is defined to be a `List<char>`, which is a simple way to understand its contents. The `ModelField` attribute for `Contents` also says how the class' concrete implementation maps into this abstract view. The behavior of methods can then be described in terms of this abstract view.

4. Packaging Contract Checks for Use in Testing

The embedded contract information described in Section 3 ensures that the contract is always available, whether or not source code is provided to clients. However, for contracts to be of practical benefit to clients during their development activities, they should be able to turn on run-time checking. However, for a binary-only .NET component, clients cannot easily insert or remove code from the assembly itself. Instead, the nContract strategy is to retrieve the contract description from the compiled component's metadata, and then generate a subclass for each type with a contract. These subclasses are used as containers to package run-time checks for contract conformance. All members of the component's interface are overridden in the subclass and contract assertions are wrapped around calls to the base class. The component client can then use a factory to create instances of the component's types, allowing the decision of whether or not to create assertion-checked or unchecked objects to be deferred until run-time, and changed without recompilation.

For every formally specified type in a given .NET component, nContract uses a template to generate a subclass for that type. For example, a subclass called `CharBufferWithChecks` would be generated for `CharBuffer`, from Figure 2. Every `public` or `protected` virtual method or property from the base class is overridden in the subclass and assertion checks are inserted around calls to the base method or property. Figure 3 outlines this subclass for the `CharBuffer` example.

The `Insert()` method in Figure 3 illustrates how checking code is inserted in the subclass. The overriding definition in the subclass determines whether specific checks for the class invariant, the method precondition, the method postcondition, and any separate postconditions for possible exceptions are enabled, and carries out the

corresponding assertion checks when necessary. Placeholders marked with double-angle brackets (*«...»*) represent assertions taken directly from the embedded contract

```
public class CharBufferWithChecks : ExampleComponent.CharBuffer
{
  ...
  public override void Insert(
    System.Int32 index, System.String value ) {
    if ( config.Insert_Int32_String.InvariantDisabled == false )
        config.CheckEntryInvariant( «InvariantCondition» );

    if ( config.Insert_Int32_String.PreDisabled == false )
        config.CheckPrecondition( «Precondition» );

    try {
        base.Insert( index, value );
        if ( config.Insert_Int32_String.PostDisabled == false )
            config.CheckPostcondition( «PostCondition» );
    }
    catch ( Exception ex ) {
        if ( config.Insert_Int32_String.ExceptionalPostDisabled
             == false )
        {
            if (ex is System.ArgumentOutOfRangeException )
            {
                System.ArgumentOutOfRangeException excep =
                    ex as System.ArgumentOutOfRangeException;
                config.CheckExceptionalPostcondition(
                    «ExceptionalPostcondition» );
            }
        }
        throw;
    }
    finally {
        if ( config.Insert_Int32_String.InvariantDisabled
             == false )
            config.CheckExitInvariant( «InvariantCondition» );
    }
  }
  ...
  private CharBufferConfiguration config;
}
```

Fig. 3. The assertion checking subclass generated for `CharBuffer`.

specification for the corresponding method or class. The subclass method implementations use the `config` data member to determine which specific checks should be executed. The `config` data member refers to a singleton instance of a class that contains these configuration settings for the subclass. nContract automatically generates such a configuration class for each checking subclass in order to implement the checking behavior controls described in Section 5.

Any properties defined in the base class are also overridden in the checking subclass using a similar template. The only other items that are included in the subclass template are properties that represent the model fields declared in the contract and a reference to the associated configuration class (i.e., the `config` variable in Figure 3).

Because of C#'s subtyping rules, an instance of `CharBufferWithChecks` can be used in any context where a `CharBuffer` object is expected. A `CharBufferWithChecks` object will behave identically to a `CharBuffer` object, but with run-time checks of contract conformance performed dynamically. During development, one can use `CharBufferWithChecks` objects, and switch to completely unchecked objects for deployment so that all object code and run-time overhead associated with the subclasses can be completely avoided.

However, to achieve this end, it is also critical to address how clients create new objects. There are two specific goals we want to achieve for object creation in client code. First, we want to be sure that client code need not change, or even require recompilation, when one wants to switch from using checking subclasses to using unchecked base classes, and vice versa. Second, we also want to minimize the costs associated with run-time checks when they are completely suppressed in an application. To achieve these goals, we can rely on the *factory method* design pattern,[19] which allows us to completely decouple the client from decisions about which concrete class is created when a new object is needed. If the client uses factory method calls to create all new instances from a component, rather than directly calling the `new` operator, we can achieve our two goals regarding object creation.

To support this scheme, for every `public` or `protected` constructor in the class being checked, nContract generates within the checking subclass a corresponding constructor and a corresponding factory method with the same parameter profile. The factory method allows for any necessary preconditions to be checked before the constructor is invoked, and it also allows for any exceptional postconditions on the constructor to be checked. Figure 4 shows the simplified structure of these features in the `CharBufferWithChecks` example.

Note that the `Create()` factory method provided in Figure 4 only allows one to create instances of the checked subclass. Using this factory method to create checking subclass instances is necessary in order to properly check contract requirements on the corresponding constructor. Of course, this is not the factory method we want clients to use to create objects, since clients should write their code without directly referring to any contract checking features.

Instead, the original component developer must decide on a factory method scheme for clients to use when creating instances of classes from the developer's component. The strategy that requires the least amount of effort is to use a generic factory that works

```
public class CharBufferWithChecks : ExampleComponent.CharBuffer
{
  ...
  protected CharBufferWithChecks( string value ): base( value )
  {
    if ( config.ctor_String.PostDisabled == false )
        config.CheckPostcondition( «Postcondition» );

    if ( config.ctor_String.InvariantDisabled == false )
        config.CheckExitInvariant( «InvariantCondition» );
  }

  public static ExampleComponent.CharBuffer Create(
    string value )
  {
    if ( config.ctor_String.PreDisabled == false )
        config.CheckPrecondition( «Precondition» );

    CharBufferWithChecks newObject = null;
    try {
        newObject = new CharBufferWithChecks( value );
    }
    catch ( Exception ex ) {
        if ( config.ctor_String.ExceptionalPostDisabled
             == false ) {
            if ( ex is ExceptionType ) {
                ExceptionType excep = ex as ExceptionType;
                config.CheckExceptionalPostcondition(
                    «ExceptionalPostcondition» );
            }
        }
        throw;
    }
    return newObject;
  }
  ...
}
```

Fig. 4. The constructor and factory generated in the assertion checking subclass.

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

Fig. 5. A developer-provided factory method.

for all classes. nContract provides an assembly called `ContractSpecification` that includes a generic `Factory<T>` class. The Factory<T> class provides a public factory method called `Create()` that provides for generic object instantiation[17]. `Create()` takes a variable number of arguments and uses reflection to create an instance of the checking subclass or the original base class, depending on configuration settings.

The use of a generic factory provides the least burden on the component developer, although the use of reflection does impose an extra run-time cost. Component developers who wish to avoid this cost can provide their own custom factory methods in the underlying classes. This provides a statically type-checked signature for factories, and allows one to avoid using reflection when run-time contract checks are completely disabled for a class. Figure 5 illustrates such a custom factory method, which could be added to the `CharBuffer` class in Figure 2. This custom factory simply uses the `new` operator when checks are completely disabled for the class in question, but delegates object creation to the generic `Factory<T>` class if any checking features are enabled for the given class.

Using factories for object creation, when combined with checking subclasses generated directly from embedded contracts, allows nContract to completely separate the client's code from any dependencies on checking subclasses, and eliminates any hard-coded decisions about whether assertion checks are present. While this covers some of the goals outlined in Section 1.2, achieving the remaining goals requires an alternative approach to managing configuration choices about how checking features are enabled, disabled, and controlled.

5. Configuring Run-time Checking Features

To provide appropriate flexibility to clients, nContract gives them the ability to individually configure precondition, postcondition, exceptional postcondition and

invariant checks at the assembly, class and method level. Rather than using compile-time options, or using a single global enable/disable setting, nContract uses a separate configuration object for each subclass containing run-time checks, an approach used successfully in prior work.[3,16] Thus, nContact uses a template to generate a custom configuration class for every formally specified class. Figure 6 shows an example of such a class for `CharBuffer`. Each subclass that implements run-time checking features contains a field, `config`, which refers to the singleton instance of the corresponding configuration class.

Each generated configuration class encapsulates all of the options controlling run-time checking for the underlying class. For example, the `CharBufferConfiguration` class in Figure 6 manages all of the options involved in controlling run-time checking features for the `CharBuffer` class in Figure 2.

When nContract generates a configuration class, it generates a field for every constructor, method and property to hold the settings for the individual check types (preconditions, postconditions, invariants, or exceptional postconditions). This allows these configuration settings to be directly referenced in the generated subclass as shown in Figure 3, which provides an efficient way to determine if a particular check type is disabled for a particular method. At the same time, since all checking subclasses for a given type will share a reference to the same instance of their configuration class, there is only a single place to change these options. The configuration class also inherits common methods from the `ContractSpecification.ClassConfiguration` base class that enable it to report contract violations in a systematic way.

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

    public ContractSpecification.MethodConfiguration ctor_Void;
    ...
    public ContractSpecification.MethodConfiguration
        Insert_Int32_String;
    ...
}
```

Fig. 6. The automatically-generated configuration class for controlling checking features on `CharBuffer`.

While only one instance of each configuration class is created at run-time, the creation (via loading an external XML file) is handled by the nContract infrastructure. As a result, the `CharBufferConfiguration` class in Figure 6 does not follow the singleton design pattern directly; its constructor is public, so that nContract can create an instance from XML data, and its static `GetConfig()` method looks up this instance using nContract's template factory infrastructure.

While the structure of the configuration class shows how checking options can be tested efficiently at run-time, it is also important to provide a way for clients to control these options without affecting their code. nContract uses XML files that are loaded at program startup to control these options.

After nContract generates the checking subclasses and configuration classes for a given assembly, it then compiles them into another .NET assembly. This second .NET assembly is loaded and an instance of each configuration class is created. These configuration classes are then serialized to XML files to control the settings for that assembly. Figure 7 shows the serialized form of the configuration class. Here is the XML configuration file that corresponds to the `CharBufferConfiguration` configuration class.

Now configuring the checks becomes as simple as modifying this XML file. For example if one wishes to disable all checks for the `CharBuffer` class, just set the `Enabled` attribute on the corresonding `ClassConfiguration` tag to false. This action will instruct the factory methods to create instances of the original class, rather than instances of the checking subclass. If checks are enabled instead, one can enable or disable specific kinds of checks on a per-method basis in a similar way. These settings are reloaded from the XML file on program startup.

While XML files may not provide the most human-friendly interface, they do provide a representation for the options that can be processed easily by other tools. For example, a graphical control panel that presents a hierarchical view of contract checking settings for an entire application is easy to devise, a technique suggested by Tan and Edwards[16] and originally inspired by the IControl tool for IContract[20]. Such a tool would allow the client to easily turn on or off specific features, and simply record the desired changes in the relevant XML files. Tan and Edwards also suggest a strategy for making such changes to run-time checking options at run-time using the same form of interface.

Finally, while the custom configuration classes handle the problems of how the client enables or disables contract checking features without requiring source code access to the original component, these classes do not address what action(s) take place when a contract violation is detected. Having the ability to execute customizable actions on failed assertions can be a big advantage to the client. When a contract violation occurs during development and debugging, different developers may want different actions to be taken. Some may want the violation to be logged, to throw an exception, to start the debugger, or simply to display a message. Allowing for customizable actions gives the developer more freedom to handle the violation in their own way.

```
<?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_String PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    ...
    <Insert_Int32_String PreDisabled="false" PostDisabled="false"
      ExceptionalPostDisabled="false" InvariantDisabled="false" />
    ...
  </ClassConfiguration>
</ArrayOfClassConfiguration>
```

Fig. 7. The XML file that controls the checking features for `CharBuffer`.

To address this need, nContract takes advantage of the .NET event model by defining a static public event called `OnAssert`, and then signaling this event whenever a contract violation is detected. As a result, a developer can create any number of custom assert handlers and subscribe them to this event. The original component developer can choose whatever handler(s) are appropriate in that context, and the component client can make completely different choices. This arrangement allows complete freedom to both parties. In contrast, other tools typically hardcode a specific action to be taken for contract violations, with the most common action being to throw an exception of some sort to terminate the application.

6. Evaluating nContract

This section discusses the limitations and tradeoffs involved in using this scheme in Section 6.1. It then assesses the strengths and impact of nContract in two ways. First, Section 6.2 presents a feature by feature breakdown and comparison with the most common tools and techniques used for run-time contract verification. This comparison focuses primarily on how the contracts are provided or packaged and on how the run-time checks are configured. Second, Section 6.3 presents the results of run-time performance testing. 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 into the performance impact of this approach.

6.1. *Limitations*

nContract relies on subclasses and type substitutability to make unchecked classes and their contract-checking counterparts interchangeable. Using subclasses to package run-time checking code introduces some specific design restrictions that a component developer must follow, however:

(1) In order to create a checking subclass that inherits from a given base class, the base class cannot be `private`, `sealed`, or `static`.
(2) All the constructors that are to be checked at run-time must be `public` or `protected`, since it is necessary to access them from the generated subclass. Further, to ensure that the client uses a factory method instead of calling the `new` operator directly, we recommend that the component developer make all the constructors `protected`.
(3) All methods or properties that will be checked at run-time must be `public` or `protected`. This is a reasonable restriction, since private features within a component do not participate in its contract with the client. In addition, however, methods or properties that will be checked must also be overridable, since that is the technique that will be used to add checking features. 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's 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.

Also, the approach detailed here is founded on the idea that the original contract is embedded in the binary (compiled) version of a component. As a result, changing the contract for a given component requires recompilation. However, one can always add new contractual requirements by creating a subclass, which inherits the contract of its parent and then extends the behavioral requirements by adding more assertions.

6.2. *Comparison to Other Contract Checking Strategies*

Table 1 summarizes how nContract compares with the various contract checking tools discussed in Section 2. There are five principal dimensions in the comparison, with a number of possible alternatives for each.

First, Table 1 indicates the means by which contracts are expressed when using different tools. Many tools rely on encoding contracts in the form of structured comments within a component's source code. Others instead offer special language

Table 1. A feature comparison of contact checking tools.

Feature	C++ Wrappers	ContractJava	Eiffel	Handshakee	IContract	JContractor	JML	nContract	Spec#	Tan-JML	XC#
Contract storage technique:											
Structured comments	✓	✓			✓		✓			✓	
Language construct			✓						✓		
Separate file				✓							
Naming convention						✓					
Metadata								✓			✓
Implementaton strategy:											
In-line assertions			✓		✓	✓			✓		✓
Separate methods		✓		✓			✓				
Separate classes	✓							✓		✓	
Configuration choices fixed at:											
Compile-time		✓	✓		✓		✓		✓		✓
Link -time	✓										
Load-time				✓		✓		✓		✓	
Granularity for controlling checks:											
Only global control		✓		✓		✓	✓				✓
For packages/assemblies	✓		✓		✓			✓	✓	✓	
For individual classes	✓				✓			✓	*	✓	
For individual methods					✓			✓	*		
Customizable assertion actions	✓							✓			

constructs for defining contracts. One tool represents contracts in a separate form using special syntax, and another uses a defined naming convention so that executable checks can be embedded in a class or component using regular methods. Finally, nContract and one other tool use metadata to encode contracts in a form that is directly accessible in the compiled component.

Second, Table 1 divides the implementation strategies for performing run-time checks into three categories. Most existing tools simply in-line checking code right in the methods being checked. Except for Java-specific approaches that use custom class loaders, this implementation strategy usually requires one to recompile a component in order to remove checks. Other tools instead embody run-time checks in their own helper methods, rename each original method, and replace each with a "wrapper" that surrounds a call to the renamed original with appropriate calls to checks. Three approaches, including nContract, place all checking code in separate classes, and use factory methods to control the kind of concrete object client code receives.

Third, Table 1 indicates when choices about the inclusion of run-time checking code are made. Many approaches make this choice at compile-time, forcing one to recompile in order to remove or re-insert checking code. This practice is unacceptable for

components distributed in binary form. One approach for C++ makes the choice at link-time. Several others make the choice at load-time.

Fourth, Table 1 indicates the granularity at which run-time checks can be controlled. Many approaches only allow global control, possibly by requiring one to "compile in" or "compile out" checks in order to enable or disable them. However, other tools allow preconditions, postconditions, and invariant checks to be turned on or off on a per-package or per-assembly basis. Most such tools also allow these categories of checks to be turned on or off for individual classes as well. Spec# is the exception: while it allows assembly-level control over all forms of checks, it only allows invariant checks to be enabled or disabled at the level of individual classes or methods. IContract and Spec# are the only tools besides nContract to allow control of assertions at the level of individual methods. However, both IContract and Spec# generate checks in-line at compile-time, requiring recompilation to remove checks. nContract is the only tool in this comparison that allows method-level control while also avoiding the need for source-code access.

Fifth, Table 1 indicates whether or not the various approaches allow one to provide user-configurable response actions to take when contract violations are discovered.

6.3. *Performance Impact*

In terms of execution time, nContract has a performance impact in three areas. First, because nContract uses method overriding to insert checks in generated subclasses, method dispatch must necessarily be virtual using dynamic binding, which will impose a penalty on method calls. Second, object creation through a factory method instead of through direct calls to the `new` operator will impose some penalty for object creation. Third, checking to see if specific contract assertions are enabled and performing the corresponding checks will add a penalty to each method that is invoked. We have examined these three issues separately through performance experimentation.

First, consider the impact of using virtual methods. To assess the size of this impact, two versions of the `CharBuffer` 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, which is similar to `Insert()`). Test runs of 100 million calls to the method under consideration were made and timed. This test was repeated 10 times on a 3GHz Pentium 4 machine with 1Gb of RAM, and the results were averaged.

Figure 8 summarizes the results. The difference in execution time for the non-empty method calls is less than 1%. While there is a significant difference between the virtual and non-virtual costs for a completely empty method, the size of this difference is negligible when compared to the cost of a realistic, non-empty method. Therefore any reasonable method that does some amount work will not suffer much from being a virtual method.

Second, consider the impact of factory methods rather than the `new` operator. Figure 9 summarizes the results obtained by timing the various object creation techniques discussed in this paper. For comparison, direct calls to new for the same class averaged

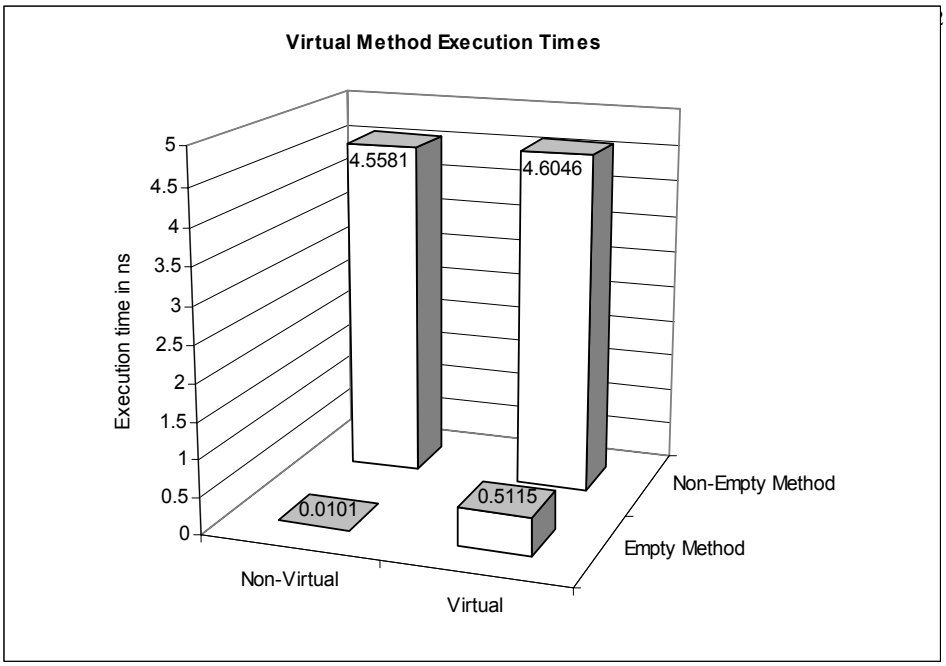

Fig. 8. A comparison of times of virtual vs. non-virtual and empty vs. non-empty method calls.

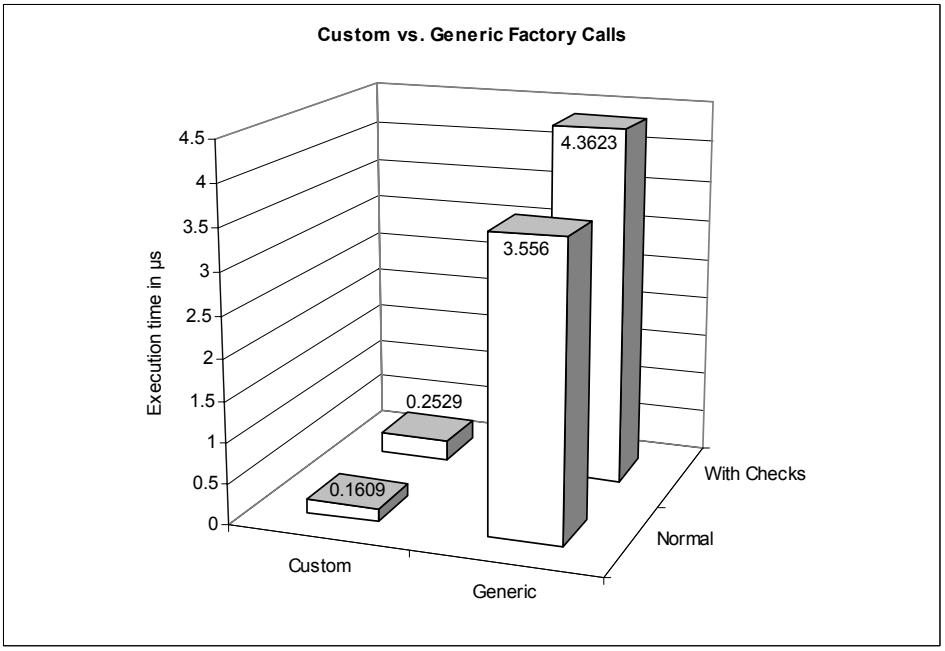

Fig. 9. A comparison of times taken to create an instance of an object using different creation methods.

0.0962 μs. As expected, there is a significant penalty involved in using the generic factory method, which use reflection-based techniques. The generic factory method for creating normal object (i.e. without embedded checks) uses the `Activator.CreateInstance()` method to create new objects. When checks are desired, it must use the checking subclass' factory method to ensure that desired checks are performed during object creation. This involves dynamically looking up the sublcass type, and then dynamically invoking the correct static factory method. Although nContract uses caching of the lookup results, the process is still noticeably more expensive than otherwise.

Alternatively, if the component developer decides to provide custom factory methods in the original component itself, substantial improvements can be made. Such custom factory methods do not require reflection. As shown in Figure 9, the result is an execution time similar in scale to direct calls to `new`. On this basis, we recommend that developers write their own factory methods where possible, rather than requiring clients to use the generic factory methods. Instead, the generic methods are a general-purpose fallback that can be used in cases where the component developer has not provided built-in factory methods.

Third, consider the impact of testing configuration settings to determine if checks are enabled, and executing the corresponding checks. To examine this issue, we measured the time taken to execute methods from the original class, the checking subclass with all checks enabled, and the checking subclass with all checks disabled. To best reflect typical class usage for our example component, each measurement run included one million executions of a sequence of actions that included a call to the `Insert()` method, a call to the `Remove()` method, and two accesses of the `Length` property, all in the `CharBuffer` class. The measurement run was repeated 10 times and the results were averaged, providing the execution time summary shown in Figure 10.

In Figure 10, the "Original" bar represents the average execution time for the given sequence of method calls on the original base class (no checks present at all). Other than the added cost of the use of virtual methods, this represents the "full speed" execution of the underlying code. In comparison, the checking subclass imposes an extra level of method call overhead for each method invoked, plus the cost of testing to see if each check is enabled, plus the cost of executing any enabled checks. With all checks disabled, Figure 10 indicates the checking subclass runs an average of 49% slower than the original class. Since the client can specifically enable or disable the use of checking subclasses on a per-class basis, the client can choose whether or not to incur this cost at any point during development of a larger application. Figure 10 also indicates that, with all checks enabled, the checking subclass is significantly slower. This additional penalty is due only to the cost of performing the checks themselves, and not to any infrastructure aspects of the overall strategy used here. The results reported here are similar to those reported for C++.[3] Typically, invariant checks and postcondition checks impose the greatest burden. Again, however, because the client can choose whether or not to inable invariant and postcondition checks at a fine-grained level, without requiring

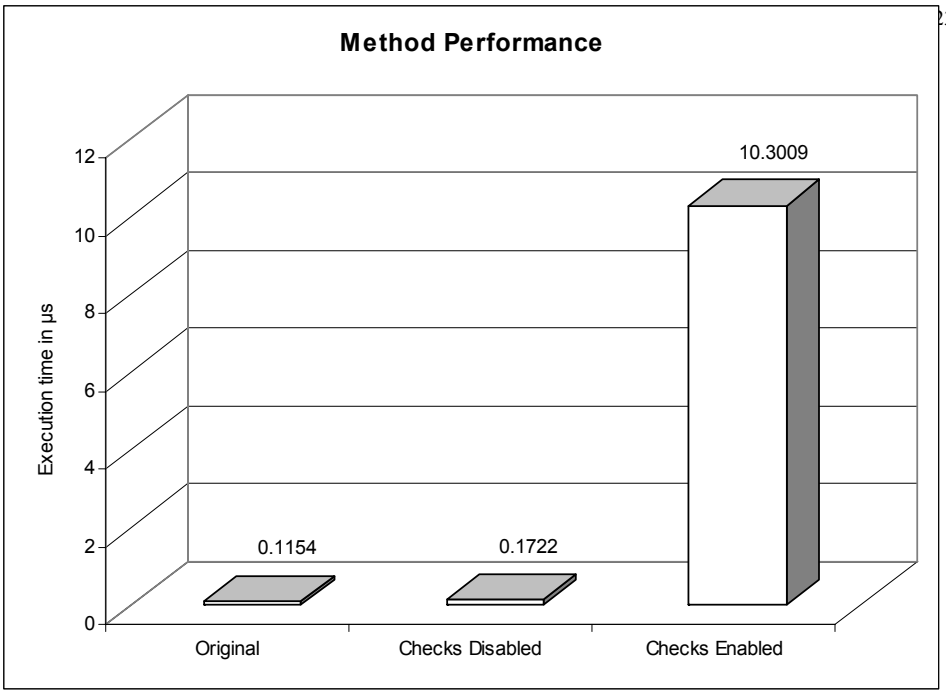

Fig. 10. A comparison of execution times when checks are enabled, disabled, or removed entirely.

recompilation or source code access, it is much easier for these features to be exercised in a controlled way when developing larger systems using third-party components.

7. Conclusions

Configurable run-time contract verification can be 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. It also helps clients 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. However, techniques that require one to have access to component source code in order to recompile it with different options are not viable in a component-based marketplace. Further, the additional overhead imposed by such checks if they were left in place in the binary versions of components that vendors provide to their customers is a strong disincentive.

If we want component clients to receive the benefits of contract checking in their own development activities, however, these problems must be overcome. nContract describes a strategy for overcoming these problems for .NET components, as typified by C# assemblies. Similar techniques could be applied in other .NET languages easily. Extrapolation to other component technologies is also possible, but may require more effort.

nContract addresses the problems involved in using run-time contract checking with .NET components by embedding contract information in the compiled component's metadata. This metadata can be extracted and processed, both to produce human-readable documentation and to generate run-time-checking subclasses. Such a checking subclass is used as a container to package run-time checks for contract conformance. All members of the component's interface are overridden in the subclass 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.

Using nContract allows component developers and clients to get the benefits of run-time contract verification with requiring the original vendor to provide source code, and allowing both parties to avoid nearly all the run-time costs imposed by contract checking strategies when those features are unneeded. Other researchers interested in investigating the implementation of nContract can download it electronically[21] or can visit the nContract forum on-line.[22]

References

1. B. Meyer, Applying 'Design By Contract', *Computer*, **25**:10(1992) 40-51.
2. B. Meyer, *Object-Oriented Software Construction*, 2nd edn. (Prentice Hall PTR, Upper Saddle River NJ, 1997).
3. S.H. Edwards, M. Sitaraman, B.W. Weide and J. Hollingsworth, Contract-checking wrappers for C++ classes, *IEEE Trans. Softw. Eng.*, **30**:11(2004), 794-810.
4. M. Barnett, K. Rustan, M. Leino, and W. Schulte, The Spec# programming system: an overview, (2004), http://research.microsoft.com/SpecSharp/papers/krml136.pdf.
5. G.T. Leavens, A.L. Baker, and C. Ruby, JML: a notation for detailed design, in *Behavioral Specifications of Businesses and Systems*, eds. H. Kilov, B. Rumpe, and I. Simmonds (Kluwer, 1999), pp. 175-188.
6. R. Kramer, iContract—the Java™ Design by Contract™ tool, in *TOOLS '98: Pro. Technology of Object-Oriented Languages and Systems*, (IEEE CS Press, 1998), p. 295-307.
7. Y. Cheon, *A Runtime Assertion Checker for the Java Modeling Language,* TR #03-09 (Dept. of Computer Science, Iowa State University, 2003).
8. R.B. Findler, M. Latendresse, and M. Felleisen, Behavioral contracts and behavioral subtyping, in *Proc. 8th European Software Engineering Conf./ 9th ACM SIGSOFT Int'l Symp. Foundations of Software Engineering*, (ACM Press, 2001), pp. 229–236.
9. eXtensible C#, (2005), http://www.resolvecorp.com/Products.aspx.
10. M. Karaorman, U. Holzle, and J. Bruno, jContractor: a reflective Java library to support design by contract, in *Proc. Meta-Level Architectures and Reflection, 2nd Int'l Conf., Reflection '99*, LNCS #1616, (Springer Verlag, 1999), pp. 175-196.
11. A. Duncan and U. Hölzle, *Adding contracts to Java with Handshake*, TRCS98-32 (Univ. of California at Santa Barbara, 1998), http://www.cs.ucsb.edu/research/trcs/abstracts/1998-32.shtml.
12. Loom.NET, (2007), http://www.dcl.hpi.uni-potsdam.de/research/loom/.
13. G.C. Necula, (1997), Proof-carrying code, in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, (ACM Press, 1997), pp.106-119.
14. A.W. Appel, (2001), Foundational proof-carrying code, in *Proc. 16th Ann. IEEE Symp. Logic in Computer Science*, pp.247-256.

15. K. Arnout and R. Simon, The.NET Contract Wizard: adding design by contract to languages other than Eiffel, in *Proc. 39th Int'l Conf. Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, (IEEE CS Press, 2001), pp. 14-23.
16. R.P. Tan and S.H.Edwards, An assertion checking wrapper design for Java, in *SAVCBS 2003: Specification and Verification of Component Based Systems*, TR #03-11, (Dept. of Computer Science, Iowa State University, 2003), pp. 29–34, http://www.cs.iastate.edu/~leavens/SAVCBS/2003/papers/full-papers/tan-edwards.pdf.
17. W. Haggard, *nContract—Creating Configurable Run-Time Contract Verification for .NET Components*, M.S. Thesis, (Dept. of Computer Science, Virginia Tech, 2005), http://puzzleware.net/download.aspx?file=/nContract/nContract-Thesis.pdf.
18. J.M. Wing, A specifier's introduction to formal methods, *Computer*, **29**:9(1990), 8-24.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1995).
20. iControl, (2005) http://icplus.sourceforge.net/iControl.html.
21. W. Haggard, nContract source code download, http://puzzleware.net/download.aspx?file=/nContract/nContractSourceCode.zip
22. W. Haggard, nContract forum, http://puzzleware.net/forums/11/ShowForum.aspx.