JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 7, September – October 2005

Support for Design by Contract™ in the C# Programming Language

Rachel Henne-Wu, California State University, Sacramento
Dr. William Mitchell, California State University, Sacramento
Dr. Cui Zhang, California State University, Sacramento

Abstract

There is evidence that "contracts," or assertion techniques involving preconditions, postconditions, and invariants, have a positive effect on overall software quality. Regrettably, very few programming languages support these techniques. Since the advent of Bertrand Meyer's Design by Contract™ method, introduced in the language Eiffel, a number of systems have been built to implement support for contracts in more commonly-used languages. Such support has not been satisfactorily implemented in C#. In this paper, we compare the different approaches of existing systems and introduce Contract Sharp, a tool that provides support for contracts in C#.

1 INTRODUCTION

From the time the first large computer programs were written, there have been efforts to improve software reliability. One informal approach to program reliability, Meyer's Design by Contract™ method (DBC), originated in the programming language Eiffel [16]. Since that time, the need for more reliable software has heightened interest in support for "contracts,"consisting of assertions that must be true for methods and classes, in other more commonly-used languages. In particular, assertions have not been satisfactorily implemented in C#. Because of its centrality to Microsoft's .NET Framework and its support for interoperability with other languages, the C# programming language is likely to become an important, widely-used language. As a result, support for contracts in C# is quite likely to be in demand in the coming years.

In this paper, we introduce "Contract Sharp," a tool that provides support for contracts in C#. Section 2 of this paper discusses the importance of contracts and surveys the various approaches to implementing support for contracts. Section 3 presents the approach taken by Contract Sharp, how it compares with the other approaches, and its functionality and architecture. Section 4 describes how it measures up to established criteria.

Cite this article as follows: Rachel Henne-Wu, William Mitchell, Cui Zhang: "Support for Design By Contract™ in the C# Programming Language", in *Journal of Object Technology*, vol. 4, no. 7, September-October 2004, pp. 65-82 http://www.jot.fm/issues/issue_2005_09/article2

2 BACKGROUND AND RELATED WORK

The Importance of Contracts

Contracts, or assertions, are an element of class composition that not has been deployed widely in commonly used programming languages. Pioneering usage by Hoare, Floyd and Dijkstra appeared in the 1960s as a mechanism for reasoning about concurrent systems. Because assertions are Boolean expressions, they can be regarded as another aspect of class design, and also can be extremely helpful in design and test/debug activities (regarding analysis and design [10], and regarding testing [4]). Many methodologies have been applied to the problem of finding and correcting defects in source programs. Correctness proofs currently have a limited place in practice, are very human intensive, and have not scaled to large programs. Inspections are a wide-focus technique with high yield but need high skill. Nevertheless, they have been valuable on some large-scale projects [23]. Lastly, there is the time-honored testing approach, which has always been the fallback technique. However, even with increasingly sophisticated automatic test suites, testing will not detect all errors of interest to the developer. When combined, all above correctness methodologies are capable of discovering almost all (98% of) defects (but with prohibitive cost in almost all projects). Assertions have the potential for reducing effort in test suites and for some systems can make less-expensive random generation testing techniques feasible [4]. Thus, assertions offer a practical compromise between the admitted difficulties of formal verification of correctness and the need for other low cost, high yield techniques. Increased use of assertions is manifested by various design systems that use them, such as UML's <<invariant>>, <<precondition>> and <<postcondition>> stereotypes [20]. In the interests of a balanced treatment, efficiency of runtime assertion checking is an issue, with reports of 25-100% overhead, depending on check settings for various assertions [17] as well as the nature of a given application. This is because well-written class clusters and libraries exhibit relatively small average routine size, and the assertion code for a typical routine might constitute 15-30% of a routine's total LOC.

To the authors' knowledge, no comprehensive study has established generally useful cost/benefit conclusions concerning assertion usage. However, we review here one well-documented experience that demonstrates their viability. IBM's development of the OS/400 operating system for the AS/400 series midrange RISC architecture is summarized (from an assertions viewpoint) in Table 1 [3]. Average class size is 143 LOC and average routine size is 22 LOC, explaining to some extent the runtime costs of assertion evaluation.

Size of delivered code (C++)	2,000,000 LOC
Number of classes	14,000
Number of methods	90,000
% of test code in form of assertions	40%
Assertion effectiveness	Assertions found 14 of the 18 most subtle bugs
Assertion use feasibility	Their cost has, so far, been a non issue
Development activity timeframe	1992-94
Conclusion on assertion usefulness	Helpful for class design, testing; evaluation ongoing

Table 1: Assertions and the OS/400 development project

The importance and use of assertions/contracts in software development is growing. They can be assimilated into complete OO implementations of inheritance, generic classes and exception handling, and thus have a role in quality and reuse. Assertions augment V&V and testing, and are a simple yet powerful form of correctness coding. They also serve as source code documentation and capture some of the essential meaning of an OO class specification.

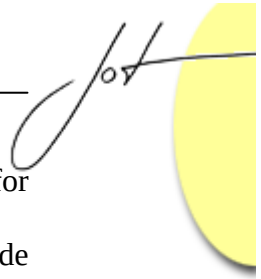
Existing Approaches to Contract Support

All approaches to contract support require programmers to explicitly specify the semantic interface for each class. This interface, informally called the contract, consists of assertions that denote the invariant property for each class, the preconditions for each method, and the postconditions for each method. A number of programming languages allow programmers to specify the semantic interface using keywords provided by the programming language. These include Euclid [13], Alphard [21], Anna [14], SR 0, D [5], and recently Python [6] and Java [12] have been extended. Unfortuately, most of the languages that do support contracts are not widely-used. While built-in support is the easiest and most elegant approach from a programmer's perspective, to so drastically change an existing programming language requires time, effort, and resources for rewriting the compiler. The fact that so few languages include it is evidence of significant effort needed. Since contract support is not part of the most commonly-used languages, four categories of approaches have been taken to implementing contracts in languages that do not support them intrinsically: 1) Code Libraries, 2) Preprocessor, 3) Behind-the-Scenes, and 4) Programming Language-Independent.

The Code Library. One approach is to create libraries of functions needed to support contracts and prescribe conventions for calling these functions. In general, it is the programmer's responsibility to call these functions in the appropriate places and to provide at least some of the exception handling. One such approach was created by

McFarlane for the C# language [15]. Another more sophisticated library approach is that used by the creators of jContractor for Java [9][25]. Programmers follow a naming convention to write contracts as methods in each class definition. jContractor identifies these patterns and uses the Java Reflection API to synthesize the original code with contract-checking code. jContractor can add the contract-checking code to the bytecode or as the classes are loaded. Some scholars categorize jContractor as a Behind-the-Scenes (metaprogramming/reflective) approach because of its use of the Java Reflection API [19]. Library approaches have the advantage of allowing programmers to use standard language syntax—and they work with any implementation of the language. No special specification language is needed for the assertions, and the naming conventions or function names are intuitive. No changes to the development, test, or deployment environment are needed, and no special tools such as modified compilers, runtime systems, or preprocessors are required. Library techniques, however, tend to be burdensome for programmers to use. Programmers must either actually write methods for each assertion while follow naming conventions or they must remember to call functions in certain places. In addition, for McFarlane's approach, inherited assertions must be handled by the programmer. Another problem may be the lack of integration with pre-existing or subsequently changed libraries. Furthermore, this approach does not take into account the situation in which a single program contains multiple instances of an object, which must be distinguished at runtime for trace purposes.

Preprocessor Approaches. A common approach to implementing contracts is the use of a preprocessor to transform code containing formal comments, macros, or keywords into compilable code with contracts. The iContract tool for the Java programming language uses just such an approach [11]. Programmers add contracts as JavaDoc comments written in a special specification language. The iContract tool converts these comments into assertion-checking code. Another example of a preprocessor approach is Jass (**J**ava with **Ass**ertions) [24]. Programmers specify contracts using the extended language Jass and use a preprocessor to translate the extensions to ordinary Java. Preprocessor approaches have a number of advantages. First and foremost, the compiler need not be rewritten. Second, the resulting code is compatible with any implementation of its particular language. Third, because contracts and code are written simultaneously, contracts are part of the design and implementation phases. In addition, preprocessor approaches often allow multiple options for regulating the degree of the translation, and they need not be run at all. Furthermore, the developer has complete control of the code and can make changes to the processed code if they wish to be involved with the details of contract implementation. Several disadvantages to preprocessor approaches are also commonly cited. First, because the original source code is changed, the line numbers of compiler errors, debugging output, and exceptions do not correspond with line numbers of the original program [19]. The contract-instrumented code is also more difficult to read because of the extra code added to enforce contracts. In addition, programmers must learn the particular specification language for expressing the assertions. As others have pointed out, the programmer is unable to add new ways of handling violations to the assertions at runtime and unable to change the level of contract checking at runtime [2][22]. Some consider the use of an external preprocessor to be burdensome. In addition, the



preprocessor approach is also not appropriate for real time systems, where the runtime for a debug build differs from that of a release build.

Code translation, a variation of the preprocessor approach, involves translating code with contracts written in a language that supports DBC into the code of a language that does not support it. This method is used by Plösch for implementing DBC in C++ based on Python [18]. This approach has some distinct disadvantages. Using this approach requires that all programmers learn both languages and development environments, which is inefficient and more expensive. Also, numerous technical problems arise during the transformation process [2].

Behind-the-Scenes Approaches. Another category of approaches makes use of DLLs to incorporate the contract-checking code into the source code at load time, in a behind-the-scenes manner, so that the programmer never sees any contract-checking code. Two tools make use of this approach for implementing contracts in the Java programming language: JMSAssert and HandShake. JMSAssert [7] uses a preprocessor similar to iContract, in which programmers write the contracts using special tags in JavaDoc comments. However, it also incorporates a dynamically linked library or "assertion runtime." The preprocessor is used to process the source code, resulting in contract files written in JMScript (a proprietary scripting language). A special extension DLL, containing the JMScript interpreter, enforces the contracts by executing the "triggers" contained within these JMScript files. A benefit to this approach is that only the contract tags and assertions are visible because contracts are enforced "behind-the-scenes." JMSAssert does have its drawbacks, however. First, programmers must learn and be disciplined enough to use the tags for expressing the Boolean assertions. Second, when implementing interfaces or inherited classes, programmers must ensure that inheritied preconditions are only weakened and post conditions are strengthened. Third, JMScript is a proprietary language, and JMSAssert is apparently compatible only with JDK 1.2. Compatibility may be a concern with future versions. JMSAssert also requires some changes to the development environment. The HandShake tool requires that the programmer create a separate "contract file" associated with each class and interface [8]. A preprocessor, the "HandShake compiler," converts the contract file to a binary file. At load time, a special DLL called the HandShake library merges the contract binary and the original file to create a class file with contracts. The HandShake tool has the advantage that contracts can be added without modifying their source code; however, requiring that contracts for each class be written in a separate contract file does not encourage the design and development of code in unison with contracts. In addition, the task of creating a separate file, written in a special syntax, is time-consuming and tedious for programmers. Furthermore, the HandShake Library may not be supported on all platforms, since it is a non-java system [9].

Programming Language-Independent Approaches. Recently with the release of Microsoft's .NET development environment, the idea of using a programming language-independent method for implementing contracts has become more prominent. Some of these proposals sound promising [22][25]; however, the only tool currently available is the Contract Wizard [2], which reads a .NET Assembly as its input. It uses the Eiffel compiler to access the appropriate information from the .NET metadata and provides fields for the user to input assertions. It then generates Eiffel classes containing the

specified contracts, and using the Eiffel compiler, generates another .NET assembly—this time containing contracts. On the plus side, the source code contains no extra code for implementing the contracts, the programmer does not need to use special tags to specify the contracts, and the GUI for adding contracts is easy to use. More importantly, it can be used for any language supported by .NET. The Contract Wizard does, however, have several disadvantages. It requires some type of Eiffel development environment—in addition to an environment for whatever language is being used to write the source code. The main disadvantage to the Contract Wizard is that contracts cannot be specified during the design or even the implementation phase. Adding contracts is done in a separate phase after implementation is complete [22]. It is not clear what happens to the contracts when the programmer goes back to modify the original language source code [2]. It appears that the contracts would have to be recreated. As a result, contracts would be added as late as possible in the development cycle. Despite its disadvantages, this "late-binding" type of approach makes changes easier.

3 THE CONTRACT SHARP SYSTEM

The Contract Sharp Approach

Among the aforementioned tools, only two could actually be used today for implementing contracts in the C# language: the McFarlane library approach and the Contract Wizard. To provide support for contracts in C#, we have developed Contract Sharp, through which we attempt to overcome the limitations of the existing tools and approaches. The approach taken by Contract Sharp is an extension of Zhang and Zheng's approach for the enhancement of Java [27]. Like their tool, Contract Sharp is a syntax-directed, code development tool with a GUI that assists C# programmers in specifying the appropriate assertions for each class and method.

Figure 1 shows the main window of the GUI for a simple Counter program that increments and decrements by 1.

The Contract Sharp approach could be characterized as a mixed one. Like a preprocessor approach, Contract Sharp transforms contracts and source code into contract-instrumented code, which is then compiled with the existing C# compiler. However, because it is implemented via a GUI, it does not place upon the programmer the full burden of writing the contracts. The programmer simply types Boolean expressions in GUI-provided text fields when defining a class or method, as shown in Figure 2. After defining the contracts, they are visible in the main window. When the file is saved, the code and contracts are represented in XML. This is a significant improvement over the McFarlane library approach where the programmer has to call the appropriate library routines at the beginning and end of each method. Also, it aids the programmer in being disciplined by requiring the contracts at the time the classes and methods are created, overcoming one of the major weaknesses of the Contract Wizard—the fact that contracts cannot be specified during the design or the implementation phases. To facilitate debugging, it generates a program trace for monitoring the runtime

verification of the assertions. In addition, it automatically generates documentation for all class and method interfaces and contracts, including programmer comments.

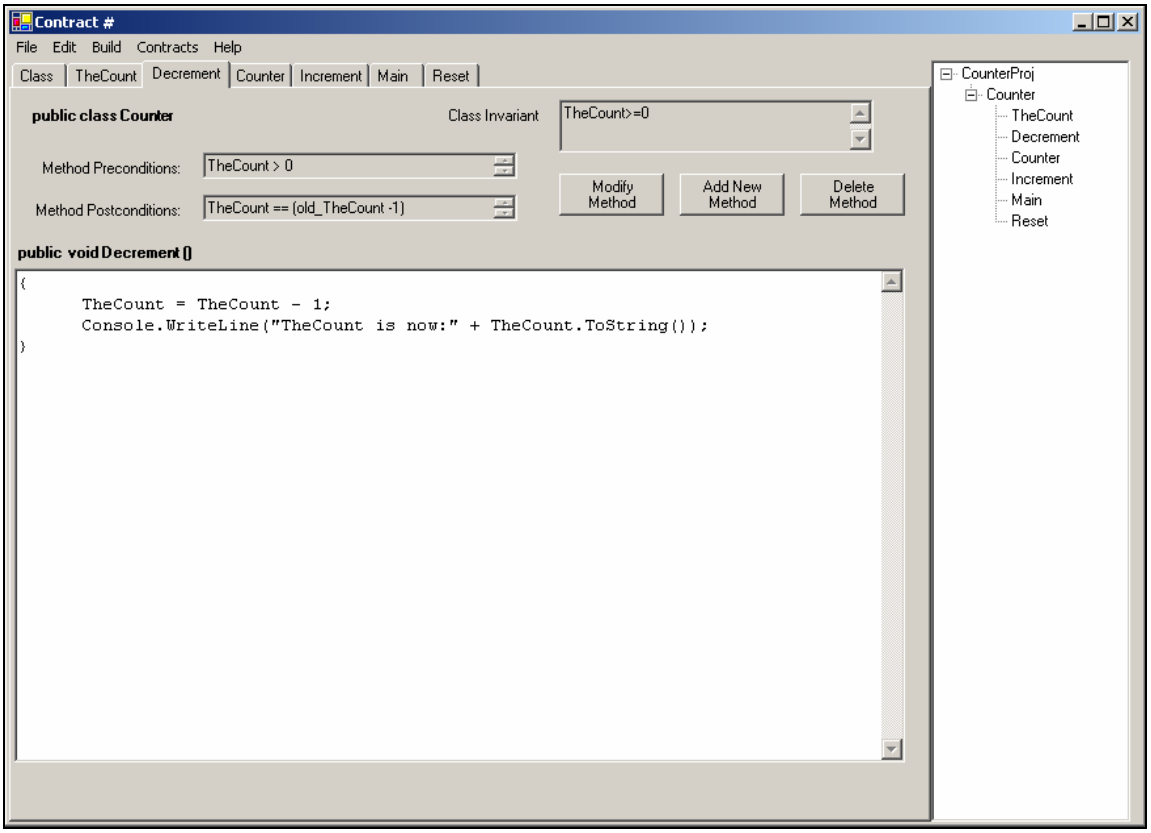


Figure 1: Main Window with Method Tab Selected

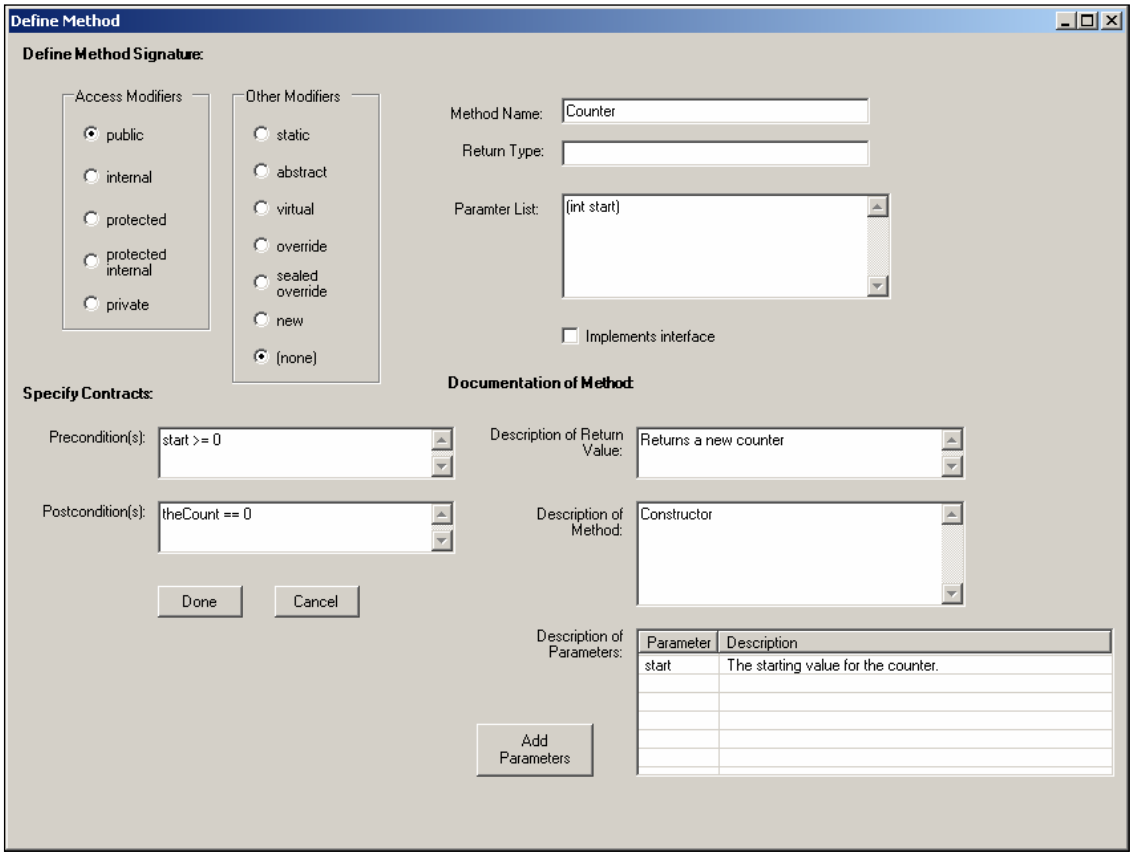


Figure 2: Define Method Window

Contract Sharp attempts to overcome some of the difficulties inherent in preprocessor approaches in the following ways:

[1] Non-corresponding line numbers. Used alone with the C# command-line compiler, Contract Sharp has this problem; however, because of its close integration with Visual Studio .NET (VS), it seems inconceivable that someone would use Contract Sharp for debugging, foregoing the sophisticated debugging tools of the Visual Studio environment. The most likely scenario is for the programmer to have both Contract Sharp and VS open at the same time, in a panel fashion. All line numbers from debugging tools would be from the contract-instrumented code. The programmer would find the bugs in the enhanced code in VS and make the changes in the Contract Sharp window. This method would be slightly inconvenient, but most programmers want the benefit of sophisticated debugging tools.

[2] Cumbersome contract-instrumented code. Because Contract Sharp implements contracts as objects, the amount of code added to each method is minimal. The contract object is declared once, and methods are called to produce the runtime trace. No "if" statements and "print" statements are needed in the source code, because they are encapsulated in the contract object. Also, "cluttering" of the source code is

avoided because Contract Sharp surrounds any contract-related code with special symbols (#####) that set it apart and enhance readability.

[3] Learning curve for specification language. Contract Sharp GUI fields capture contracts, so no specification language or special tags are necessary. This is an advantage even over built-in approaches, which require keywords.

[4] Inability to switch contract checking dynamically. Contract Sharp allows the user to change the level of contract checking at runtime on a class-by-class basis, without recompiling. In addition, the programmer has the option of handling violations to the assertions at runtime. Figure 3 shows the window in which checking levels are specified.

[5] Burdensome use of an external preprocessor. While some might consider a preprocessor to be burdensome, running the Contract Sharp preprocessor is a straightforward task, and Contract Sharp actually adds some benefits for the programmer by automatically generating part of the code for each class as well as the embedded XML tags for documentation.

Next, we discuss how Contract Sharp overcomes some of the weaknesses inherent in Zheng's Enhancement to Java. 1) Through the use of XML, Contract Sharp allows the developer the option of going back to modify the source code without requiring contracts to be recreated. That is, a programmer will be able to open a previously-created source code file containing contracts, and Contract Sharp (unlike the Contract Wizard) will make possible the modification of classes or contracts. 2) Contract Sharp improves upon Zheng's inheritance mechanism. For a child class that inherits from a parent class, Zheng's tool separately checks both the parent and the child preconditions and postconditions. It does not combine the contracts of the parent and child in a manner in which the preconditions of the subclass only weaken the parent preconditions and the postconditions only strengthen those of the parent class. Contract Sharp combines the contracts of a parent class with its child, allowing preconditions of the overriding methods of the subclass only to weaken the parent preconditions and the postconditions of the overriding functions of the child only to strengthen those of the parent. In addition, Contract Sharp creates combined class invariants in a manner that only strengthens the class invariant of the parent class. 3) As previously mentioned, Contract Sharp provides the option of changing the level of contract checking at runtime on a class-by-class basis, so that changing levels does not require recompilation. 4) Contract Sharp also improves documentation content and readability. Contract Sharp's automatic documentation includes additional programmer comments, including descriptions of parameters and Method return values. Because Contract Sharp transforms the XML documentation file created by the C# compiler into HTML using a stylesheet, the output can be rendered as HTML. In addition, the programmer can modify the stylesheet to change both format and content of the final HTML file.

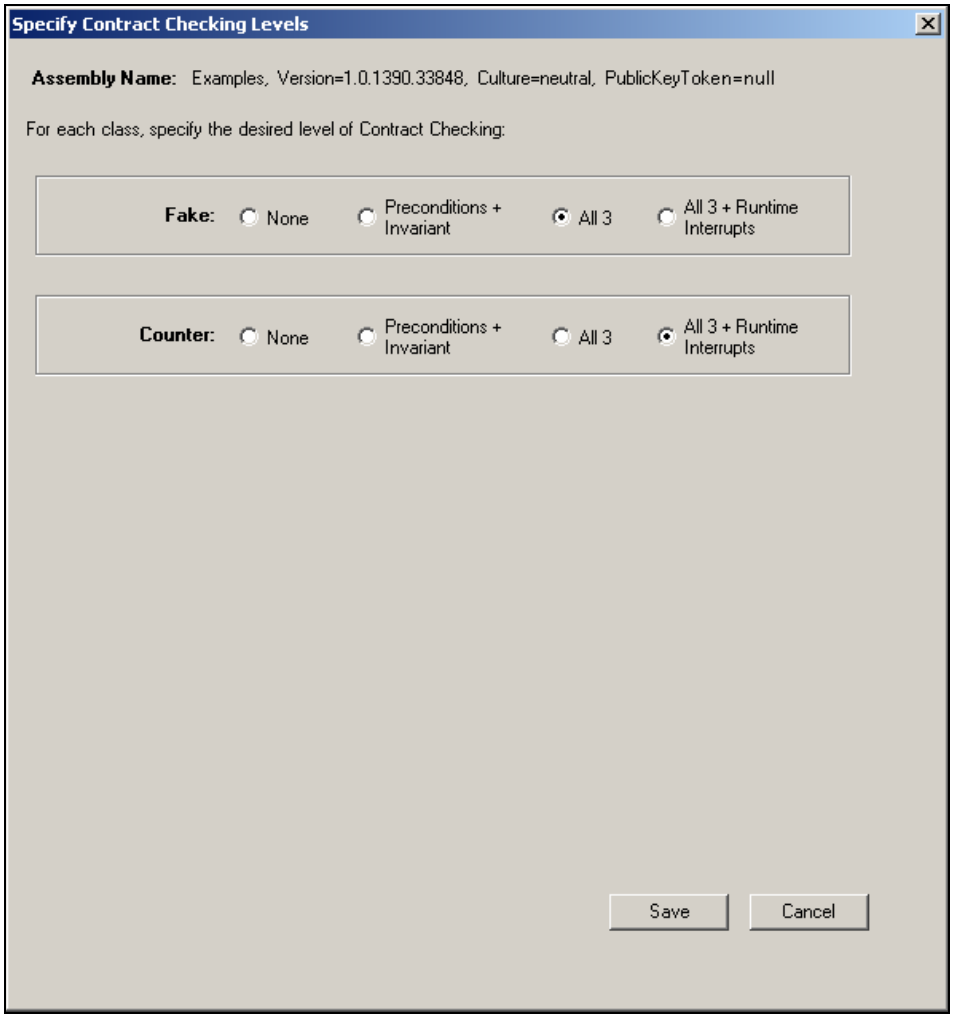


Figure 3: Specify Contract Checking Levels Window

The Functionality and System Architecture

Contract Sharp supports contracts through a set of event-driven functions. The dataflow diagram in Figure 4 provides an overview of these functions.

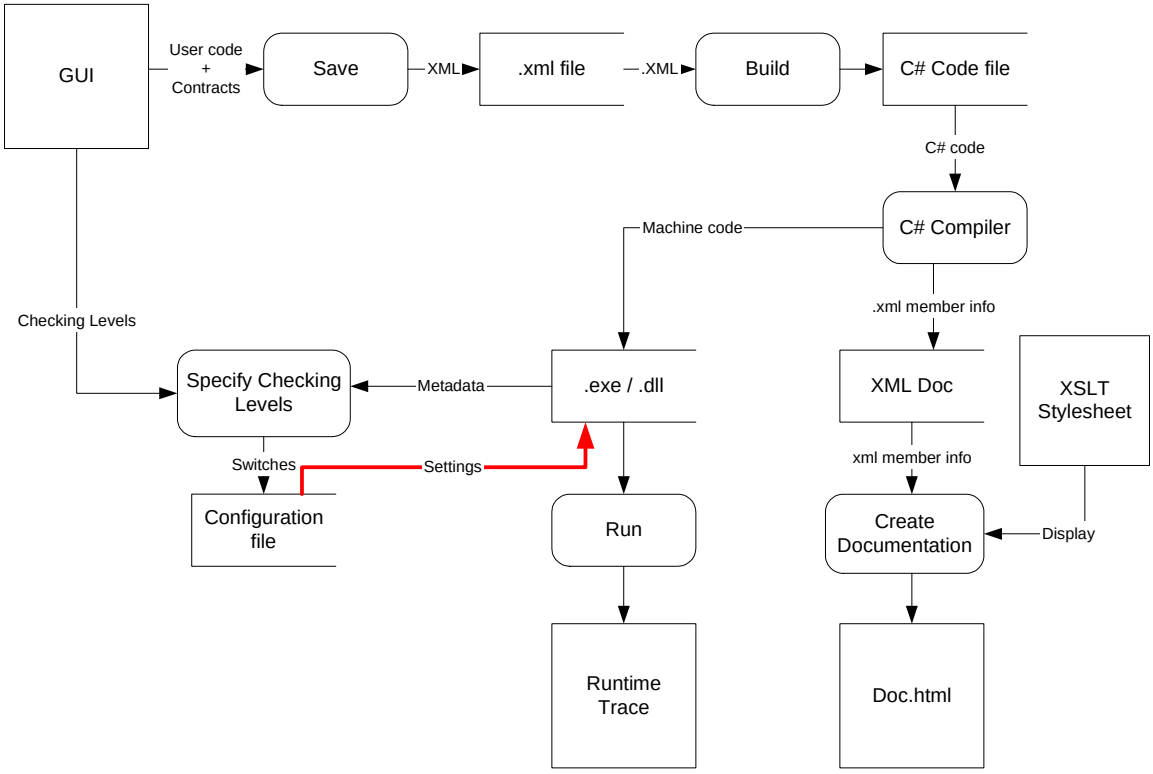


Figure 4: Contract Sharp Dataflow Diagram

The user creates C# code and specifies contracts by using Contract Sharp's GUI, shown in Figures 1 and 2. When coding is complete, the code, contracts, and mandatory comments for documentation purposes are saved as XML. When the user triggers a "build" event by selecting the appropriate command, Contract Sharp transforms the code and contracts into standard C# code, and it transforms the comments into a compiler-recognizable form of embedded XML in the source code. The C# source code must then be compiled into executable or library files by the C# compiler. In addition, during compilation, the C# compiler creates an XML documentation file containing the embedded documentation that describes each class member. When the user triggers a Create Documentation event, Contract Sharp transforms the XML documentation file into HTML documentation of all of the classes and class members. This process performs the XSLT transform specified by a Contract Sharp XSLT Stylesheet. When the executable file is run (or in the case of a library file, when a program instantiates one of the classes in that file), a runtime trace is automatically produced. This text file traces program

execution and prints out the results of the evaluation of the contracts at each checkpoint: object instantiation, the beginning of each method call, and the end of each method call.

After compilation of an assembly, the user may specify on a class-by-class basis the degree to which contracts should be checked. The Specify Checking Levels function uses .NET reflection to read metadata embedded by Contract Sharp in the source code. The function determines which contract-supporting classes are present in the assembly, and for each such class, allows the user to specify the level of contract checking. Contract Sharp sets the checking levels by modifying the assembly's corresponding configuration file with the appropriate settings. The runtime trace output is controlled by these settings.

In addition to the dataflow, a high-level view of Contract Sharp's software architecture is also included here to show the system design. Figure 5 depicts the software components of the Contract Sharp system, with an arrow indicating that a component calls another. The software components are as follows:

- **Central System Control.** Calls various components based on user input; captures output and passes it to appropriate component.
- **GUI.** Captures user input and passes it to Central System Control; presents output to user.
- **Contract Sharp Preprocessor.** Reads in the XML code, contracts, and documentation, along with the Build information and generates contract-instrumented C# code with XML-embedded documentation.
- **XML Manager.** Generates XML, modifies XML, and performs version control.
- **Application Configuration**. Modifies the runtime configuration file of the current assembly to set the appropriate contract-checking levels.
- **Documentation Generator**. Reads in documentation from doc file produced by C# compiler and creates HTML documentation.
- **C# Compiler.** Microsoft's existing C# compiler is used in the system.

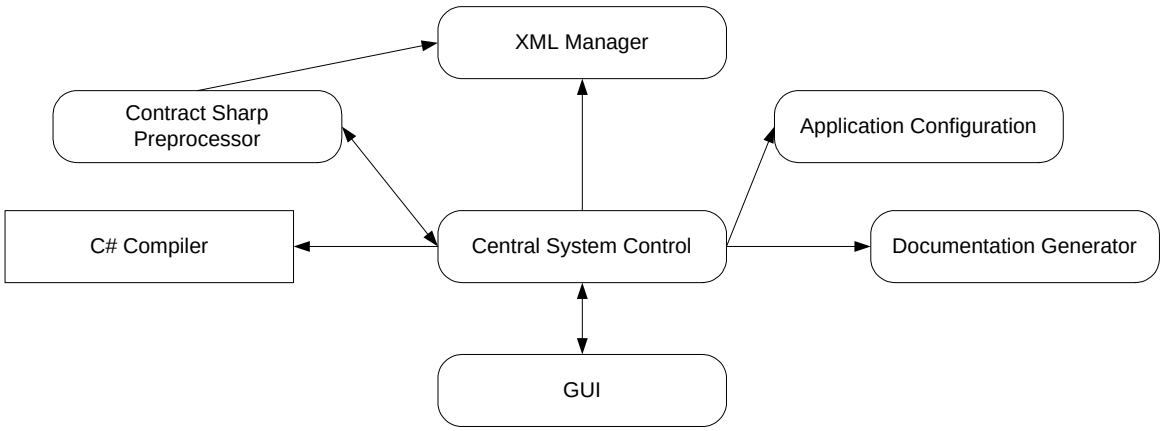
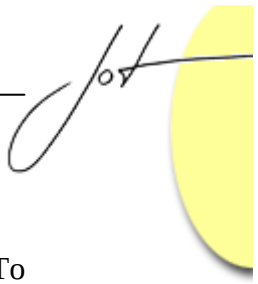


Figure 5: High-Level System Software Architecture



4 DISCUSSION AND CONCLUSION

Currently, there is little basis for comparison of tools that support contracts in C#. To provide an objective basis for comparing Contract Sharp to similar systems, a set of criteria, developed by Plösch [19], is applied to Contract Sharp in the tables that follow. Although Plösch evaluates only Java systems, the criteria themselves are not language-specific. Table 2 to Table 5, inclusive, show the application of Plosch's four categories of criteria.

Plosch's Criterion	**Contract Sharp**
BAS-1 (Basic assertions): Does the system support basic assertion annotations in the implementation of a method?	Yes.
BAS-2 (Preconditions and Postconditions): Does the system support preconditions? Does the system support postconditions?	Yes.
May assertion expressions access properties of a class?	Yes, public properties.
Are properties of a class guaranteed to remain unchanged during assertion checking?	Yes.
May assertion expressions also contain method calls?	Yes.
Is it guaranteed, that a method call does not produce any side effects (especially changes of the state of the object)?	No.
BAS-3 (Invariants): Is it possible to formulate invariants?	Yes.
Are there any restrictions in formulating invariants (compared to the formulation of preconditions or postconditions)?	No.

Table 2: Basic assertion support (BAS) Criteria

Plosch's Criterion	**Contract Sharp**
AAS-1 (Enhanced assertion expressions): May assertions contain Boolean implications?	No. This is a limitation inherent in C# language.
May postconditions access the original values of parameters, i.e., the values at method entry?	Yes.
May postconditions access the original values of instance variables, i.e., the values at method entry?	Yes.
May an arbitrary expression be evaluated at method entry?	Yes.
AAS-2 (Operations on collections): Does the system support assertion expressions on collections?	No. This is a limitation inherent in C# language.
Is it guaranteed that collections remain immutable in assertion expressions?	N/A
May universal quantifications be expressed in the expression language?	No. This is a limitation inherent in C# language.
May existential quantifications be expressed in the expression language?	No. This is a limitation inherent in C# language.
AAS-3 (Additional expressions): Does the assertion expression language have additional features?	No. This is a limitation inherent in C# language.
Are these additional features guaranteed to be side effect free?	N/A

Table 3: Advanced assertion support (AAS) Criteria

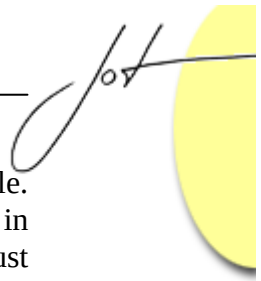
Plosch's Criterion	Contract Sharp
SBS-1 (Interfaces): Is it possible to specify contracts for interfaces?	Yes.
May contracts be added for classes implementing assertion-enriched interfaces?	Yes.
SBS-2 (Correctness I): Does the system impose any restrictions on subcontracts?	Yes.
Does the system ensure that preconditions may only be weakened?	Yes.
Does the system ensure that postconditions may only be strengthened?	Yes.
SBS-3 (Correctness II): Does the system impose stronger requirements on subcontracts as specified in SBS-2?	Yes.
Does the system ensure, that the correctness rules for behavioral subtyping are not violated?	Yes.

Table 4: Support for Behavioral Subtyping (SBS) Criteria

Plosch's Criterion	Contract Sharp
RMA-1 (Contract violations): Is an exception handling mechanism available in case of violations of assertions?	Yes
Are there additional features available for dealing with assertion violations (e.g., log files)?	Yes.
RMA-2 (Configurability): Is it possible to enable and disable precondition checking, postcondition checking and invariant checking selectively?	Yes, but no postcondition checking without precondition and invariant checking.
Is it possible to enable and disable assertion checking on a package, class or even method level?	Yes, on a class level.
RMA-3 (Efficiency): Are there any additional memory requirements even when assertion checking is disabled?	No.
Is there any additional processor usage even when assertion checking is disabled?	No.

Table 5: Runtime monitoring of assertions (RMA) Criteria

Contract Sharp provides a number of useful features that are not available in other systems, such as the option of changing the level of contract checking at runtime on a class-by-class basis and the ability to inherit contracts. We see certain improvements, however, that would be beneficial as part of our future work. The interface for Specify Checking Levels has a design that is not suitable for applications with a large number of classes. In addition, an XML Schema Definition (XSD) is under way to ensure any XML file opened by Contract Sharp is in a schema-compliant format. We also intend to



continue work on inheritance. Base classes should not be limited to only this project file. The GUI could be improved to handle multiple project files, but they would have to be in XML. A drawback of the system is that the XML representation of all parent classes must be available. Parent classes could be compiled separately and reflection could be used to access their contracts. Further, Contract Sharp does not process nested classes, although the system allows the programmer to create nested classes. In addition to the above improvements, we are in favor of giving more control to the programmer for dealing with a contract violation. The current exception handling mechanism is rigid in that programmers cannot write their own exception handlers. This is because of the default exception processing associated with C#. The choices are: to continue execution, step into the debugger, or stop execution.

An additional drawback is one inherent in the approach itself: it does require the user to learn a new development environment. A number of the other approaches allow programmers to use their existing development tools. Contract Sharp is currently an experimental, not commercial, system. It demonstrates the feasibility of a GUI, mixed approach, and it would ideally be integrated into a commercial GUI system so that developers would need to learn only a single development environment.

Future Directions

We have begun to evolve this C#-specific tool into a language-independent tool for use with multiple languages. Contracts are specified in unison with the development of the code (as in our approach), and contract-instrumented code is generated in a number of different languages. By analyzing the work presented to identify both the language-independent components and the language-dependent components of the system, we have initiated an effort to develop a language-independent software framework for the enhancement of object-oriented programming languages to support contracts. It is our hope that, with such a framework available in the future, adding the support for Design by Contract™ to any object oriented programming language would be a highly efficient and straightforward step because all the language-independent parts are provided by the framework and only the language-specific parts would need to be addressed.

REFERENCES

[1] Andrews, G. R. and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. Redwood City, CA: Benjamin/Cummings, 1993.

[2] Arnout, Karine and Raphaël Simon, "The .NET Contract Wizard: Adding Design by Contract to Languages Other Than Eiffel," Goleta, CA: Interactive Software Engineering, Inc, 2001. http://se.inf.ethz.ch/publications/arnout/conferences/tools_usa/2001/contract_wizard.pdf

[3] Berg, W, M.Cline, and M.Girou, "Lessons learned from the OS/400 OO project", CACM, vol. 38, no. 10, 1995.

[4] Binder, R., "Testing Object-Oriented Systems," Addison Wesley Longman, 2000, pp. 807-916.

[5] "Contracts." Digital Mars: The D Programming Language. March 28, 2003, http://www.digitalmars.com/d/dbc.html.

[6] "Contracts for Python," June 21, 2003, http://www.wayforward.net/pycontract/

[7] "Design by Contract™ for Java™ Using JMSAssert™ ," http://www.mmsindia.com/DBCForJava.html

[8] Duncan, Andrew and Urs Holzle. "Adding Contracts to Java with Handshake." Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, http://citeseer.nj.nec.com/duncan98adding.html

[9] Karaorman, Murat, Urs Hölzle, and John Bruno, "jContractor: A Reflective Java Library to Support Design by Contract," In *Proceedings of Meta-Level Architectures and Reflection*, Lecture Notes in Computer Science, nr 1616, Springer Verlag, 1999. http://people.cs.uchicago.edu/~robby/contract-reading-list/ jContractor.pdf

[10] Kott and Peasant, "Representation and Management Of Requirements—The Rapid_WS Project," Concurrent Engineering: Research and Applications, vol. 3, no. 2, June 1995.

[11] Kramer, Reto. "iContract – The Java ™ Design by Contract ™ Tool," *Proceedings of Technology of Object-Oriented Languages*, TOOLS 26, IEEE Computer Society, 1998. http://www.reliable-systems.com/tools

[12] Lackner, Martin, Andreas Krall, and Franz Puntigam, "Supporting Design by Contract in Java," Journal of Object Technology, vol. 1, no. 3, special issue: TOOLS USA 2002 proceedings, pp. 57-76, August 2002, http://www.jot.fm/issues/ issue_2002_08/article4

[13] Lampson, B. W. J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. "Report on the Programming Language Euclid," *SIGPLAN Notice*, vol. 12, number 2, pp. 1-79, 1977.

[14] Luckham, D. and F. W. von Henke. “An Overview of Anna, A Specification Language for Ada.” *IEEE Software*, vol. 2, number 2, pp. 9-22, 1985.

[15] McFarlane, Kevin. “Design by Contract Framework,” The Code Project, July 13, 2002, http://www.codeproject.com/csharp/designbycontract.asp.

[16] Meyer, Bertrand. *Eiffel: The Language*. Prentice Hall, 1992.

[17] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

[18] Plösch, Reinhold, “Design by Contract for Python,” *Proceedings of Asia Pacific Software Engineering Conference*, IEEE Computer Society, 1997.

[19] Plösch, Reinhold, “Evaluation of Assertion Support for the Java Programming Language,” *Journal of Object Technology*, vol. 1, no. 3, special issue: TOOLS USA 2002 proceedings, pp. 5-17, August, 2002, http://www.jot.fm/issues/ issue_2002_08 /article1

[20] Rumbaugh, Jacobson, and Booch, “The Unified Modeling Language reference manual,” Addison Wesley Longman, 1998.

[21] Shaw, M. *Alphard: Form and Content*. New York: Springer-Verlag, 1981.

[22] Sjögren, Andreas. “A Method for Support for Design by Contract on the .NET platform.” Extended Report for I. Crnkovic and M. Larsson (editors), “Building Reliable Component-Based Systems.” Artech House, July 2002.

[23] Stavely, A., “Toward Zero-Defect Programming,” Addison Wesley, 1999.

[24] “The Jass Page: Introduction to Design by Contract.” University of Oldenburg, Germany http://csd.informatik.uni-oldenburg.de/~jass/doc/intro.html

[25] Tran, Nam, Christin Mingins, David Abramson, and Alex Mikunov, “Rotor Project Proposal: Support for Assertions in the Rotor Runtime,” School of Computer Science and Software Engineering, Monash University, Australia, June, 2002. http://www.csse.monash.edu.au/~namtt/rotor/MonashRotorProject.html.

[26] “What is jContractor?” http://jcontractor.sourceforge.net/

[27] Zhang, Cui and Lucy Y. Zheng, “Object-Oriented Programming with Assertions,” International Journal of Computer Science and Information Management, vol.3, no.1, pp. 59-71, 2000.

About the authors

Rachel Henne-Wu earned her Master's Degree in Computer Science from the California State University, Sacramento, in May of 2004. She is currently employed by Old Republic Title Information Concepts, in Roseville,California, as a software developer. She can be contacted at rachel_henne@comcast.net .

Dr. William Mitchell is a professor of Computer Science at California State University, Sacramento. His areas of interest include database systems, simulation, and OO languages and systems. Recent work has involved simulation environments for distributed computing and E-commerce workloads.

Dr. Cui Zhang is a full professor in the Department of Computer Science, California State University Sacramento (CSUS). Her research and teaching interests include software engineering, object-oriented techniques and methodologies, formal methods for software engineering and for computer security, and programming languages. She received her Ph.D. in Computer Science from Nanjing University, China, in 1986. She has published more than fifty (50) research papers in international conferences and journals including publishing as primary author in Theoretical Computer Science and Journal of Software Testing Verification and Reliability. She has also served on the program committees for multiple international conferences.