Static and Dynamic Contract Verifiers For Java

Hongming Liu, Lizhang Qin, Jing Wang, Naveen Vemuri, Xiaoping Jia
School of Computer Science, Telecommunication and Information Systems
DePaul University
Chicago, Illinois, USA

ABSTRACT

Design By Contract (DBC) is a systematic approach to specifying and implementing object-oriented software systems. DBC has been proved to greatly benefit software development. However, Java does not natively support DBC. We have developed a comprehensive solution to bring DBC into Java. The static and dynamic contract verifier is the most crucial part of the solution. We have developed a toolset support DBC using these two verifiers. This paper presents details of their design and implimentation. The tool used for dynamic contract verifier is ContractChecker, which generates test code into original Java code, thus enables runtime validation. Static contract verification is done by Static Contract Verifier, which uses an automated theorem prover to verify contract.

KEYWORDS

Design By Contract, Java, Dynamic Contract Verifier, Static Contract Verifier

1 Introduction

Design by Contract(DBC) was first introduced by Bertrand Meyer as part of the Eiffel Project. Under the DBC theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations: contracts [1]. DBC provides a formal way to incorporate specification information, which is obtained from comments, into the code. By doing this, the codes implicit contracts are transformed into explicit requirements that must be satisfied.

DBC can be viewed as a systematic approach to specifying and implementing object-oriented software systems. Meyer enumerates the primary benefits of this approach [2]:

- A better understanding of object-oriented software construction in general
- A systematic approach to building correct software systems
- An effective framework for quality assurance
- A method for documenting software
- Better control of the inheritance mechanism
- A technique for dealing with abnormal cases, leading to a proper use of exception handling mechanisms

The three key elements of DBC are method preconditions, method postconditions and class invariant. Method preconditions are conditions that the client must meet before a method is invoked. Method postcondtions are conditions that a method must meet after its execution. Class invariant are consistency conditions of objects, which must hold for all instances.

Java itself does not support DBC, though the assert statement introduced in JDK 1.4 does represent a step in this direction. Assert provides a mean for programmers to provide runtime checking for their programs. Assert, however, does not provide contract information to indicate the functionality of a method or class such as method preconditions, postconditions and class invariant, which are emphasized by DBC.

In this paper we present techniques for both static and dynamic contract verification, which is the key part of a comprehensive solution for bringing DBC into Java. Our static contract verification technique is theorem proving based. The theorem prover helps in evaluating the implications of conditions defined in the program. The dynamic contract verification is implemented by generating test code into original Java code, therefore enables runtime validation.

The rest of the paper is organized as follows: Section two discusses the technique we used for static contract verification. Section three discusses the details of dynamic contract verification technique. We compare our techniques with other DBC tools in section four. Finally we conclude and discuss our future work in section five.

2 Overview of DBC Toolset Components

To demonstrate our solution for bringing DBC into java, we develop a DBC toolset, which is designed to incorporate DBC capability into the Java language, not only providing the conventions for DBC specification, but also consisting of three major components: ContractChecker, Static Contract Verifier and DocGen, which are intended to make use of the DBC infor-

mation in the programs. Figure 1 shows the overview of these components.

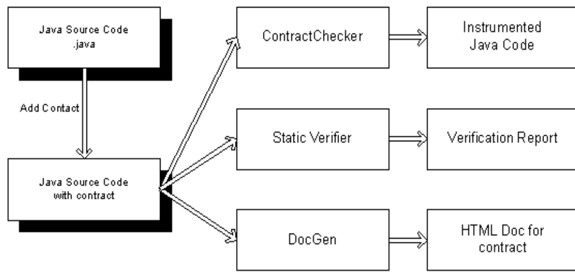

Figure 1. Components of Our DBC toolset

ContractChecker does runtime checking for the contracts by generating new pieces of code and inserting them into the original code, so when executing the new code, any violations against the contract information programmer provide will be reported and avoid further severe exceptions.

Static Contract Verifier statically analyzes the program for potential null pointer and array out of bounds exceptions and also incorporates contract verification for each Java class and method. Contract verification indicates whether a specific method is able to satisfy the postconditions if the methods preconditions are assumed to be true. If the postconditions are not guaranteed given satisfied preconditions the contract is invalid.

DocGen is a Java doclet developed on the basis of the standard doclet. It is able to extract and display any contract, including any inherited contract that is documented in a formal comment unit, as long as the documentation follows a certain format. DocGen has several major features that make it different from other similar tools: 1) It is based on the newest version of Java doclet, 2) It automatically processes all inherited contracts; 3) It allows some degree of flexibility in terms of where the contract should be documented.

In this paper, we will focus on discussing ContractChecker and Static Contract Verifier which provide the dynamic and static contract verification.

3 Notations for Specifying Contracts

The essential issue to incorporate DBC support to Java is to introduce the notations to specify contract information. In our convention, any contract has to be commented according to a certain specification. Briefly, besides the Java documentation convention, the following rules must be followed:

- The contract should only be placed in a formal Java comment unit, which is wrapped by /** and */. Any information in other comment form will be ignored.
- Certain custom tags are used to express these contracts.
- Different contracts are placed at specific levels of the source code. Contracts that are placed in the wrong place will be ignored.

The advantage of denoting DBC information in comment form instead of within normal code (such as by creating a DBC API, for example) is that DBC enabled Java code with comment-based contract information is backwards-compatible with any Java compiler. All contract information starts with a custom tag which indicates the type of this contract, such as method precondition, method postcondition, or class invariant. The custom tag is followed by a boolean expression which specifies the condition to be held.

In our project, we are using an extended form of Java expression syntax to describe contract expressions. The only valid expression following with a custom tag is a boolean expression. To be able to express more complicated conditions, several new syntaxes are introduced to specify quantified expressions, the return value of a method, and the post-stage value of an object along with more operators.

Operators In Expression

All normal Java operators are valid for describing contract conditions. Two additional operators are also allowed to add more capability for logical expression.

- => operator is used to specify logical implication between two predicates, such as P=>Q
- <=> operator is used to specify logical equivalence between two predicates, such as P<=>Q

Quantified Expressions

Two kinds of quantified expressions are introduced: forall and exists. With the use of forall and exists the programmer can express properties that must be valid for all elements of and finite set of assertions for one element of this set. Syntax of forall and exists is given in Appendix.

Return Value of a Method

At times it is necessary to refer to the return value of a method as a type of method postcondition. We have a new syntax to express the return value given a method with name m!, where m! refers to the return value of method m.

Post-stage Value of an Object

A mutable object may change its value within a method. It is also desirable to be able to express the post-stage value of an object. We use the following syntax: obj where obj is the name of the object. For example, o is the post-stage value of object o. Note that the post-stage value can only be applied to an

object. The correct way to specify the post-stage value of a field of a object is: obj.field, where obj is the name of the object and field is the name of the field.

Figure 2 shows an example of a Java class with design by contract information. The source code includes precondition, postcondition and invariant contracts.

```
/**
 * @invariant x > 10
 */
public class A {
  int x;
  /**
   * @pre val > 20
   * @post this'.x == this.x + val
   */
  public void setX(int val){x = x + val;}
  /**
   * @post getX! == x
   */
  public int getX(){return x;}
}
```

Figure 2. Example of a Java class using DBC

More custom tags and syntax details are provide in Appendix.

4 Dynamic Contract Verifier

ContractChecker is a tool that checks the DBC contracts at runtime. It consists of two modules: Contract Extraction Module and Code Generation Module. By running ContractChecker on Java source code, modified code will be generated with the addition of the test code. The generated code is compiled and executed in a same way as the original code. If a contract is violated, exceptions will be raised and recorded in a trace file.

Contract Extraction Module handles all of the extended custom tags. It extracts and parses the contract elements while Code Generation Module generates the test code. The generated code used to verify contracts are inserted at different levels of the original Java source code. The new code will be compiled and executed, and adds the ability to detect any contract violations.

Custom Exceptions

In order to indicate a particular violation, four runtime exceptions have been defined.

- PreconditionException are thrown when preconditions of a method are violated.
- PostconditionException are thrown when postconditions of a method are violated.
- InvariantException are thrown when invariant of a class are violated.
- ContractCannotBeCheckedException are thrown when elements in a contract can not be tested. For example, if an object is not cloneable, it cannot be preserved. Thus the postcondition that contains a post-stage value of that object cannot be tested.

No specific exceptions are defined for violation of fact, assert or loopInvariant, since they are instrumented as assert statement.

Check Points

It is important to know when the contract elements are checked. The contracts indicated by @fact, @assert and @loopInvariant are checked immediately before the corresponding statement. Class Invariants are checked at instance method entry, exit and constructor exit. Method precondition is checked at instance method entry, constructor entry and private or static method entry. Method postcondition is checked at instance method exit, constructor exit and private or static method exit.

Any code generated from the precondition for a constructor is placed right after a super() or this() call, possibly before the rest of the statements. The java compiler forces this convention. In addition, the class invariant will not be enforced for any private methods, since a private method may temporarily modify the fields.

Contract Inheritance

Class hierarchies include class extension, interface implementation, interface extension and innerclasses. ContractChecker supports the propagation of class invariants, method preconditions and postconditions in class hierarchies. The class invariant of a sub-class are the conjunction of the class invariants of its superclasses or interfaces. The postconditions of a method in a subclass or subtype are the conjunction of the postconditions of the corresponding method in its superclass or interface. The preconditions of a method of a subclass, or subtype, are the disjunction of the preconditions of the corresponding method in its superclass or interface.

The same rule applies to an inner-class and its enclosing class.

The example show in Figure 3 further demonstrate how contract inheritance is handles by Contract Checker.

Instrumentation

Instrumentation is only performed for concrete classes or non-abstract methods in abstract classes.

```
/**
  * @invariant length()>0
*/
public interface Line {
  public int length();
  /**
   * @pre amount>0
   * @post cut! > 0
  */
  public int cut(int amount);
}

/**
  * @invariant length()<10
  */
class ShortLine implements Line {
  public int length() {}
  /**
   * @pre amount>-10
   * @post  cut! < 50
   */
  public int cut(int amount) {}
}

/**
  * @invariant length() >0 && length()<10
  */
class ShortLine implements Line {
  public int length() {}
  /**
    * @pre amount>-10 || amount >0
    * @post  cut! < 50   && cut! > 0
    */
  public int cut(int amount) {}
}
```

Figure 3. Exmaple for Contract Inheritance

@assert, @fact and @loopInvariant are simply transformed into assert statements at their existing location. @pre, @post and @invariant are transformed into if-else statements. If the contract is violated, an exception will be thrown and the violation will be reported to the trace.txt file.

If a postcondition contains the pre-stage/ after-stage value of an object, a temporary variable is introduced to preserved the original value. However, if the object is not cloneable, a ContractCannotBeCheckedException will be thrown. If a postcondition contains the return value a temporary variable will also be introduced before the return statement to represent the return value. The return statement will simply return the value of that variable.

For example, if we have a Contract specification for invariant:

```
@invariant exp (For class c)
```

The generated checking codes for entry and exit point of public method in class c is showing in Figure 4.

```
// Entry point for each public method in class c
if (! exp) {
  Contract contract = Contract.getInstance();
  contract.reportViolation(new InvariantException(...));
}
// Exit point for each public method in class c
if (! exp) {
  Contract contract = Contract.getInstance();
  contract.reportViolation(new InvariantException(...));
}
```

Figure 4. Instrumentation Sample Code 1

5 Static Contract Verifier

5.1 Static Contract Verification

The static contract verification is one of the functionality of Static Contract Verifier component in our DBC toolset. The goal of static analysis and this DBC tool is to replace runtime analysis as much as possible. Static analysis, unlike software testing, does not depend on the comprehensibility and completeness of specific test cases. Software testing using test cases has two primary drawbacks, the first of which is that preparing test cases can be a time-consuming process, and the second of which is that a series of test cases can only show program correctness for those specific cases according to specific coverage criteria. Static analysis, which does not rely on test cases, goes much further towards proving overall program correctness and adherence to a contract than software testing does. Unfortunately, static analysis cannot at this time completely replace software testing because in complex systems, static analysis becomes too inefficient.

The static contract verification functionality utilizes the DBC tags(Appendix A). This functionality will analyze specific methods and throw an InvalidContract exception if the implementation does not conform to the specification. The violation will be reported in the trace.txt file. The Static Contract Verifier makes use of the @assert, @fact, and @loopInvariant tags that may be included in the method body, but will attempt the analysis even without this information, the whole verification process can be conduct just using contractual interfaces like precondition, postconditions and invariant.

5.2 Our Approach

Our approach is based on the concept of weakest precondition as defined by [3]. The technique attempts to formulate certain global program properties from the class under examination and evaluates them with a theorem prover. These properties are classified as

class invariant. The class invariant are combined with the precondition for a variable under examination to evaluate the post condition. This way of formulating obligations inferred from the static properties of the program under examination and evaluating against a theorem prover shows the benefit of mathematically verifying the expected postcondition against the prevailing preconditions for each program property under examination. The theorem prover partially assists in checking the conditions used to control the logical flow of a program by evaluating the proof obligations formed from the path under examination.

The basic contract verification process comes from the well known Hoare Triple[4] as follows:

{P} S; {Q}

where S is a segment of statements; P is the already known precondition(s) which holds before execution of S; Q is the expected postcondition(s) which are to hold after execution of S In the above triple, the proof obligation to insure the correctness of the statement is

P => wp(S, Q)

where wp(S, Q) is a predefined function. By given a segment of statements S, and a postcondition Q to be held, wp(S, Q) returns the weakest precondition that need to held in order to make Q true after executing S

In order to construct proof obligation, P, S, Q and the function wp() must be available. Much research has been done about the wp() function[11] and the calculation in Java semantics[5][6][7]. Code segment S is explicitly given by the source code, {P} and {Q} are given by contract specification.

6 Comparison with Other DBC Tools

Since the concept of DBC was first introduced, many DBC tools have been created. Among them iContract, Eiffel, and JMSAssert are more commonly used. Using specified syntax with quantifier support are the common functionality provided by these tools including our toolset. However, using static analysis in contract verification is provided only by our toolset. The following is more detailed comparison of these tools.

Specification Expressiveness

The DBC toolset in our project brings a rich set of syntax. It supports not only the three key elements of DBC, but also extensions like the @assert, @fact and @loopInvariant. This is a highlighted feature compared with some DBC tools, such as JMSAssert [8]. Our toolset also has specific syntax for quantified expressions, which greatly improves the expressiveness.

Support of inter-method and inter-class

The DBC toolset in our project provides support for inter-method and inter-class contract verification.

Class Inheritance

The DBC toolset in our project provides an extensive JavaDoc support: the DocGen. Other DBC tools, such as JASS [9] or iContract [10] may also have JavaDoc support, but do not capture the inherited DBC information. There are also separate doclets that support DBC tags, such as iDoclet, but most are developed based on JDK version 1.3 or lower. DocGen, on the other hand, fully supports the inheritance of the contracts. It can be used separately, as well as with the ContractChecker.

Static Contract Verification

Static analysis, and particularly static contract verification, is perhaps the most important benefit of our DBC toolset when compared to other DBC products. Using automated theorem prover to verify the contract specification, runtime violations can be effectively detect in static checking. iContract does not include static analysis while JMSAssert contains a utility called JStyle that does do limited static anomaly detect, which serves as more of an automatic code review than an analysis of code correctness. JStyle helps to catch problems such as class or member naming conventions, class section ordering, and unsafe exception handling [10]. This can be a useful utility, but it does not provide as much value as static contract verification, which can statically analyze whether the implementation conforms to the specification or not.

7 Conclusions and Future Work

Design by Contract yields great benefits in software development. Our DBC toolset brings DBC into Java, and provides both dynamic and static contract verification which makes it possible to formalize the DBC information, and to utilize DBC during for both runtime and static checking.

Dynamic contract verifier ContractChecker creates test code based on the DBC information. The test code is inserted into the original code. This process is independent of compiling and executing the code. At runtime, if a contract is violated, a particular exception is thrown and reported to alert the programmer.

Static contract verifier Static Contract Verifier checks for null pointer and array out of bounds errors and utilizes Design by Contract information to check the correctness of a written contract. If a contract is not assured of fulfilled postconditions given satisfied preconditions Static Contract Verifier will identify this and alert the programmer of the problem.

Despite the advantages of our DBC toolset, it does has some limitations. For example, the test fails if

a recursive call occurs during the check. This problem can be solved by using a register file or stack, to track the occurrence of checks, and make to sure that finite number of checks is performed. Although the syntax of quantified expressions are specified, the toolset does not generate correct code for them. In the future, this can be achieved by introducing a loop statement through the range of the target variable. Our future work will be focus on eliminating these limitations.

References

[1] B. Meyer *Applying Design by Contract* Computer, October vol25, no10, pp.40-51. 1992

[2] B. Meyer *Building bug-free O-O software: An introduction to Design by Contract,* Object Currents, SIGS Publication, Vol. 1, No. 3, March 1996

[3] D. Gries *The science of Programming.* Springer-Verlag, 1981

[4] A. Hoare *An Axiomatic Basis for Computer Programming.* Communications of ACM, 12(10), pp576-580, 1969

[5] B. Jacobs, E. Poll, *A Logic for the Java Modeling Language JML*

[6] B. Jacobs, *Weakest Precondition Reasoning for Java Programs with JML Annotations.*

[7] S. Skevoulis *A Light-weight Approach to Applying Formal Methods in Software Development*, Technical Report, DePaul University, 1999

[8] S. Skevoulis, X. Jia *Generic Invariant-Based Static Analysis Tool for Detection of Runtime Errors in Java Programs*,

[9] R. Kramer, *Examples of Design by Contract in Java,* Object World Berlin 99, Berlin, May 1999

[10] K. Rangaraajan *Invasive Testing of JavaTM Classes*, Man Machine Systems

[11] E. Dijkstra, Guarded Commands em Nondeterminacy and Formal Derivation of Program. Communications of ACM, 18(8), pp 453-458, 1975

[12] C. Fischer, *Combination and Implementation of Processes and Data: from CSP-OZ to Java* PhD Thesis, University of Oldenburg, 2000

[13] B. Meyer *Object-Oriented Software Construction* Prentice Hall, 1988

[14] B. Meyer, *Toward More expressive contracts,* Journal of OO Programming (JOOP), July-August 2000, pp. 39-43.

[15] G. Pomberger , G. Blaschek *Object-Orientation and Prototyping in Software Engineering,* Prentice Hall, The object-oriented series, Hemel Hempstead, 1996

[16] K. Rangaraajan *Critiquing Java Source Code*

A Syntax for Expression

Quantified Expressions

Forall and exists expressions are produced through the following EBNF.

```
forallExpression ::=
   forall { variableDeclaration ( ; variableDeclaration )*
            ( : expression )? @ expression }
existsExpression ::=
   exists { variableDeclaration ( ; variableDeclaration )*
            ( : expression )? @ expression }
```

B Custom Tags

In the traditional DBC world, the three well known kinds of contracts are method precondition, method postcondition and class invariant. They correspond to these custom tags: @pre, @post and @invariant Method preconditions and postconditions are allowed only before the method to which they apply. A class invariant applies to a Java class as a whole, so it is allowed before a Java class. In practice, however, most class invariants contain information about the class fields. For programmers convenience, an invariant tag is also allowed to appear before a field.

In addition to these standard tags, we expand the family of custom tags used to specify the contract information. Three more tags are introduced. These tags are @assert, @fact, and @loopInvariant. @assert and @fact are allowed before a statement. @loopInvariant is allowed before a loop.