Automated Validation of Class Invariants In C++ Applications

Tanton H. Gibbs, Brian A Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
USA.
{thgibbs,malloy}@cs.clemson.edu

James F. Power
Computer Science Department
National University of Ireland
Maynooth, Co. Kildare
Ireland
James.Power@may.ie

Abstract

In this paper, we describe a non-invasive approach for validation of class invariants in C++ applications. Our approach is fully automated so that the user need only supply the class invariants for each class hierarchy to be checked and our validator constructs an InvariantVisitor*, a variation of the Visitor Pattern, and an* InvariantFacilitator*. Instantiations of the* InvariantVisitor *and* InvariantFacilitator *classes encapsulate the invariants in C++ statements and facilitate the validation of the invariants. We describe both our approach and our results of validating invariants in* keystone*, a well tested parser front-end for C++.*

1 Introduction

The current focus in the software industry is to improve the quality rather than the speed or size of an application. A report by the Workshop on Strategic Directions in Software Quality asserts that software quality will become the dominant success criterion in the software industry [23]. One process that supports software quality is *testing*, which exercises the software to gather information about the software. However, studies indicate that testing consumes more than fifty percent of the cost of software development, with some estimates placing the cost even higher [12].

An alternative to testing that promises to improve software quality is *Design by Contract*, which advocates establishing and checking assertions about the attributes of a class. Design by Contract was originally advocated by Alan Turing [15], further developed by Hoare [13, 14], Floyd [8] and Dijkstra [7] and made popular in the Eiffel language [22]. However, Eiffel has not garnered the wide acceptance of other object-oriented languages such as C++ and Java. Nevertheless, the importance of the Design by Contract approach is noteworthy, and one of the hypotheses of our work is that the Design by Contract approach may complement software testing by exposing errors that are not found by implementation-based or specification-based testing.

In this paper, we describe a non-invasive approach for validation of class invariants in C++ applications. We use the Object Constraint Language (OCL) to specify the invariants but our approach is amenable to other specification languages such as Z [28], Object-Z [27] or VDM [17]. Our approach is fully automated so that the user need only supply the class invariants for each class hierarchy to be checked and our validator constructs an *InvariantVisitor*, a variation of the Visitor Pattern [2, 21], and an *InvariantFacilitator*. Instantiations of the *InvariantVisitor* and *InvariantFacilitator* classes encapsulate the invariants in C++ statements and facilitate the validation of the invariants. We describe both our approach and our results of validating invariants in *keystone*, a well tested parser front-end for C++ [20, 24, 25].

In the next section we provide background about the Visitor pattern, class invariants, OCL and *keystone*. In Section 3, we overview our approach including a description of the *InvariantVisitor* and the *InvariantFacilitator* classes. In Section 4 we describe our incorporation of invariant validation into *keystone*, and in Section 5 we benchmark our validator and provide interesting insights into the validation of *keystone*. In Section 6 we review related work and in Section 7 we draw conclusions.

Accepted for the 17th IEEE International Conference on Automated Software Engineering
Edinburgh, UK, September 23-27, 2002, pp. 205-214

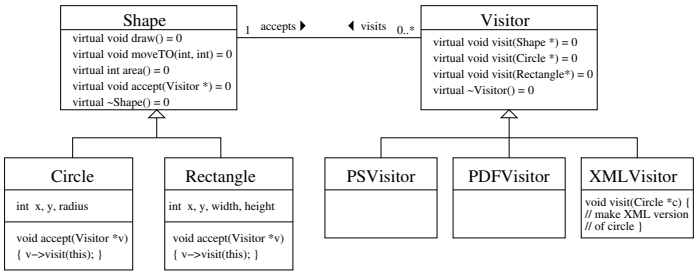

Figure 1. *The Visitor Pattern*. **The Visitor pattern consists of two inheritance hierarchies, an** *element hierarchy*, **illustrated on the left by the** *Shape* **hierarchy, and a** *Visitor hierarchy*, **illustrated on the right.**

2 Background

To facilitate automation of invariant validation, we exploit a variation of the Visitor pattern, the Acyclic Visitor [2, 21], which we describe in Section 3. In this section we first describe the Visitor pattern [9, 30], and then provide background about invariants and the object constraint language, OCL. We conclude this section with an overview of *keystone*, an application that provides a parser front-end for the ISO C++ language; we use *keystone* as our case study in Sections 4 and 5.

2.1 The Visitor Pattern

The Visitor pattern permits the designer of a class hierarchy to add functionality without "polluting" the hierarchy with unrelated operations [9]. The Visitor pattern can also be used to obviate *down casting* by enabling operations that depend on the concrete classes [30]. These class dependent operations, as well as additional operations, can be added to an existing class hierarchy without modifying the classes in the hierarchy.

The Visitor pattern consists of two inheritance hierarchies, an *element hierarchy* and a *Visitor hierarchy*. We use the familiar Shape example [2, 9, 22, 29] as our element hierarchy, illustrated on the left of Figure 1, and a Visitor hierarchy illustrated on the right of the figure. The Shape hierarchy consists of an abstract base class with five methods and two derived classes for specializing Shape. Four of the five methods in class Shape represent typical operations that one might perform on all shapes, such as drawing, moving, finding the area or deleting a Shape. The fifth method is accept(Visitor *), which permits a visitor object to visit a particular element.

An instantiation of a particular visitor encapsulates additional functionality to augment the functionality already in the Shape hierarchy. When an instance of a class derived from Shape calls visit on the visitor object, the instance effectively identifies its type to the visitor. The visitor operation that is called can perform whatever action is appropriate for the particular instance. For example, an instance of XMLVisitor can build an XML representation of a Circle, as illustrated by the XMLVisitor class shown on the right side of Figure 1.

2.2 Class invariants

An invariant on a class C is a set of Boolean conditions or predicates that every instance of C will satisfy after instantiation (i.e., after constructor invocation) and before and after every method invocation by another object [22]. A class invariant is a property of a class instance that must be preserved by all methods of the class. In spite of its name, an invariant is not required to hold at all execution points. For example, a method might violate the invariant while working toward its goal; however, the invariant must be re-established before the method terminates execution.

Class invariants are used to ensure that the operations performed on instances of the class maintain the integrity constraints of the class. These constraints are described in terms of the member functions and data attributes of the class.

2.3 The Object Constraint Language (OCL)

A Unified Modeling Language (UML) diagram, such as a class diagram, provides a high level of program

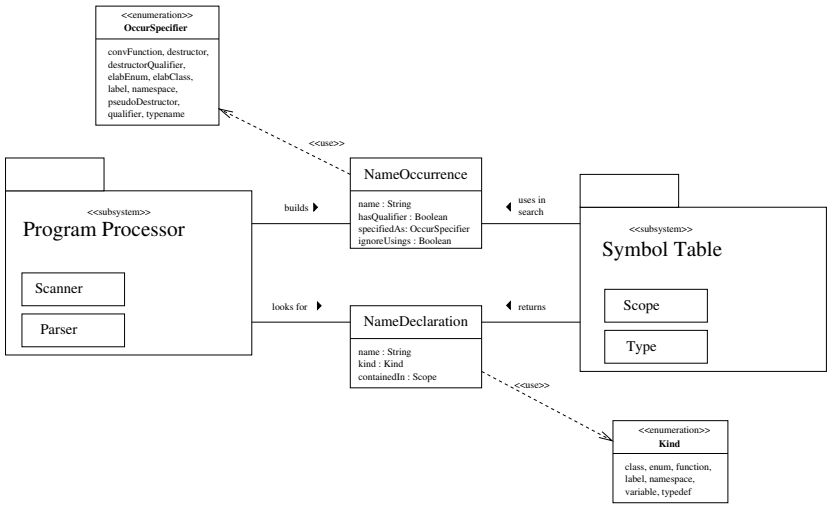

Figure 2. Keystone summary. The Program Processor **subsystem, illustrated on the left, is responsible for directing symbol table construction and name lookup. The** Program Processor **marshals information about the name in a** NameOccurrence **object and directs the search for a corresponding** NameDeclaration **in the** Symbol Table **subsystem, illustrated on the right.**

abstraction [4]. These UML diagrams are not refined enough to describe low-level aspects of a specification, such as invariant conditions that must hold for instances of objects in the system. The Object Constraint language (OCL) is a formal language used to describe expressions on models specified in the Unified Modeling Language (UML) [3, 34]. These expressions typically specify invariant conditions that must hold for the system being modeled, or queries over objects described in a model. OCL expressions, when evaluated, do not have side effects, so their evaluation cannot alter the state of the corresponding executing system even though an OCL expression can be used to specify a state change. OCL expressions allow the modeler to express invariants in a language independent manner.

2.4 The keystone parser front-end

Figure 2 summarizes the design of our case study application, a parser front-end for ISO C++ [16]. The figure presents two subsystems, illustrated as tabbed folders and designated by the ≪subsystem≫ stereotype. The Program Processor subsystem is shown on the left and the Symbol Table subsystem is shown on the right of Figure 2.

The Program Processor subsystem includes a Scanner and Parser and is responsible for initiating and directing symbol table construction and name lookup. This responsibility includes two phases: (1) assembling the necessary information for creation of a NameOccurrence object, and (2) directing the search for a corresponding NameDeclaration object in the Symbol Table subsystem. The Symbol Table subsystem is the symbol table in the parser, including class hierarchies for type information, Type, and scope information, Scope, shown on the right of Figure 2. The Symbol Table subsystem, the target of our validation effort, is discussed further in Section 4.

3 Overview of the Technique

In this section, we use the Shape example from Section 2 to illustrate our approach for automating validation of class invariants. We begin with an overview of the approach and in Section 3.2 we list some sample invariants in OCL for the Shape example. In Sections 3.3 and 3.4 we describe the InvariantVisitor and InvariantFacilitator classes that are automatically generated in our system.

3.1 The validator system

Figure 3 provides an overview of our validator, with input to the system shown on the left and output shown

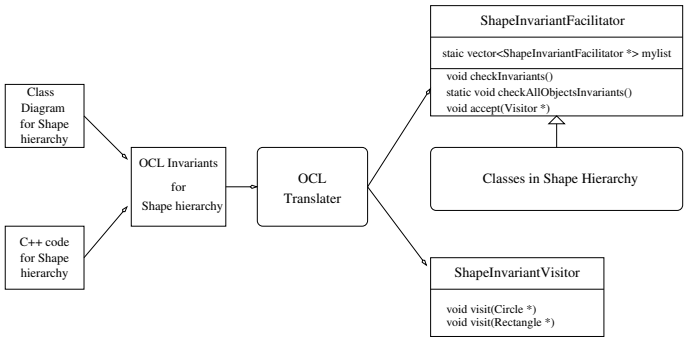

Figure 3. *System overview*. **This figure illustrates the central module in our validator, the** OCL Translator**, which takes an OCL specification of the invariants for a class hierarchy and produces two classes as output, a facilitator class and a visitor class. These classes are listed in Figures 4 and 6, and explained in Section 3.**

on the right of the figure. The central module is the OCL Translator, shown in the center of the figure. Our OCL Translator is a recursive-descent parser for OCL, with semantic actions inserted into the parse to construct the ShapeInvariantFacilitator and the ShapeInvariantVisitor classes, shown on the right of the figure.

The only modification required in the class hierarchy under validation is to make the ShapeInvariantFacilitator a super class. An instantiation of ShapeInvariantFacilitator implements the Acyclic Visitor pattern, which we discuss later in this section.

OCL expressions are specified in terms of a UML model, including the attributes and associations of a class. Our choice of OCL as our specification language enables us to reuse our design artifacts to express invariants on the classes in our system. The invariants that we report in this paper are extracted from our C++ code; however, as we extend our target application, we expect that our invariants may also be part of the design of the system. These two options are illustrated on the left of Figure 3.

```
1 context Circle
2 inv: self.Area() = PI * self.Radius() *
3       self.Radius()
4 inv: self.Diameter() = 2 * self.Radius()

5 context Rectangle
6 inv: self.Area() = self.Width() * self.Height()
```

Figure 4. OCL invariants for Shape

3.2 Invariants for Shape

To provide a flavor of the OCL expressions, we illustrate the invariants for the Shape hierarchy in Figure 4. The phrase **context** Circle on line one in the figure provides the context in which the invariants on lines 2–4 are applied. The invariant on lines 2–3 of the figure is an assertion about the area of a Circle and the invariant on line four is an assertion about the diameter of a Circle. Similarly for the Rectangle on lines five and six of the figure.

```
1 class ShapeInvariantVisitor : public Visitor {
2  virtual void visit( const Circle* self ) {
3   assert( self->Area() == PI*self->Radius()*
4     self->Radius() );
5   assert( self->Diameter() == 2*self->Radius()*
6  }
7  virtual void visit( const Rectangle* self ) {
8   assert( self->Area() == self->Width() *
9     self->Length() );
10  }
11 };
```

Figure 5. The Visitor class

3.3 The invariant visitor

Figure 5 presents the ShapeInvariantVisitor that is generated by our OCL translator for the invariants listed in Figure 4. Each invariant is implemented as an assertion, shown on lines 3, 5 and 8 of the figure. Our OCL

```
1  class ShapeInvariantFacilitator {
2  public:
3    ShapeInvariantFacilitator() {
4      s_ptrs.push_back(this);
5    }
6    virtual ~ShapeInvariantFacilitator() { }
7    void checkInvariants() const {
8      ShapeInvariantVisitor visitor;
9      accept(&visitor);
10   }
11   static void checkAllClassesInvariants() {
12     ShapeInvariantVisitor visitor;
13     std::for_each( s_ptrs.begin(),
14       s_ptrs.end(), std::mem_fun(
15       &ShapeInvariantFacilitator::checkInvariants
16       ) );
17   }
18 private:
19   virtual void accept( Visitor* ) const;
20   static std::vector<ShapeInvariantFacilitator*>
21      s_ptrs;
22 };
23 void ShapeInvariantFacilitator::accept(
24         Visitor* v ) const {
25   if( const Rectangle* self =
26         dynamic_cast<const Rectangle*>(this) ) {
27     v->visit(self);
28   }
29   if( const Circle* self =
30         dynamic_cast<const Circle*>(this) ) {
31     v->visit(self);
32   }
33 }
```

Figure 6. The Facilitator class

translator uses the Perl Parse::RecDescent module, together with an OCL grammar to automatically translate the OCL into C++.

3.4 InvariantFacilitator: an Acyclic Visitor

Our translator generates an InvariantFacilitator for each class hierarchy under consideration, which accomplishes several tasks. First, the InvariantFacilitator includes a method to check the invariants for the current object, illustrated as checkInvariants() on lines 7–10 of Figure 6. Second, the InvariantFacilitator includes a method, checkAllClassesInvariants(), lines 11–17, that iterates through the static vector of objects, shown on lines 20–21 of the figure. Using the checkAllClassesInvariants() method, the user may check the invariants for a class hierarchy at various points during program execution. Finally, the InvariantFacilitator includes method accept(), shown on lines 23–33 of Figure 6, which obviates the inclusion of an accept method in each class in the hierarchy under validation. Moreover, this accept method uses dynamic type information to remove the cyclic nature of the classic Visitor pattern [9]. Thus, the InvariantFacilitator implements the Acyclic Visitor.

4 Validation in Keystone

In this section, we describe our validation of *keystone*. We begin with a description of the class hierarchies that we validated and motivate the importance of automating the validation process. In Sections 4.2 and 4.3, we describe the symbol table and some of the invariants on the symbol table for *keystone*.

4.1 The Keystone Hierarchies

In Section 2.4 we overviewed the structure of *keystone* including the Program Processor and the Symbol Table subsystems. *Keystone* is a research project whose goal is the design and implementation of a parser front-end that will parse the language described by the ISO C++ standard [16], including an implementation of name lookup. *Name lookup* is the process of finding, for each occurrence of a name in a program, the corresponding declaration of that name. Thus, the more important of the *keystone* subsystems is the Symbol Table subsystem, where names are stored as instances of Scope or NameDeclaration, and there may be a Type associated with a name. The NameDeclaration, Type and Scope classes are illustrated in Figure 2 and details of the Scope hierarchy are illustrated in Figure 7.

Figure 7 shows a base class, Scope, and five derived classes for some of the specializations of Scope that can occur in a C++ program. The most notable omission from the Scope hierarchy is a specialization for *template* scope. Our ongoing work includes the incorporation of templates into the *keystone* front-end. These prospective changes underscore the importance of automatically validating invariants because, as *keystone* evolves, additional invariants will be incorporated into the system, and some existing invariants may be updated. The automated generation of the InvariantVisitor and InvariantFacilitator classes from the OCL specification facilitate repeated validation of the invariants as well as the maintenance of consistency of the implementation with the specification. Our case study entailed validation of the NameDeclaration, Type and Scope hierarchies.

Accepted for the 17th IEEE International Conference on Automated Software Engineering
Edinburgh, UK, September 23-27, 2002, pp. 205-214

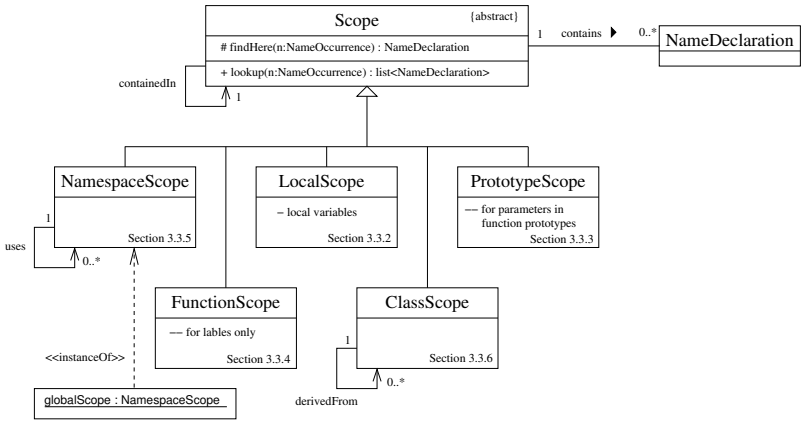

Figure 7. *Class Diagram for the Scopes Hierarchy*. **Each** Scope **object contains a list of** NameDeclaration **instances, along with name lookup functionality. The** Scope **class has six subclasses, as detailed in the referenced sections of the ISO standard.**

4.2 The Keystone Symbol Table

One of the characteristics of *keystone* is that once a name is installed in the symbol table, the typical operation on that name is lookup and other query operations. Once installed in the symbol table, instantiations of NameDeclaration, Scope and Type are generally not modified, except for an occasional back-patching of type information for class attributes.

Thus, we validate *keystone* invariants at the end of the parse of an input program. This entails placing a call in `main()` to the function `checkAllClassesInvariants` in the `InvariantFacilitator` class of each inheritance hierarchy being validated (see Figure 6). This call can be placed anywhere in the code; however, with *keystone*, we only validate invariants at the end of the program.

An alternative to validating invariants at the end of the program is to validate invariants (1) after the execution of the body of a constructor, (2) before the execution of the body of a destructor, (3) as part of the pre-conditions of a method, and (4) as part of the post-conditions of a method. Given the stability of objects in the *keystone* symbol table, we are able to obtain assurance about invariants through validation at program termination. However, our non-invasive approach does not easily extend to applications whose objects are not stable, but are created and destroyed during program execution.

```
1 context Scope
2 inv: self.getDecl() <> NULL implies
3       self.getDecl().getCorrespondingScope() = self
4 inv: self.getContainingScope() <> NULL xor
5       self.getName() = "_GlobalNamespace"
6 inv: self.getLocals()-> forAll(
7      n:NameDeclaration |
8      n.getContainingScope() = self )

9 context ClassScope
10 inv: self.getLocals()-> forAll(
11       n:NameDeclaration
12       | n.getType() <> Type::namespaceType )
13 inv:
14   self->getContainingScope().oclIsTypeOf(NamespaceScope)
15    or self->getContainingScope().oclIsTypeOf(ClassScope)
16    or self->getContainingScope().oclIsTypeOf(LocalScope)
```

Figure 8. Invariants: Scope & ClassScope

4.3 OCL Invariants for Scope Hierarchy

Figure 8 provides some invariants for the Scope and ClassScope classes in *keystone*. The invariants for ClassScope are the conjunction of its invariants and the invariants of the parent, Scope. The invariant on line 2 of Figure 8 states that if the NameDeclaration for the current scope is not NULL, then the corresponding scope of the NameDeclaration is self, the current scope. Since no C++ class can contain a namespace, line 10 states that none of the NameDeclaration objects in the current class scope may be namespaces. This invariant uses the OCL *forAll* construct, which requires

Test case	lines	classes	classes w/ fns
encrypt	946	1	1
Clause 3	952	40	34
php2cpp	1,920	6	6
fft	2,238	51	36
graphdraw	4,354	199	76
ep matrix	4,944	78	51
taxonomy	5,322	573	471
vkey	8,556	279	44

Figure 9. *Test suite*.

that our translator generate a function containing a **for** loop that iterates through the **vector** of NameDeclaration objects local to the class scope.

5 Results

In this section we describe the results of our study of automated validation of class invariants. The target application for our study is *keystone* [20, 24, 25], a parser front-end for the ISO C++ language [16]. The validator was executed on a Dell Precision 530 workstation with a Xeon 1.7 GHz processor equipped with 512 MB of RDRAM, running the Red Hat Linux 7.1 operating system. Our implementation languages were C++ [29] and Perl [32], compiled with GNU *gcc* version 2.96 and the Perl interpreter version 5.6.0. In the next section we describe the test suite for the study and in Section 5.2 we measure the number of objects for which invariants were validated as well as the number of invariants executed. In Section 5.3 we describe the impact on performance due to the invariant checking and, in 5.4 we present the improvement in *keystone* through invariant validation.

5.1 The test suite

The table in Figure 9 summarizes our suite of eight test cases, listed in the rows of the table as encrypt, Clause 3, php2cpp, fft, graphdraw, ep matrix, vkey and the taxonomy testsuite. The test cases in the suite were chosen because of their range and variety of application; they are listed in sorted order by number of lines of code, not including comments or blank lines. We note that keystone had been previously tested using this same test suite and was thought to run successfully.

Test case encrypt is an encryption program that uses the Vignere algorithm [1] and Clause 3 is a sequence of examples taken from Clause 3 of the ISO C++ standard [16]. The php2cpp test case converts the PHP web publishing language to C++ [5] and fft performs fast Fourier transforms [18]. graphdraw is a drawing application that uses *IV Tools* [31], a suite of free XWindows drawing editors for Postscript, TeX and web graphics production. The ep matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable C++ number theory library [26]. vkey is a GUI application that uses the *V GUI* library [33], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications. The taxonomy testsuite is a validation suite for a taxonomy that describes classes in object oriented languages [6].

The columns of the table in Figure 9 list details about the number of lines of code, not including comments or blank lines, the number of classes, and the number of classes with functions for each of the test cases. All of the test cases are complete applications, except Clause 3 and taxonomy, and three use large libraries: ep matrix, vkey and graphdraw use the *NTL*, *V GUI* and *IV Tools* libraries respectively.

5.2 Number of objects and invariants checked

Figure 10 summarizes the number of objects and invariants that are checked during *keystone*'s parsing of the test suite. The rows of the table list the test cases and the columns list the data acquired by monitoring the InvariantFacilitators. The columns labeled **Scope Objects**, **Type Objects** and **NameDeclaration Objects** list the number of validated objects in the Scope, Type and NameDeclaration hierarchies respectively. The **Invariants** column is a summation of the invariants validated for all three hierarchies and shows that a large number of invariants are checked for some of the test cases. For example, the graphdraw program, listed in the fifth row of Figure 10, required 203,234 invariants to be validated.

5.3 Efficiency

In Section 4, we described options for placement of invariant checks in applications and provided a rationale for our evaluation of invariants at the end of the program, saving all objects in a static vector local to the facilitator class. Nevertheless, in view of the large number of invariants that are validated in some of the test cases, such as the graphdraw and ep matrix programs, we conducted some timings to measure the cost of our automated approach.

Test case	Scope Objects	Type Objects	NameDeclaration Objects	Invariants
encrypt	634	1,769	1,534	21,445
Clause 3	430	1,436	653	10,132
php2cpp	570	2,021	2,229	29,909
fft	1,105	3,868	4,069	54,906
graphdraw	5,518	18,342	14,488	203,234
ep matrix	5,071	15,300	11,334	161,450
taxonomy	3,778	7,255	4,868	73,227
vkey	2,097	12,978	12,131	162,744

Figure 10. *Results of Study*. **The rows of the table list the number of scope, type, and name declaration objects validated for each test case. The final column summarizes the number of invariants checked for each test case.**

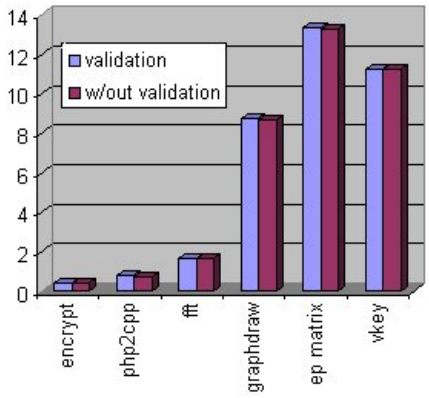

Figure 11. *Cost of validation*.

Figure 11 depicts running times for six of the eight test cases. There is a pair of bars for each of the six test cases timed, with the left bar of the pair representing the execution time including invariant validation, and the right bar of the pair representing the execution time without invariant validation. The ep matrix program required the longest execution time, using 13.33 seconds to parse the ep matrix program and validate the 161,450 invariants, and 13.26 seconds to parse the program without validating invariants. All times were wall clock times. We recorded twelve timings for each program, discarded the lowest and highest value; the times in Figure 11 are the averages of the ten remaining executions.

5.4 Impact of validation on keystone

Our goals in automating invariant validation are to measure the cost of validation, to establish a level of confidence in our software and to determine if the code is consistent with our design documents. Before validation, we manually checked *keystone*'s behavior after parsing each of the testsuite's source files. However, we found the manual check of class invariants to be time consuming and error prone. By automating the invariant validation in *keystone*, we uncovered 11 previously unknown errors, 5 were OCL errors that represented a mismatch between the design and implementation of *keystone*, and 6 were implementation errors. The 5 OCL errors were repaired during validation. For the 6 implementation errors, 3 were repaired during validation and the remaining 3 have been placed in a database of known errors.

The *invariant inheritance rule* states that the invariant of a class is the *Boolean* **and** of the invariant of the class with the invariant of its parent, if the class has a parent [22]. This rule raises the possibility of inconsistency between the invariant of a class and its parent. We found no inconsistencies in combining the invariants in the *keystone* inheritance hierarchies.

Some of the invariants that we validated include variables whose values are stored in private data members. Our *keystone* application contains accessor methods for all relevant private data attributes; this accessibility is required for successful application of our technique.

6 Related Work

In this section, we overview some of the work related to invariant validation and support of Design by Contract in C++; a more thorough overview can be found in reference [19]. This support falls into one of four categories: (1) use the language constructs to provide support, (2) use macros, (3) extend the syntax of the language and implement the extensions through a pre-processor, and

(4) propose a language extension. Each of the categories (2), (3) and (4) lacks the orthogonality of category (1), and makes it difficult to separate the assertions from the ordinary source code. In addition, (2) uses a mechanism more properly reserved for conditional compilation and deprecated in C++ for other purposes. Categories (3) and (4) have the disadvantage of being non-standard additions to C++, whereas the trend, at least since the publication of the ISO standard, has been towards convergence between C++ dialects. Further, none of the categories (2), (3) or (4) can interact well with a source-level debugger, since the C++ code being executed differs from the code written by the programmer. Our work falls into the first category and the papers that we review in this section describe approaches similar to ours.

Reference [19] presents an approach to emulating Design by Contract in C++ that uses VDM-SL [17] as the specification language. The approach represents an attempt to address the shortcomings of a Graphical R-Matrix Atomic Collision Environment (GRACE), which makes extensive use of the Standard C++ Library (STL). A library of STL "lookalikes" is constructed and users of the system publicly derive thier STL containers from the "lookalikes". To validate invariants, the user writes a function called *inv* that contains the invariants and returns a Boolean indicating whether or not the invariants were satisfied. The technique is not automated.

Reference [10, 11] presents a framework for Design by Contract that requires the programmer to supply functions that check pre-conditions, post-conditions and class invariants. The technique is intrusive and cannot be automated without parsing and modifying the program.

7 Concluding Remarks

In this paper, we describe a non-invasive approach to validation of class invariants for C++ applications. Our case study of *keystone* shows that our approach has minimal impact on the execution time of *keystone* and there are important benefits in validating invariants. First, there are several kinds of errors that are exposed by our approach. In particular, we are able to expose a mismatch between the specification and the implementation. Also, by validating invariants, we expose inconsistencies in the code that currently do not produce incorrect output. Second, our automated approach permits us to validate many more invariants than one could reasonably validate manually and to avoid errors that might result from manual validation. Third, by automating the validation we can generate *InvariantVisitor* and *InvariantFacilitator* classes for modified class hierarchies as well as new hierarchies, so that validation becomes a maintenance activity.

There are threats to both our approach and our case study. First, in our approach, invariants are validated dynamically and the potential of the invariants to expose errors is necessarily dependent on the coverage provided by the test suite. If the test suite does not adequately cover the code, than our invariant validation is weakened. Second, our case study involved an application whose objects, once created and installed in a symbol table, remain relatively stable. For an application where the objects in the system are frequently updated, the invariants must be validated more frequently with a corresponding rise in the cost of validation.

Acknowledgement
We would like to thank the anonymous referees for their helpful comments, which improved several sections of our paper.

References

[1] S. Alexander. The C++ resources network. http://www.cplusplus.com, October 2001.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] Boldsoft, Rational Software Corp, IONA and Adaptive Ltd. Response to the UML 2.0 OCL RfP. Technical report, OMG Document ad/2002, March 1 2002.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[5] F. J. Cavalier. Debugging PHP using a C++ compiler. *Dr. Dobbs Journal*, pages 42–46, March 2002.

[6] P. Clarke and B. A. Malloy. A unified approach to implementation-based testing of classes. In *Proceedings of 1st Annual International Conference on Computer and Information Science (ICIS '01)*, Orlando, Florida, USA, October 3-5 2001.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[8] R. Floyd. Assigning Meanings to Programs. *Proceedings of American Mathematical Society Symposium on Applied Mathematics*, 19:19–31, 1967.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] P. Guerreiro. Another mediocre assertion mechanism for C++. In *Technology of Object-Oriented Languages and Systems*, pages 226–237, St. Malo, France, June 2000.

[11] P. Guerreiro. Simple support for design by contract in C++. In *Technology of Object-Oriented Languages and Systems*, pages 24–34, Santa Barbara, CA, USA, August 2001.

[12] M. J. Harrold. Testing: A roadmap. *Proceedings of the International Conference on Software Engineering*, June 2000.

[13] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, February 1969.

[14] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[15] C. Hoare. The Emperor's Old Clothes (1980 Turing Award Lecture). *Communications of the ACM*, 24(2):75–83, February 1981.

[16] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[17] C. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.

[18] O. Kiselyov. Fast Fourier transform. *Free C/C++ Sources for Numerical Computation*, March 2002. http://cliodhna.cop.uop.edu/~hetrick/c-sources.html.

[19] D. Maley and I. Spence. Emulating design by contract in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–75, Nancy, France, June 1999.

[20] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *submitted*, pages 1–33, 2002.

[21] R. Martin. Acyclic visitor. Technical Report wucs-97-07, Washington University Technical Report, September 4–6 1996.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, second edition, 1997.

[23] L. J. Osterweil and et al. Strategic directions in software quality. *ACM Computing Surveys*, 4:738–750, December 1996.

[24] J. F. Power and B. A. Malloy. An approach for modeling the name lookup problem in the C++ programming language. In *ACM Symposium on Applied Computing, SAC'2000*, pages 792–796, Como, Italy, March 2000.

[25] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in iso C++. In *Technology of Object-Oriented Languages and Systems, TOOLS 2000*, pages 57–68, Sydney, Australia, November 2001.

[26] V. Shoup. Number theory library. http://www.shoup.net/ntl/, March 2002.

[27] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[28] J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics*. Cambridge University Press, 1992.

[29] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[30] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

[31] J. M. Vlissides and M. A. Linton. IV tools. http://www.vectaport.com/ivtools/, March 2002.

[32] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl: Third Edition*. O'Reilly & Associates, third edition, 2000.

[33] B. Wampler. The V C++ GUI framework. http://www.objectcentral.com, October 2001.

[34] J. Warmer and A. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison-Wesley, first edition, 1999.