TO APPEAR IN SOFTWARE TESTING, VERIFICATION AND RELIABILITY
This version prepared on May 31, 2005

Exploiting design patterns to automate validation of class invariants

Brian A. Malloy[1]* and James F. Power[2]

[1] *Computer Science Dept., Clemson University, Clemson, SC 29634, USA.*
[2] *Computer Science Dept., National University of Ireland, Maynooth, Co. Kildare, Ireland.*

SUMMARY

In this paper, techniques are presented that exploit two design patterns, the Visitor pattern and the Decorator pattern, to automatically validate invariants about the data attributes in a C++ class. To investigate the pragmatics involved in using the two patterns, a study of an existing, well-tested application, *keystone*, a parser and front-end for the C++ language, is presented. Results from the study indicate that these two patterns provide flexibility in terms of the frequency and level of granularity of validation of the class invariants, which are expressed in the Object Constraint Language, OCL. The quantitative results measure the impact of these approaches and the additional faults uncovered through validation of the case study.

KEY WORDS: Class invariants, validation, design patterns, Object Constraint Language

1. Introduction

A report published in 1996, by the Workshop on Strategic Directions in Software Quality, asserts that software quality will become the dominant success criterion in the software industry [1]. A more recent report published in 2002, by the National Institute of Standards and Technology, suggests that software quality has not yet reached adequate standards [2]. The cost of software system nonperformance and failure is expensive, with sometimes catastrophic impact on property and life. Due to the resulting negative impact from system failure, the trend in the development of software systems has shifted toward attempts to construct reliable, robust models, whose mean time to failure is months or years rather than hours or days.

One process that supports the construction of robust software is testing, where the program is executed with input data, or test cases, and the output data is compared to expected results.

*Correspondence to: Brian Malloy at `<malloy@cs.clemson.edu>`

However, many of the techniques developed in testing research do not scale to real programs and many are too difficult for the average practitioner to grasp; thus, many of the testing techniques developed by researchers have not been adopted by developers. The inadequacy of the testing infrastructure has been well documented, including quantitative measurement of the impact of inadequate testing for both developers and industry users of software [3, 2].

One approach that can augment the testing process and thereby improve software quality entails the use of assertions to monitor the data attributes of functions or classes. The use of assertions was originally advocated by Alan Turing [4], further developed by Hoare [5, 6], Floyd [7] and Dijkstra [8]. At its simplest, an assertion is a logical predicate that must always be true at a given point in the program. As originally envisaged by Floyd and Hoare, such assertions can be used for function pre- and post conditions and for loop invariants, and can form the basis for (static) program verification.

A more pragmatic approach represents assertions as boolean-valued expressions in the program code, and checks that they are true at run-time. The authors of this paper refer to this process as *validation*, since the assertions can be regarded as part of the software requirements. This paper uses *validation* in preference to *verification* to distinguish the approach presented here from formal verification techniques, such as program derivation and model checking. The effectiveness of such validation is closely related to the effectiveness of the corresponding testing strategy. In an object-oriented context, it is usual to establish assertions on the data attributes of classes, and to demand that these are not violated by public methods. Such assertions are referred to as *class invariants* or *invariants*, and have been incorporated directly into the Eiffel programming language [9]. However, Eiffel has not gained the wide acceptance of other object-oriented languages such as C++ and Java.

Rosenblum provides an excellent study of the use of invariants in C programs [10]. While there is basic support for run-time assertion validation as part of the C++ standard library, the language does not directly support the validation of class invariants. Existing approaches to add such functionality to C++ programs include language extensions, preprocessors and the elaborate use of macros. In general these techniques do not interact well with other programming tools, tend to be intrusive and cannot be automated without parsing and modifying the program.

In this paper, techniques are presented that exploit two design patterns, the Visitor pattern and the Decorator pattern, to automate validation of assertions about the data attributes in C++ classes or class hierarchies [11]. The approach presented in this paper is relatively non-intrusive, does not require a pre-processor for the application and can be applied either during development or as a maintenance activity. The use of design patterns enables the architecture of the developed system to be smaller, simpler and more understandable than a system developed without the use of patterns. Moreover, developers who are familiar with the Visitor and Decorator pattern will immediately recognize the architecture of the system.

The implemented validation techniques are incorporated into an existing, well-tested application, *keystone*, a parser and front-end for the C++ language [12]. The Object Constraint Language, OCL, is used to express the class invariants [13, 14]. The pragmatics involved in using the visitor and decorator approaches are explored and the modularity of these two patterns is exploited to provide flexibility in terms of the frequency and level of granularity of validation of the class invariants.

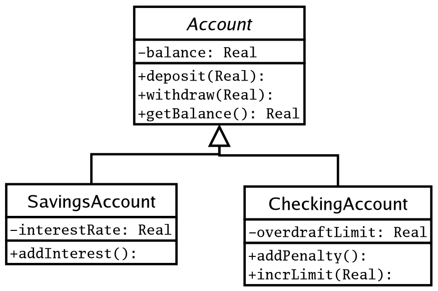

Figure 1. **The Account Hierarchy**. This figure illustrates an inheritance hierarchy with base class Account and derived classes SavingsAccount and CheckingAccount specializing Account.

Quantitative results are provided that measure the impact of these approaches and the additional faults uncovered in the case study.

Section 2 provides background about class invariants, OCL and validating invariants. The Decorator approach to validating class invariants is reviewed in Section 3, and the Visitor approach is reviewed in Section 4. In Section 5 a case study is presented, *keystone*, a parser and front-end for ISO C++ and the incorporation of invariant validation into *keystone* is described. In Section 6 the validator is benchmarked and some interesting insights into the validation of *keystone* are presented, enabling a comparison of the use of the two patterns. In Section 7 related work is reviewed and conclusions are drawn in Section 8.

2. Background and Overview of the Approach

The presented approach to implementing class invariants in C++ exploits two separate design patterns, the Visitor pattern and the Decorator pattern [11, 15]. In subsequent sections, both patterns are applied to invariant validation, enabling objects to be tracked without affecting the object. Moreover, invariant validation can be withdrawn or the level of validation can be modulated, with minimal modification to the application.

In Section 2.1, class invariants are first described and in Section 2.2 the Object Constraint Language, OCL, is described including an example of invariants expressed in OCL. In Section 2.3 the approach to exploiting the two patterns to validate invariants in a C++ application is reviewed.

2.1. 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 [9]. A class invariant is a property of a class instance that must be preserved by all public 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

```
1 context SavingsAccount
2    inv positiveBalance: self.balance > 0
3    inv rateIsPercent: self.interestRate >= 0 and
4                       self.interestRate < 100

5 context CheckingAccount
6    inv underLimit: self.balance >= self.overdraftLimit
7    inv negativeLimit: self.overdraftLimit < 0
```

Figure 2. **Sample class invariants**. This figure illustrates class invariants for the SavingsAccount and CheckingAccount classes, expressed in OCL.

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.

2.2. The Object Constraint Language (OCL)

A Unified Modeling Language (UML) diagram, such as a class diagram, provides a higher level of program abstraction [16] than, say, the code. 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 UML [13, 14], including the attributes and associations of a class. The choice of OCL as the specification language enables the reuse of design artifacts to express invariants on the classes in the system.

OCL 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. There is a plethora of UML tools available, including Rational Rose© and TogetherSoft™[17]; however, there is no UML modeling tool available that incorporates OCL into round-trip engineering for C++.

Figure 1 illustrates an inheritance hierarchy with a base class, Account, and derived classes SavingsAccount and CheckingAccount that specialize Account. This hierarchy is intended to represent a simple bank account with class SavingsAccount used to represent a savings account where interest is paid on balances that are assumed to be positive. The class CheckingAccount represents a checking account with overdraft facilities on which a fixed fee is levied.

To provide a flavor of OCL expressions, Figure 2 illustrates class invariants for the Account hierarchy of Figure 1. The phrase **context** SavingsAccount on line one in Figure 2 provides the class context in which the invariants on lines 2–4 are applied. These invariants insist that

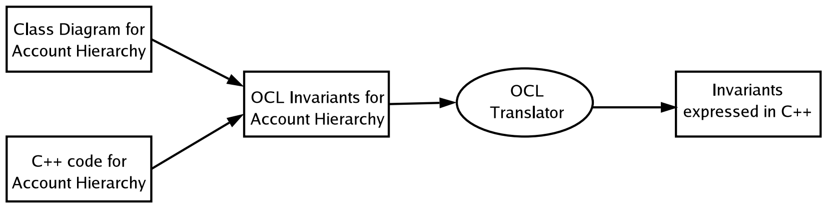
Class Diagram for Account Hierarchy
C++ code for Account Hierarchy
OCL Invariants for Account Hierarchy
OCL Translator
Invariants expressed in C++

Figure 3. **Validation system overview**. This figure illustrates the central module in the visitor-based validator, the OCL Translator, which takes an OCL specification of the invariants for a class hierarchy and translates them into C++.

for any SavingsAccount object, the balance must be positive, and the interest rate must be between 0 and 100. Similarly for the CheckingAccount class, invariants assert that the balance may not go below the overdraft limit, which is expressed as a negative amount. OCL can also be used to express method pre- and post-conditions, but pre- and post-conditions are not considered further in this paper.

2.3. Implementing Class Invariants in C++

Figure 3 provides an overview of the validator, with input to the system shown on the left and output shown on the right of the figure. Inputs to the system include the OCL invariants for the class or class cluster being validated, the Account hierarchy in this case. In the approach, the invariants are written manually, guided by the class diagram for the class or class cluster, shown on the upper left of Figure 3, together with the code for the class or class cluster being validated, shown on the lower left of the figure. The central module is the OCL Translator, shown in the center of the figure. The OCL Translator is a recursive-descent parser for OCL, with semantic actions inserted into the parse to automatically construct the class or classes required to represent the invariants.

The invariants reported in this paper are derived from the C++ code; however, as the target application is extended, it is expected that the invariants may also be part of the design of the system. These two options are highlighted by the input illustrated on the left of Figure 3.

A set of OCL invariants for a class is represented in C++ by a method that asserts each invariant in turn. There are three basic choices for the definition of these methods; they can be implemented (1) as a method in the class itself, (2) as a separate friend method of the class, or (3) as a separate method, with access to attributes only through the public methods of the class.

The first and second choices allow unrestricted access to the instance variables of the class, and thus have a lower overhead than the third choice. However, both also involve modifying the class code. In addition, the second approach breaks encapsulation by using friend methods, which may pose maintenance problems in the future. Since the overall goal is to minimize changes to the original class code, the third option is chosen, and implement class invariants in separate code, using accessor methods to refer to the class attributes.

```
 1 namespace AccountInvariant
 2 {
 3   using namespace Invariant;
 4   void Check( const Account* self ) {
 5     // No invariants to check here
 6   }
 7   void Check( const SavingsAccount* self ) {
 8     Check(dynamic_cast<const Account *>(self)); // Check parent's invariant
 9     if (! acquire_invariant_lock()) return;
10     VALIDATE("positiveBalance", self->getBalance() > 0, self);
11     VALIDATE("rateIsPercent", self->getInterestRate() >= 0
12                          && self->getInterestRate() < 100, self);
13     release_invariant_lock();
14   }
15   void Check( const CheckingAccount* self ) {
16     Check(dynamic_cast<const Account *>(self)); // Check parent's invariant
17     if (! acquire_invariant_lock()) return;
18     VALIDATE("underLimit", self.getBalance() >= self.getOverdraftLimit(), self);
19     VALIDATE("negativeLimit", self.getOverdraftLimit() < 0, self);
20     release_invariant_lock();
21   }
22 }
```

Figure 4. **Class invariants implementation in C++**. This figure shows the generated C++ implementation of the class invariants from Figure 2.

The C++ code generated for the OCL invariants of Figure 2 is shown in Figure 4. Each `Check()` method is responsible for validating a single class invariant, and calls `VALIDATE` to check each condition in turn. The `VALIDATE` operation is defined as a macro that simply calls a validation method, passing the current file and line number that are printed, along with a suitable message, if the invariant is false.

Three conditions are usually imposed on methods called using class invariants. First, such invariants should not change the state of the object during validation. This is enforced in the C++ code by declaring the object as a `const` parameter to the `Check()` method on lines 4, 7 and 15 of Figure 4. Second, any method called during invariant validation should not trigger the validation of *its* class invariants. This is implemented using a simple locking variable, which is acquired at the start of each `Check()` method and released at its end, lines 9, 13, 17 and 20. Third, any instance of a class should also satisfy the invariants of any of its base classes. This is handled by dynamically casting the parameter to its base class(es), and calling the corresponding `Check()` method, lines 8 and 16.

Note that the invariant code in Figure 4 has translated the direct reference to the instance variables into calls to appropriate accessor methods, `getBalance()`, `getInterestRate()` and `getOverdraftLimit()`. The success of this approach depends on the existence of such methods, their definition as `const` methods that do not change the object's state, and the use of a consistent naming scheme for them. If this is not the case, then it may be preferable to declare the `Check()` method as a friend of the class being checked.

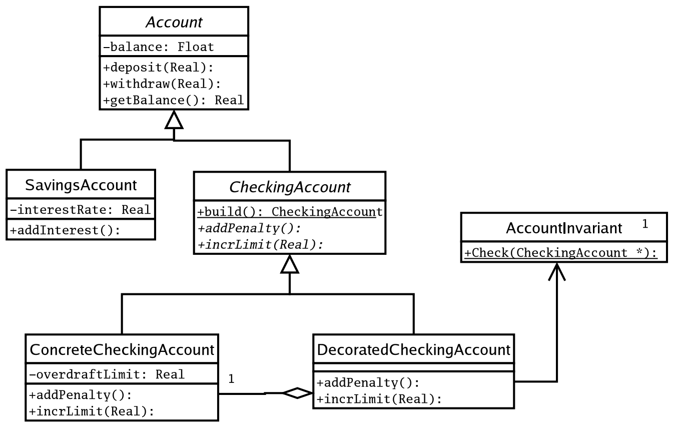

Figure 5. **Decorating the Account Hierarchy**. This figure shows the Account hierarchy from Figure 1, where the class CheckingAccount has been decorated.

3. The Decorator Approach to Validation

In Section 3.1, the Account example from Section 2 is used to describe an implementation of automated invariant validation using the Decorator pattern. In Section 3.2, two difficulties that are encountered during the implementation of the Decorator classes are reviewed.

3.1. The Invariant Decorators

The Design by Contract approach stipulates that class invariants should be true before and after the execution of all public methods of a class. Since the purpose of the Decorator pattern is to augment the functionality of methods in some uniform way, this pattern can be exploited by adding Decorators to check class invariants.

As an example of using Decorators to check class invariants, consider the Account example from Figure 1. In order to add invariant-checking code to, say, the CheckingAccount class, the class hierarchy must be refactored so that it looks like Figure 5. Here, the class ConcreteCheckingAccount implements the old functionality of CheckingAccount from Figure 1, while the new version of CheckingAccount simply maintains the same public interface, with all of its original methods being declared virtual and pure.

The class DecoratedCheckingAccount implements the invariant validation. Each of its methods is defined to call the corresponding method in ConcreteCheckingAccount, and, in addition, verify the class invariants before and after the call. Since both DecoratedCheckingAccount and ConcreteCheckingAccount inherit from CheckingAccount, they may be used interchangeably, and in a manner that is transparent to the rest of the code, which simply refers to CheckingAccount.

The only other change relates to the choice of which derived class to create when a CheckingAccount is to be created. This is implemented by augmenting the class

CheckingAccount with a static `build()` method, a factory method that can construct either an instance of DecoratedCheckingAccount or ConcreteCheckingAccount at the programmer's discretion. Any code that called the original constructor of CheckingAccount needs to be changed to use the factory method instead, but this is a relatively trivial modification.

In summary, to decorate a class UserClass, proceed as follows:

1. Rename UserClass to ConcreteUserClass, and make this a derived class of UserClass.
2. Generate the code for UserClass by making all the methods of the old UserClass pure and virtual. Remove any protected or private variables and methods. Make the constructor protected, and declare a build method that takes the same parameters as the constructor. The default implementation of the build method is to create an instance of ConcreteUserClass.
3. Generate a class DecoratedUserClass, which is a derived class of UserClass, and contains a single instance variable of type ConcreteUserClass. All public methods in DecoratedUserClass are implemented so that they check the class invariant before and after calling the corresponding method in ConcreteUserClass.

3.2. Issues with the Decorator Approach

The goal in using design patterns to implement invariant checking was to minimize the changes to the original class hierarchy, yet it is clear from Figure 5 that this is not entirely achievable using the Decorator approach. While most of the changes are minimal, and can be implemented automatically, the intrusion into the original class hierarchy can cause some difficulties if complex inheritance graphs are involved.

For example, suppose that in Figure 5 the base class Account is to be decorated with its own set of invariant checking. This would necessitate the creation of classes ConcreteAccount and DecoratedAccount as derived classes of Account. The class DecoratedCheckingAccount would then have to be changed so that it inherits from DecoratedAccount as well as CheckingAccount. However, these two classes share a common base class, Account, and care must be taken to ensure that this common base class is inherited *virtually* so as to avoid duplication.

In some cases, however, duplication is unavoidable. Consider the situation where Account itself had some base class, say B. Any instance of DecoratedAccount thus inherits indirectly from B, and contains an effectively useless copy of B's instance variables. The copy of B is useless since the functionality will be supplied when the methods are dispatched to the corresponding method in ConcreteAccount, which also inherits from B.

A possible solution, of course, is to define a version of B with the same interface, but with no instance variables, but the promulgation of this approach back through a deep inheritance hierarchy my be undesirable. The alternative involves allowing the creation of a version of B as part of DecoratedAccount, but care must be taken to ensure that the appropriate constructors are available. In practice, the programmer should proceed with caution when applying the Decorator approach to classes with a high depth of inheritance.

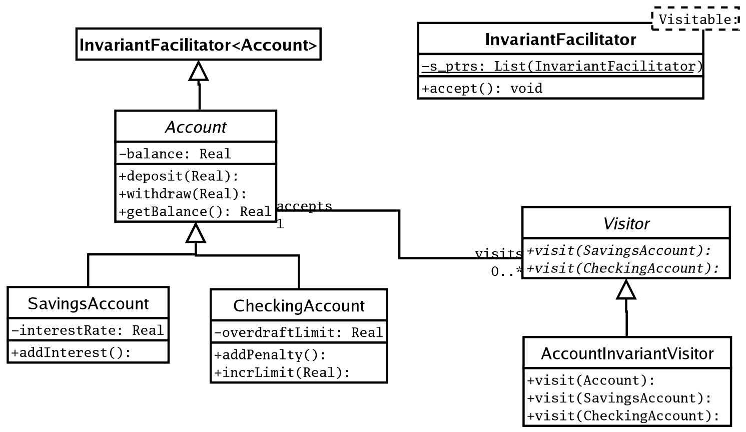

Figure 6. **A Visitor for Account**. This figure illustrates The Visitor pattern, applied to the Account Hierarchy described in Section 2 and illustrated in Figure 1.

4. The Visitor Approach to Validation

In Section 4.1, the Account example from Section 2 is used to illustrate the approach to automated validation of class invariants using the Visitor pattern. Section 4.1 begins with a review of the Visitor pattern, followed in Sections 4.2 and 4.3 with a description of the InvariantFacilitator and InvariantVisitor classes that are automatically generated in the system. In Section 4.4 a summary and an algorithm is provided to facilitate reproduction of the results presented in this paper.

4.1. The Visitor Pattern

The Visitor pattern permits a developer to add functionality to an existing class hierarchy without "polluting" the hierarchy with unrelated operations [11]. Thus, the Visitor pattern usually consists of two inheritance hierarchies, an *element hierarchy*, which is extended by visitation, and a *Visitor hierarchy*. To incorporate visitation into an existing element hierarchy, a single method, usually denoted `accept(Visitor)`, is the only modification required to be performed on the element hierarchy and an `accept` method is added to each class in the hierarchy.

The Visitor hierarchy consists of (1) a base class with pure virtual methods that define the interface for all concrete Visitor instances, and (2) derived classes specialized to enable incorporation of the additional functionality into the element hierarchy. The interface of the Visitor base class contains a `visit` method for each concrete class in the element hierarchy. To use the Visitor pattern, each instance of the element hierarchy calls `accept` with a concrete Visitor as a parameter to `accept`. The `accept` method then invokes the visit function, passing self as a parameter. When the instance of the element hierarchy calls `visit` on the Visitor object, the instance effectively identifies its type to the Visitor, which then computes the added functionality. The invocation of the `accept` and then the `visit` methods forms a double method dispatch that is characteristic of the Visitor pattern [18]. The interested reader may consult reference [11] for more information on this behavioral pattern.

Since the goal is to use the Visitor pattern to automate invariant validation, a variation of the pattern is exploited that is less intrusive of the element hierarchy and obviates one of the characteristic dispatches of the Visitor pattern. The approach consists of automated generation of both the Visitor hierarchy and a new base class for the element hierarchy that obviates the insertion of an `accept` method in each class of the element hierarchy.

To describe the Visitor approach to invariant validation, the simple Account example is used as the element hierarchy, originally illustrated in Figure 1 and now shown on the left side of Figure 6. The right side of the figure illustrates a Visitor hierarchy with a base class, Visitor, and a derived class, AccountInvariantVisitor. In the figure, the element hierarchy is extended by adding a base class for Account, InvariantFacilitator, which contains a list of Account objects and a virtual `accept` method, that uses run time type inferencing to choose the `visit` method to invoke. The use of run time type inferencing can induce maintenance problems if the element hierarchy is extended or modified but, in this case, the InvariantFacilitator is generated automatically, which enables the maintenance of stricter control over the selection process. Moreover, the use of type information obviates the insertion of an `accept` method in each of the Account classes. In the next section, details are provided about the InvariantFacilitator class and in Section 4.3 details about the AccountInvariantVisitor class are provided.

4.2. The InvariantFacilitator

To facilitate automated validation, a version of the Visitor pattern is used that obviates any modification to the element hierarchy, except for adding a single base class called an InvariantFacilitator. The InvariantFacilitator is defined once, templated by the class that is to be visited, and accomplishes several tasks. First, the InvariantFacilitator includes a method to check the invariants for the current object, illustrated as `checkInvariants()` on lines 11–14 of Figure 7. Second, the InvariantFacilitator includes a method, `checkAllClassesInvariants()`, lines 15–21, that iterates through the static vector of objects, declared on line 24 of the figure. Using the `checkAllClassesInvariants()` method, the user may check the invariants for a class hierarchy at various points during program execution. Finally, each instantiation of the InvariantFacilitator template must define a method `accept()`, shown on lines 27–36 of Figure 7. This obviates the explicit inclusion of an `accept` method in each class in the hierarchy under validation.

4.3. The Invariant Visitor

The right side of Figure 6 illustrates a Visitor hierarchy to perform invariant validation of the Account example. The base class, Visitor, defines an interface consisting of pure virtual methods to visit each concrete instance of the Account hierarchy. The derived class,

```
template<class Visitable>
class InvariantFacilitator {
public:
  InvariantFacilitator() {
    m_idx = s_ptrs.size();
    s_ptrs.push_back(this);
  }
  virtual ~AccountInvariantFacilitator() {
    s_ptrs[m_idx] = NULL;
  }
  void checkInvariants() const {
    InvariantVisitor visitor;
    accept(&visitor);
  }
  static void checkAllClassesInvariants() {
    InvariantVisitor v;
    std::vector<InvariantFacilitator *>::const_iterator it;
    for(it = s_ptrs.begin(); it != s_ptrs.end(); ++it ) {
      if( *it ) (*it)->accept(&v);
    }
  }
  virtual void accept( Visitor* ) const;
private:
  static std::vector<InvariantFacilitator*> s_ptrs;
  int m_idx;
};

void InvariantFacilitator<Account>::accept( Visitor* v ) const {
  if( const SavingsAccount* self =
       dynamic_cast<const SavingsAccount*>(this) ) {
    v->visit(self);
  }
  if( const CheckingAccount* self =
       dynamic_cast<const CheckingAccount*>(this) ) {
    v->visit(self);
  }
}
```

Figure 7. **The facilitator class** This figure illustrates a generic InvariantFacilitator, a C++ class definition that includes methods checkInvariants and checkAllClassesInvariants that choreograph invariant validation.

AccountInvariantVisitor, contains `visit` methods that check the invariants for the respective instance of the Account hierarchy.

Figure 8 presents the AccountInvariantVisitor that is generated by the OCL translator for the invariants listed in Figure 2. Since the invariant validation code is contained in the class AccountInvariant, the methods in the Visitor simply need to call the appropriate check method.

```
class AccountInvariantVisitor : public Visitor {
  void visit( const SavingsAccount* self ) {
    AccountInvariant::Check(self);
  }
  void visit( const CheckingAccount* self ) {
    AccountInvariant::Check(self);
  }
};
```

Figure 8. **Using Visitor to incorporate invariants**. This figure illustrates a C++ class definition for an AccountInvariantVisitor class, which encapsulates the invariants for classes SavingsAccount and CheckingAccount.

4.4. Summary of Visitor Approach

In summary, to implement class invariant validation using Visitors for some class UserClass, the following procedure is used:

1. Add `InvariantFacilitator<UserClass>` as an extra base class for UserClass.
2. Define a method with the signature:
   `void InvariantFacilitator<UserClass>::accept(Visitor* v) const`
   This should call the `visit` method of the formal parameter `v`. For a Visitor working on a class hierarchy, it will be necessary to include a series of dynamic casts in this method, as shown in Figure 7.
3. Add a definition of a `visit` method to the class `InvariantVisitor` that takes a UserClass object as its only parameter, and calls the invariant checking code, following the structure of Figure 8.

A phase of invariant validation can then be initiated using a call to:
`InvariantFacilitator<UserClass>::checkAllClassesInvariants();`

5. Case Study of the Use of Invariants in Keystone

Section 5.1 contains a description of the use of invariants in the validation of *keystone*, a research project whose goal is the design and implementation of a parser and front-end for ISO C++ [19]. *keystone* includes an implementation of name lookup: the process of matching each occurrence of a name in a program with the corresponding declaration of that name. The goals of the study are to measure the efficiency and the effectiveness of the two approaches to invariant validation.

Section 5.2 gives a description of the class hierarchies that are validated together with motivation of the importance of automating this validation process. In Sections 5.3 and 5.4,

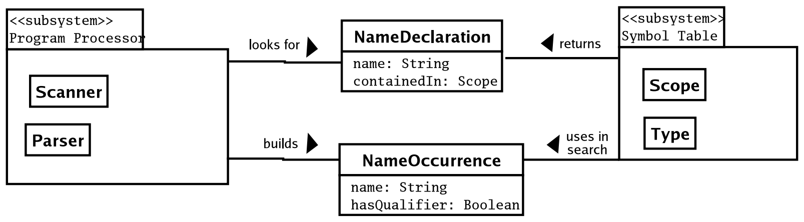

Figure 9. **Keystone summary**. The Program Processor subsystem, illustrated on the left, is responsible for directing symbol table construction and name lookup. The Program Processor marshals information about the name in a NameOccurrence object and directs the search for a corresponding NameDeclaration in the Symbol Table subsystem, illustrated on the right. If the name is not found than a NameDeclaration is installed in the symbol table. The result of name lookup is that a NameDeclaration is returned to the Program Processor and the corresponding semantic information about the name may be used to facilitate parsing.

the symbol table is described and examples of invariants associated with the symbol table for *keystone* are also provided.

5.1. The Keystone Parser Front-end

Figure 9 summarizes the design of the case study application, a parser front-end for ISO C++ [19]. 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 9.

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 9.

The NameOccurrence object encapsulates local information relevant to the lookup, including the String representation of the name and a Boolean to indicate name qualification (by class or namespace). The NameDeclaration object includes the String representation of the name and a pointer to the enclosing scope.

5.2. The Keystone Hierarchies

One of the most important subsystems of *keystone* 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 9 and details of the Scope hierarchy are illustrated in Figure 10.

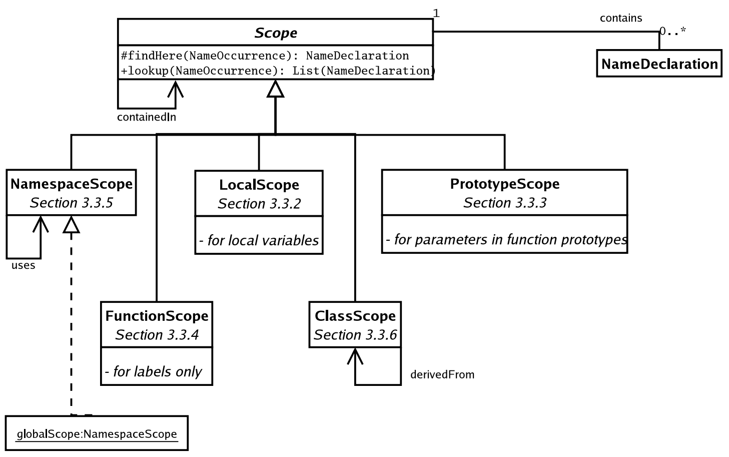

Figure 10. **Class Diagram for the Scopes Hierarchy**. Each Scope object contains a list of NameDeclaration instances, along with name lookup functionality. The Scope class has six derived classes, as detailed in the referenced sections of the ISO C++ standard.

Figure 10 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. The ongoing work includes this 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 C++ code from the OCL specification facilitates repeated validation of the invariants as well as the maintenance of consistency between specification and implementation. The case study describes automated validation of class invariants in the NameDeclaration, Type and Scope hierarchies.

5.3. OCL Invariants for Scope Hierarchy

Figure 11 provides an example of class invariants for the Scope and ClassScope classes in *keystone*. The invariants for ClassScope are the conjunction of its invariants and the invariants of the base class, Scope. The invariant on line 2 of Figure 11 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 that the translator generate a function containing a **for** loop that iterates through the **vector** of NameDeclaration objects local to the class scope.

```
context Scope
  inv: self.getDecl() <> NULL implies
         self.getDecl().getCorrespondingScope() = self
  inv: self.getContainingScope() <> NULL xor
         self.getName() = "_GlobalNamespace"
  inv: self.getLocals()-> forAll( n:NameDeclaration
                               | n.getContainingScope() = self )

context ClassScope
  inv: self.getLocals()-> forAll( n:NameDeclaration
                               | n.getType() <> Type::namespaceType )
  inv:
    self->getContainingScope().oclIsTypeOf(NamespaceScope)
     or self->getContainingScope().oclIsTypeOf(ClassScope)
     or self->getContainingScope().oclIsTypeOf(LocalScope)
```

Figure 11. **Sample keystone invariants**. This figure illustrates class invariants for the Scope and ClassScope classes, expressed in OCL.

5.4. Approaches to Validating Class Invariants in the Keystone Symbol Table

The standard approach for validating class invariants 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. This granularity of validation is readily accomplished with the Decorator approach described in Section 3, where methods are wrapped to enable validation before and after invocation of the actual method.

Most systems implementing invariant validation provide several levels of granularity to permit the developer to adjust the overhead incurred due to validation. The Visitor approach described in Section 4 facilitates less intensive validation checking by permitting the developer to initiate invariant validation at strategic points during the program's execution.

One of the characteristics of the target application of the case study, *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.

Given the stability of objects in the *keystone* symbol table, Visitors provide assurance about invariants through validation at program termination. Nevertheless, an intensive approach, more suitable to scenarios where objects are created and destroyed during program execution, is always available through the Decorator pattern.

Thus, in the section that follows, the various granularities of invariant validation in *keystone* provided by the two approaches are compared. In particular, frequent invariant validation

Table I. **Test suite**. This table lists the programs used as input to the *keystone* parser and front-end. The *Totals* in the Size column reflect the sum of the values for each program; the *Totals* in the Coverage column reflect the cumulative coverage for all programs.

Test case	**Size**		**% Coverage**			
	lines	**classes**	**line**	**branch**	**jump**	**call**
encrypt	853	1	64	48	40	46
ep-matrix	3,698	78	65	48	41	47
fft	1,891	51	64	48	41	47
graphdraw	3,213	199	65	48	41	48
php2cpp	1,644	6	54	39	32	37
vkey125	8,146	279	61	44	37	43
ISOclauses	3,887	364	78	57	49	59
Total	*23,332*	*978*	*80*	*58*	*51*	*61*

using the Decorator pattern is compared with invariant validation near the end of execution using the Visitor pattern.

6. Results

In this section, the results of the study of automated validation of class invariants are described. The target application for the study is *keystone* [12, 20, 21], a parser front-end for the ISO C++ language [19]. The validator was executed on a Dell Dimension 2350 PC with a 2.4 GHz Intel Pentium IV processor equipped with 1 GB of 266MHz DDR RAM, running the Red Hat 9.0 distribution of the GNU/Linux operating system. The implementation languages were C++ [22] and Perl [23], compiled with GNU *gcc* version 3.3 and the Perl interpreter version 5.6.0. In Section 6.1, the test suite for the study is described, and in Section 6.2, the number of objects for which invariants were validated as well as the number of invariants executed are measured. In Section 6.3, the impact on performance due to the invariant checking is described and in 6.4, the improvement in *keystone* through invariant validation is presented. Finally, Section 6.5 describes the impact of validation on the *keystone* application and, in Section 6.6, the threats to the validity of the work are presented.

6.1. The Test Suite

Table I summarizes the suite of seven test cases, listed in the rows of the table as encrypt, ep-matrix, fft, graphdraw, php2cpp, vkey125 and ISOclauses. In this paper, a *test case* is one of the applications listed in Table I, and used as input to the *keystone* parser and front-end. We refer to the collection of seven test cases as a *test suite*, or *suite*.

The test cases in the suite were chosen because of their range and variety of application. Column one of Table I lists each of the test cases, column two lists the number of (non-blank,

Table II. **Number of objects and invariants checked**. The rows of the table list the number of scope, type, and name declaration objects validated for each test case. The final two columns summarize the number of invariants checked for each test case using the Visitor approach (EOP), and Decorator approach (ATT), respectively.

	Objects			**Invariants**	
Test case	**Scope**	**Type**	**Name**	**EOP**	**ATT**
encrypt	637	1,538	1,125	6,011	1,892,316
ep-matrix	5,079	10,133	8,446	42,397	99,226,130
fft	1,110	3,800	2,716	13,144	8,144,287
graphdraw	2,384	7,879	5,505	29,621	11,698,644
php2cpp	575	2,043	1,386	6,588	4,869,845
vkey125	2,156	11,859	9,823	40,382	48,592,825
ISOclauses	3,565	5,329	4,119	24,409	902,924

non-comment) lines of code, and column three lists the number of classes in each test case. Columns four through seven list the coverage each test case achieves when used as input to *keystone*, as collected by the *gcov* utility, included in *gcc*: the percentage of line coverage, the percentage of branch coverage, the percentage of branches taken, and the percentage of function call coverage is illustrated in these four columns. The final row of the table lists the total lines of code and the total number of classes for all seven test cases. In this final row, columns four through seven list the total accumulated coverage for each of the four coverage criteria. For example, the last row shows that 80% of the lines of code in *keystone* are covered when all seven test cases are used as input.

Test case encrypt is an encryption program that uses the Vignere algorithm [24] and the ep matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable C++ number theory library [25]. php2cpp converts the PHP web publishing language to C++ [26]. The test case fft performs fast Fourier transforms [27] and graphdraw is a drawing application that uses *IV Tools* [28], a suite of free XWindows drawing editors for Postscript, TeX and web graphics production. vkey125 is a GUI application that uses the *V GUI* library [29], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications. ISOclauses is a suite of 428 C++ programs gleaned from the ISO C++ standard, and previously used to measure the conformance of C++ compilers [30].

Apart from the small, intricate examples in ISOclauses, all of the test cases are complete applications, except for three that use large libraries: ep matrix, vkey125 and graphdraw use the *NTL*, *V GUI* and *IV Tools* libraries respectively. *keystone* had previously been tested using this same test suite and was considered valid with respect to the test suite.

6.2. Number of Objects and Invariants Checked

Table II summarizes the number of objects and invariants that are checked during *keystone*'s processing of the programs from the test suite. The rows of the table list the test cases and

Test case	NONE	EOP	ATT
encrypt	0.48	0.48	1.04
ep matrix	32.36	32.43	79.87
fft	2.05	2.05	5.00
graphdraw	5.12	5.15	9.47
php2cpp	0.88	0.88	2.57
vkey125	14.71	14.74	34.10

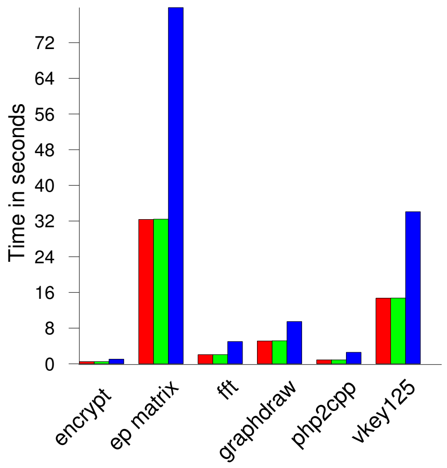

Figure 12. **Efficiency**. The table in this figure lists execution times for the approaches to validation: *no validation*, NONE, the *End of Program* approach, EOP and the *All The Time* approach, ATT. The graph summarizes the results for each test case.

the columns list the data acquired by monitoring the objects involved in invariant validation. Class invariants were attached to classes in three of *keystone*'s class hierarchies, Scope, Type and NameDeclaration. The columns labeled **Objects** in Table II list the number of objects created in each of these hierarchies.

The default configuration for the Visitor-based approach is to validate invariants once for all objects, at the *End of the Program*, or **EOP**. The default configuration for the Decorator-based approach is to validate invariants for each object at the end of constructor execution, at the beginning of destructor execution and at the beginning and at the end of all public methods, which are referred to as *All The Time*, or **ATT**. The rightmost two columns of Table II, labeled **Invariants**, report the number of times an invariant was validated for all three hierarchies for the default Visitor and Decorator approaches.

For example, the graphdraw program, listed in the fourth data row of Table II, required 29,621 invariants to be validated for the Visitor approach and 11,698,644 for the Decorator approach. Clearly, the Decorator approach, which validates invariants at a finer level of granularity than the Visitor, checks the invariants considerably more often.

6.3. Efficiency

In this section, the overhead of invariant validation is evaluated in terms of the time taken to run seven of the test cases. ISOclauses are excluded from the timings since it consists of a series of very small programs with negligible individual execution time.

Figure 12 contains a table listing execution times for each of three approaches as well as a graph summarizing these results. The first column of the table lists the test cases, the second column shows the running time in seconds when no invariants are validated, shown as **NONE**. The third column lists results using Visitors at EOP, and the final column lists execution times for ATT validation using Decorators. For example, consider the fourth data row of the table, showing results for the graphdraw test case, where 5.12 seconds were required to run this test case with no invariant validation, 5.15 seconds were required to validate invariants at EOP, and 9.47 seconds were required to validate invariants ATT.

The graph in Figure 12 summarizes the results for each test case. There are three sets of bars for each test case. The first bar of each set summarizes results for NONE, the second bar for EOP and the third bar for ATT.

The results in Figure 12 show that validating invariants ATT generally imposes a considerable overhead on the program's running time. Conversely, validation using Visitors at EOP imposed little or no overhead in most cases. These timings are consistent with the results reported in Table II where the number of invariants checked for the Decorator approach, ATT, was consistently greater than the number of invariants checked for the Visitor approach, EOP.

The goals in presenting Table II and Figure 12 are to provide some results about the number of invariants that must be checked and the efficiency for each of the granularities of validation. Checking invariants *all the time*, ATT, is likely to incur greater overhead in number of invariants that must be checked and efficiency, as compared to checking invariants at the *end of program*, EOP. However, it is difficult to directly correlate the number of invariants checked with the cost to check the invariants since the time to check each invariant is not the same. For example, some invariants can be checked through examination of a local data attribute; however, other invariants must be checked by calling other functions, such as getCorrespondingScope, whose time behavior depends on the structure of the symbol table at the time of the search. Nevertheless, the study of *keystone* illustrates the relative cost impact of validation for the various granularities of validation.

6.4. Modulating the Level of Invariant Validation

One of the significant advantages to the approach to invariant validation in C++ programs is the ability to vary the level of invariant checking at run-time, and at a fine-grained level. This is in contrast to preprocessor-based approaches, which typically only allow all invariants to be switched on or off at compile-time. To demonstrate the additional flexibility provided by the approach, experiments with visiting and decorating a proportion of the objects in *keystone* are conducted, which allows us to modulate the trade-off between validation coverage and overhead.

The default operation of the Decorator based approach is to decorate all relevant objects as they are created. However, it is a feature of the Decorator pattern that objects may be selectively decorated, and that decoration may be added or withdrawn at run-time. As an example of this approach, *keystone* can be modified so that only a proportion of the objects are decorated; this modification involved a trivial change to the factory method for each class.

Program	ATT	2	4	8	16	32	64	128	256	512	1024
encrypt	1.04	0.73	0.61	0.53	0.50	0.50	0.50	0.51	0.50	0.49	0.49
ep-matrix	79.87	55.85	45.51	39.24	36.71	35.79	35.19	34.73	34.73	34.65	34.41
fft	5.00	3.56	2.79	2.46	2.27	2.21	2.18	2.13	2.11	2.10	2.09
graphdraw	9.47	7.33	6.33	5.83	5.50	5.42	5.33	5.29	5.24	5.18	5.21
php2cpp	2.57	1.74	1.32	1.12	1.04	0.99	0.94	0.95	0.93	0.93	0.92
vkey125	34.10	24.18	19.25	17.23	15.86	15.50	15.27	15.10	14.93	14.93	15.04

Figure 13. **Decreasing the decoration Ratio,** R. The table in this figure lists execution times for the decorator pattern, where every R^{th} object is decorated. A ratio of $R = 1$ corresponds to ATT in Figure 12.

Program	EOP	32768	16384	8192	4096	2048	1024	512	256	128	64
encrypt	0.48	0.48	0.46	0.48	0.46	0.47	0.48	0.49	0.50	0.53	0.59
ep-matrix	32.43	32.30	32.43	32.44	32.62	32.58	32.86	33.06	33.78	35.21	38.52
fft	2.05	2.05	2.04	2.06	2.05	2.06	2.10	2.12	2.19	2.31	2.60
graphdraw	5.15	5.15	5.12	5.15	5.16	5.19	5.28	5.41	5.70	6.29	7.46
php2cpp	0.88	0.88	0.88	0.90	0.89	0.89	0.90	0.90	0.92	0.95	1.04
vkey125	14.74	14.78	14.76	14.75	14.82	14.87	15.08	15.36	16.06	17.36	19.69

Figure 14. **Increasing the visitation interval,** I. The table in this figure lists execution times for the visitor pattern, where visitation occurs after every I^{th} object is created.

Table 13 presents results of decreasing the *decoration ratio* R, where $1 \leq R \leq 1024$. For example, a decoration ratio of 1024 means that one object is decorated out of every 1024 objects created. When $R = 1$ all objects are decorated, and this corresponds to **ATT** in Figure 12; similarly, when $R = 0$ no objects are decorated, and this corresponds to the **NONE** column of Figure 12. As expected, Table 13 depicts a gradual decrease in overhead as the decoration ratio is increased, lowering the level of coverage of the objects being tested. Other modulations of the level of decoration are possible. For example, it would be possible to add decoration to a set of objects during a critical section of the program, and to withdraw this decoration at its end.

The Visitor-based approach validates all objects in a class hierarchy in a single sweep. In previous subsections data is presented for validating invariants at the *End of the Program*, but the Visitors can be called at any stage during a program's run. To demonstrate this, Table 14 presents timings for invariant validation using Visitors, where the *visitation interval* I is varied, where $32768 \geq I \geq 64$. For example, a visitation interval of 1024 means that after each 1024 objects are created, invariant Visitors are called for all live objects at that point in the program. As can be seen from Table 14, as I decreases, the level of validation increases, and the time taken also increases. Overall the increase is slight, since the use of Visitors for validation is infrequent, imposing relatively little overhead.

The approach measured in Table 14 is only intended as an example of modulating the level of visitation. The EOP approach suited *keystone* since all relevant objects were still alive at the end of the program. However, for different applications, Visitors could be called at the end of a particular phase in a program, or could be called with increased frequency during critical sections in a program.

There are many possibilities for modulating the level of invariant validation using either the Visitor or Decorator based approaches. The level of granularity possible reflects the nature of the design patterns used. The Visitor pattern can be applied at the end of a program or before or after phases in a program. These are all *behavioral* criteria, reflecting the fact that the Visitor is a behavioral pattern. The Decorator pattern can be varied at the object level or indeed at the class level, reflecting the fact that the Decorator is a *structural* pattern.

6.5. Impact of Validation on Keystone

The goals in automating invariant validation are to measure the cost of validation, to establish a level of confidence in the software and to determine if the code is consistent with the design documents. Prior to the use of automated validation in *keystone*, the behavior was manually checked after parsing each of the testsuite's source files. However, the manual check of class invariants was found to be time consuming and error prone. By automating the invariant validation in *keystone*, 11 previously unknown errors were uncovered, 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 [9]. This rule raises the possibility of inconsistency between the invariant of a class and its parent. No inconsistencies were found in combining the invariants in the *keystone* inheritance hierarchies.

An interesting issue encountered during invariant validation is that of invariants that are *temporarily* invalid. To illustrate this kind of invariant, consider a name declaration that is instantiated to facilitate name lookup, but the declaration of the name has not yet been encountered. For example, a class data attribute may be initialized for a class before the declaration of the attribute is encountered. This type of invariant is clearly not the normal class invariant even though these invariants do ultimately become valid before the end of the program. This issue encouraged us to investigate the notion of *temporal invariants*, or invariants that are *eventually* valid [31].

6.6. Threats to Validity

The study, similar to any study that compares various approaches to validation, has limitations that impact the validity and generalizability of the findings. In this section some of these limitations are identified, discuss their impact and the attempts to address the limitations.

A drawback in using case studies as an empirical approach is that, in general, they face stronger threats to internal validity than formal experiments. An important internal threat to the findings is that the results are based on a single case study, *keystone*, a parser and front-end for ISO C++. Nevertheless, *keystone* is a real, non-trivial software system that has provided the basis for the construction of other applications [32, 33, 34, 35]. Moreover, the test cases used for the study are also real programs representing various kinds of applications and libraries. In the study, the symbol table subsystem of *keystone*, consisting of a cluster of classes that enables maintenance and lookup of the names in a C++ program, is validated. The symbol table subsystem implements the most critical aspect of token-decorated parsing, a technique used in *keystone* to facilitate disambiguation of difficult-to-parse language constructs [12].

To facilitate analysis of the program under parse, the *keystone* parser maintains a symbol table containing the names declared and used in the program; these names are maintained until program termination. Thus, once created, most objects in *keystone* remain until termination of the parse. An important aspect of the approaches presented in this paper is the ability to vary the level of granularity of invariant checking performed during program execution. In the visitor approach, invariants are validated at points in the program and the granularity of validation is varied from validating at the end of the program, EOP, to validating at smaller program intervals. For a case study such as *keystone* where objects remain until program termination, it is unlikely that a particular type of object will remain unvalidated. However, for the study of a program where objects are created and destroyed at frequent intervals in the program, it is possible that a particular type of object may be created and destroyed at an interval in the program where validation does not occur; thus, it is possible that in the Visitor approach, one or more type of object may remain unvalidated.

To address this internal threat, the Decorator approach is provided, where the granularity of decoration varies from decorating all of the objects to decorating a ratio of the objects, where even for a small ratio of objects, greater than one, invariants are necessarily validated for each type of object in the program. Thus, given sufficient coverage of the program by the suite of test cases, the class invariants for each type of object in the program will be validated using the Decorator approach.

There are several other threats to the internal validity of the approaches. For example, both the Visitor and the Decorator approaches involve addition of code to an existing application so that the resulting executable is larger than an executable built without validation code. Since the size increase is small, this threat is unlikely to have an impact on the findings. Nevertheless, further study is required where conditional compilation of invariant validation is used to enable measurement of the independent variables with and without validation code.

Another internal threat concerns access to data attributes. In the approach to validation, invariants are maintained outside of the class being validated so that the system must rely on class methods to access private data attributes and these methods must use a consistent naming convention. If an accessor method is not provided or if the nomenclature deviates from the convention, the automation of invariant validation will require either a friend declaration or a parse of the program.

Threats to construction validity of the study concern the appropriateness of the measures for capturing independent variables. The measures of time and the numbers of invariants validated capture some aspects of these variables. However, as indicated in Section 6.3, the time to validate each invariant varies with the invariant and is dependent on the amount and level of data that must be accessed. Thus, time to validate cannot be directly correlated with the number of invariants validated. Nevertheless, the results in the aggregate help to establish the fact that validating invariants all of the time, ATT, is more expensive than validating invariants at the end of the program, EOP. Moreover, the results also support the finding that decreasing the decoration ratio or decreasing the visitation interval increases the expense of invariant validation.

Finally, an external threat to the validity of the approach involves the use of OCL, which may lack the expressivity required for complex queries [36]. OCL was chosen because of its accessibility to the practitioner and its inclusion in the UML. Moreover, no problems were encountered in the formulation of invariants for *keystone* using OCL. However, to address the problem of expressivity, a developer might consider a more expressive formal language.

7. Related Work

In this section, an overview of the work related to invariant validation and support for Design by Contract in C++ is presented. Much of the work on assertion validation for functions in C can be extended to functions in C++. However, straightforward extensions, such as that employed by *GNU Nana* [37], do not include class invariants, and are thus not directly comparable to the work.

Support for Design by Contract in C++ typically falls into one of three categories: (1) use the language constructs to provide support, (2) use macros and the standard C++ pre-processor, and (3) extend the syntax of the language and implement the extensions through a specialized pre-processor. Both of the categories (2) and (3) lack the orthogonality of category (1), and make 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. Category (3) has the disadvantage of being a non-standard addition to C++, whereas the trend, at least since the publication of the ISO standard, has been toward convergence between C++ dialects. Further, neither of the categories (2) or (3) can interact well with a source-level debugger, since the C++ code being executed differs from the code written by the programmer. This work falls into the first category and the papers that are reviewed in this section describe approaches similar to the work presented in this paper.

Reference [38] presents a Percolation pattern, and demonstrates its application to invariant validation. This pattern prescribes the structures that a programmer must implement to ensure that invariants are inherited correctly. However, it does not provide any independent support for invariants, nor does it provide any direct assistance in automating the process. Indeed the use of the Percolation pattern is entirely orthogonal to both the Visitor and Decorator based approaches, and may be used in either the Visitor or Decorator class hierarchies to support inheritance of invariants.

Reference [39, 40] 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. By contrast, while both the Visitor and Decorator based approaches involve adding classes to a program, in most cases neither approach involves substantial modification to the user's own code. In addition, the modular approach ensures that the invariant-checking code may be extracted from the code base for production-quality releases.

Reference [41] presents an approach to emulating Design by Contract in C++ that uses VDM-SL [42] as the specification language. A library of Standard Template Library (STL) "lookalikes" is constructed, and users of the system publicly derive their STL containers from the "lookalikes". To validate invariants, the user writes a specialization of a template function called *inv* containing the invariants. The technique is not automated, nor is it clear how it may be readily deployed outside the containers for which it is defined.

In previous work, preliminary versions of both the Visitor and Decorator based approaches [43, 44] were presented. The present paper extends the work in several ways. First, the use of both patterns is further developed, and smoothed out some technical details. Second, both patterns are presented in a unified framework to facilitate their comparison. Third, present results are compared to both approaches in terms of their impact on the programs being validated.

8. Concluding Remarks

In this paper, the use of two design patterns to automate validation of class invariants in C++ applications is described. An overview of the Visitor and Decorator patterns is also presented together with a detailed description of their use in validation. Using each of the patterns, a case study is presented of invariant validation in *keystone*, a parser and front-end for C++, Quantitative results are presented that measure the impact of these approaches on the case study.

Each approach to invariant validation is automated with the relevant Visitor or Decorator classes being generated directly from the OCL specification of the class invariants. All of the implementation uses standard ISO C++ and requires no language extension. Moreover, the approach is modular in terms of the generated code and can be added to or removed from an existing project as needed by the developer. The pragmatics involved in using the Visitor and Decorator approaches are explored and the results indicate that the use of these two patterns provides flexibility in terms of the level of granularity of invariant validation.

ACKNOWLEDGEMENTS

The authors would like to acknowlege the contribution of Tanton H. Gibbs and Edward B. Duffy to sections of the *keystone* codebase.

REFERENCES

1. L. J. Osterweil and et al. Strategic directions in software quality. *ACM Computing Surveys*, 4:738–750, December 1996.
2. NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report, May 2002.

3. H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *International Symposium on Empirical Software Engineering*, pages 60–70, Redondo Beach, CA, USA, 2004.
4. C.A.R. Hoare. The emperor's old clothes (1980 Turing Award Lecture). *Communications of the ACM*, 24(2):75–83, February 1981.
5. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, February 1969.
6. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
7. R.F. Floyd. Assigning meanings to programs. *Proceedings of American Mathematical Society Symposium on Applied Mathematics*, 19:19–31, 1967.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
9. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, second edition, 1997.
10. D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. B.A. Malloy, J.F. Power, and T.H. Gibbs. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33:19–39, January 2003.
13. 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.
14. J. Warmer and A. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison-Wesley, first edition, 1999.
15. J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.
16. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.
17. TogetherSoft. Together Control Center 5.5. *http://www.togethersoft.com/*, November 2001.
18. S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, first edition, 1996.
19. ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.
20. 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*, pages 792–796, Como, Italy, March 2000.
21. J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *Technology of Object-Oriented Languages and Systems*, pages 57–68, Sydney, Australia, November 2001.
22. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
23. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl: Third Edition*. O'Reilly & Associates, third edition, 2000.
24. S. Alexander. The C++ resources network. http://www.cplusplus.com, October 2001.
25. V. Shoup. Number theory library. http://www.shoup.net/ntl/, March 2002.
26. F. J. Cavalier. Debugging PHP using a C++ compiler. *Dr. Dobbs Journal*, pages 42–46, March 2002.
27. O. Kiselyov. Fast Fourier transform. *Free C/C++ Sources for Numerical Computation*, March 2002. http://cliodhna.cop.uop.edu/~hetrick/c-sources.html.
28. J. M. Vlissides and M. A. Linton. IV tools. http://www.vectaport.com/ivtools/, March 2002.
29. B. Wampler. The V C++ GUI framework. http://www.objectcentral.com, October 2001.
30. B.A. Malloy, J. F. Power, and T.H. Gibbs. C++ compilers and ISO conformance. *Dr. Dobb's Journal*, 28(11):54–60, November 2003.
31. T.H. Gibbs and B.A. Malloy. Weaving aspects into C++ applications for validation of temporal invariants. In *7th European Conference on Software Maintenance and Reengineering*, pages 249–258, Benevento, Italy, March 2003.
32. P. Clarke and B. A. Malloy. A unified approach to implementation-based testing of classes. In *1st Annual International Conference on Computer and Information Science*, pages 226–234, Orlando, Florida, USA, October 3-5 2001.
33. P. Clarke, B. A. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in oo software. In *7th European Conference on Software Maintenance and Reengineering*, pages 213–222, Benevento, Italy, March 2003.
34. B. A. Malloy, P. J. Clarke, and E. L . Lloyd. A parameterized cost model to order classes for integration testing of C++ applications. In *14th International Symposium on Reliability Engineering*, pages 353–364,

Los Alamitos, CA, November 2003. IEEE Computer Society Press.

35. S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan. Reveal: A tool to reverse engineer class diagrams. In *40th International Conference on The Technology of Object-Oriented Languages and Systems*, pages 13–21, Sydney, Australia, February 2002.
36. M. Vaziri and D. Jackson. Some shortcomings of OCL, the object constraint language of UML. Technical report, MIT, December 1999.
37. P.J. Maker. GNU Nana: Improved support for assertions and logging in C and C++. http://www.gnu.org/manual/nana, February 27 1998. (version 1.14).
38. R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
39. P. Guerreiro. Another mediocre assertion mechanism for C++. In *Technology of Object-Oriented Languages and Systems*, pages 226–237, St. Malo, France, June 2000.
40. 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.
41. 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.
42. C. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.
43. E.B. Duffy, J.P. Gibson, and B.A. Malloy. Applying the decorator pattern for profiling object-oriented software. In *International Workshop on Program Comprehension*, pages 84–93, Portland Oregon, May 10–11 2003.
44. T.H. Gibbs, B.A. Malloy, and J.F. Power. Automated validation of class invariants in C++ applications. In *International Conference on Automated Software Engineering*, pages 205–214, Edinburgh, UK, September 2002.