Weaving Aspects into C++ Applications for Validation of Temporal Invariants

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

Abstract

In this paper, we describe temporal invariants, which are class invariants that are qualified by the operators eventually, always, never, or already. Temporal invariants can capture assertions that may not be valid initially but, as the program continues, must eventually become valid. Moreover, temporal invariants can indicate references to memory that should eventually be deallocated. To facilitate incorporation of temporal invariants as a maintenance or reengineering activity, we weave invariants into the system as aspects. In our case study of a C++ system, the aspects are woven into join points using policies. We investigate the effectiveness of temporal invariants and we compare the performance of our aspect-oriented implementation with several other approaches.

1 Introduction

A class invariant is a property that applies to all instances of the class [22]. The majority of properties that are established during verification of programs are either invariants or depend crucially on invariants [29]. Invariants have been exploited to improve system robustness [23, 26] and to help define strategies for building reliable components [14]. Additionally, invariants have been used to improve testability [26] or to complement testing to expose additional faults [10]. Recently, class invariants have been exploited to improve the *diagnosis scope*, or the distance between a faulty statement and the observation of the fault in the flow of control in a program [5]. During maintenance, invariants have proven invaluable for disclosing locations in the program where the maintainer may have introduced a fault [10].

The problem with class invariants is that some important assertions about a class are initially invalid, due to circular dependencies or the monotonically increasing nature of the system under construction [8]. For example, during compilation of a program, a name object is constructed to facilitate name lookup. However, certain fields in the name object, such as the corresponding scope of the name, may not be known until after name lookup has occurred [24]. Moreover, assertions about a class may have an important impact on memory management. For example, some heap-based objects of a class must be eventually deleted or system performance will degrade, while others need not be deleted. Thus, assertions about class attributes may not be valid initially, but may have an important impact on the system under construction.

In this paper, we describe *temporal invariants*, which are assertions about a class that are qualified by the operators *eventually*, *always*, *never*, or *already*[1]. Temporal invariants can capture assertions that may not be valid initially but, as the program continues, must eventually become valid. In addition, temporal invariants can indicate references to memory that should eventually be deallocated. To facilitate incorporation of temporal invariants as a

[1] In this paper, we discuss two kinds of invariants: *class invariants* and *temporal invariants*. *Class invariants* are traditional invariants about class attributes; *temporal invariants* are described in this paper. When the meaning is clear, we simply refer to invariants.

Published in Conference on Software Maintenance & Reverse Engineering (CSMR'03), Benevento Italy, March 2003

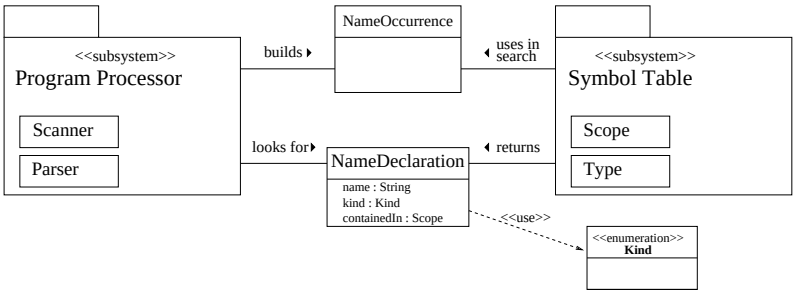


Figure 1. Keystone summary. The Program Processor **subsystem, illustrated on the left, marshals information about a name in a** NameOccurrence **object and directs the search for a corresponding** NameDeclaration **in the** Symbol Table **subsystem, illustrated on the right. The** Symbol Table **is composed of class inheritance hierarchies for** Scope **and** Type**. We use temporal invariants to validate the** Scope**,** Type **and** NameDeclaration **class hierarchies.**

maintenance or reengineering activity, we weave assertions into the system as aspects [16].

We present a case study of a C++ system where the aspects are woven into join points using policies [2]. Our case study includes an investigation of the effectiveness of temporal invariants and we compare the performance of our aspect-oriented implementation with validation of class invariants at the end of the program [10], as well as validation of class invariants at the end of constructors, at the beginning of destructors and at the beginning and end of methods [8]. Our temporal invariants are expressed in the Object Constraint Language (OCL) [3], extended to accommodate temporal considerations [25].

In the next section, we review terminology and background that relates to our work and in Section 3 we describe temporal invariants including some examples of their use. In Section 4 we describe our model of temporal invariants and our technique for weaving temporal invariants as aspects into the case study application. In Section 5 we provide some results of our study and in Section 6 we review some of the work that relates to temporal invariants. We draw conclusions in Section 7.

2 Background

In this section we provide background about terminology and some of the systems and concepts that we use in our paper. In the next section we review aspect-oriented programming and in Section 2.2 we describe *keystone* [19, 24], the application in our case study. In Section 2.3 we review an approach to validating class invariants at the end of program execution [10].

2.1 Aspect-Oriented programming

Aspect-oriented programming, AOP, extends the object-oriented paradigm by enabling more maintainable code using units of modularity called *aspects*. Aspects encapsulate elements such as performance optimization, error checking, exception handling, logging and debugging, which cut across traditional class or module boundaries. AOP uses an aspect weaver to process the development and aspect languages, composing the aspects at join points to produce the desired total system operation [16].

2.2 The keystone parser front-end

Figure 1 summarizes the design of our case study application, *keystone* [19, 24], a parser and front-end for ISO C++ [13]. 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 1.

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 1. The Symbol Table subsystem, the target of our validation effort, is discussed further in Section 3.

2.3 Validating invariants at end of program

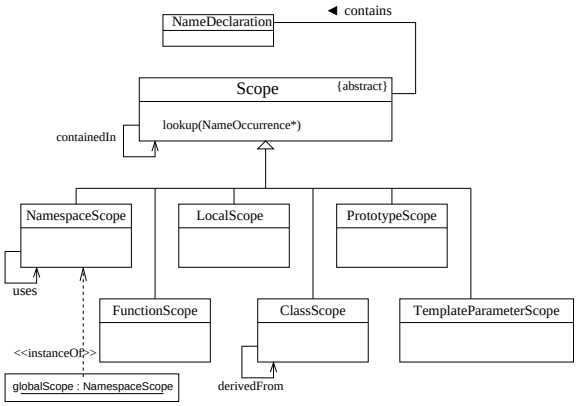
Reference [10] describes a non-invasive approach for validation of class invariants in C++ applications, where the invariants are specified using the Object Constraint Language (OCL). The approach is fully automated so that the user need only supply the class invariants for each class hierarchy to be checked and a 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 at the end of program execution. However, the technique in reference [10] does not easily extend to validating invariants at arbitrary points in a program.

3 Temporal Invariants

In this section, we describe temporal invariants and compare them to class invariants. We then present some examples of temporal invariants expressed in the Object Constraint Language (OCL) [3], extended to accommodate temporal considerations [25].

3.1 Terminology

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 [23]. Temporal invariants are class invariants that are qualified by *always*, *eventually*, *never* and *already*.




```
(1)  context Scope
(2)  inv: always(
(3)    self.getContainingScope() <> NULL xor
(4)    self.getName() = '_GlobalNamespace')
(5)  inv: always(
(6)    self.getLocals()→ forAll (
(7)      n:NameDeclaration |
(8)      n.getContainingScope() = self ))

(9)  context TemplateParameterScope
(10) inv: eventually(
(11)   self.getLocals()→ forAll ( n:NameDeclaration
(12)     | n.getType().oclIsKindOf(namespaceType)
(13)     = NULL))
```

Figure 2. Temporal invariants. In this figure, we illustrate some temporal invariants for the Scope & TemplateParameterScope classes.

An *always valid* invariant is an assertion about a class that must be valid at the end of a constructor, at the beginning and end of all method invocations and at the beginning of a destructor. An always valid invariant is similar to a traditional class invariant except that always valid invariants are also checked at program termination. An *eventually valid* invariant is an assertion about a class that must become valid before an instance of the class reaches the end of a destructor or before program termination. A *never valid* invariant is equivalent to an always valid invariant whose assertion is negated. Finally, an *already valid* invariant is an assertion about a class that must be valid at the beginning of a constructor. An already valid invariant might be used, for example, to describe a file that must have already been opened before an instance of the class is created.

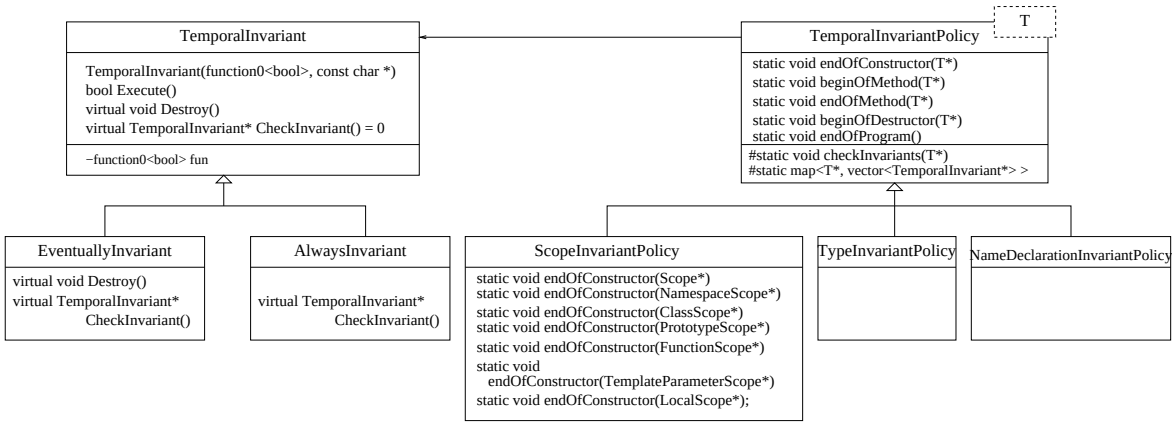


Figure 3. Class diagram for temporal invariants.

In this paper, we use two kinds of temporal invariants: *eventually valid* and *always valid*.

3.2 Examples of temporal invariants

To formulate temporal invariants, we use the Object Constraint Language, OCL, extended to accommodate the temporal qualifications that we require. The target application for our study is *keystone* [19, 24], a parser front-end for the ISO C++ language [13]. Figure 2 illustrates the classes in the Scope hierarchy used in the construction of the *keystone* symbol table, including the base class Scope and derived classes NamespaceScope, FunctionScope, LocalScope, ClassScope, PrototypeScope and TemplateParameterScope. At the bottom of Figure 2 are some temporal invariants for the Scope hierarchy. The class at the top of Figure 2 is NameDeclaration, used to store information about each scope declaration in a program.

Figure 2 illustrates some *eventually valid* and *always valid* temporal invariants to facilitate validation of instances of Scope and TemplateParameterScope. The temporal invariants for TemplateParameterScope are the conjunction of its temporal invariants and the temporal invariants of its parent, Scope. The *always valid* temporal invariant on line 2 of Figure 2 states that either the current scope is contained in another scope or the current scope is the global namespace. The *always valid* temporal invariant on line 5 states that the containing scope for all of the local name declarations must be the same as the current scope. Finally, the *eventually valid* temporal invariant on line 10 states that no C++ template parameter scope can contain a namespace; thus none of the NameDeclaration objects in the current template parameter scope may be namespaces.

4 Using Policies to Weave Aspects

In this section we describe our technique for weaving temporal invariants into an application using aspects [16] implemented as policy classes [2]. We first present the temporal invariant model for *keystone*, our target application, and in Section 4.2, we describe the policy class, join points, and reengineering mechanism used to weave temporal invariants into an application.

4.1 The temporal invariant model

Figure 3 illustrates our model of a temporal invariant system, composed of two inheritance hierarchies. The hierarchy on the left of the figure represents temporal invariants and the hierarchy on the right represents policies for temporal invariants. All classes in the system can be reused for other applications except for the classes derived from TemporalInvariantPolicy, which are specialized to validate the Scope, Type and NameDeclaration hierarchies in our target application, *keystone*.

The TemporalInvariant base class on the left of Figure 3 is a general temporal invariant with four member functions and a data attribute. The TemporalInvariant constructor accepts and stores a generic

```
(1)  AlwaysInvariant*      AlwaysInvariant::CheckInvariant(){
(2)      MAKE_ASSERTION( Execute(), GetErrorMsg() );
(3)      return this;
(4)  }
(5)  EventuallyInvariant*
     EventuallyInvariant::CheckInvariant() {
(6)      if( Execute() ){
(7)          delete this;
(8)          return NULL;
(9)      }
(10)      return this;
(11) }
(12) void EventuallyInvariant::Destroy() {
(13)     MAKE_ASSERTION( false, GetErrorMsg() );
(14) }
```

Figure 4. Two functions for checking temporal invariants.

function object that executes an invariant, and a string for an error message if the invariant is not satisfied. We use the boost::function library [4] to implement the function object, but any generic function object can be used. The function object will be called by Execute, which returns true if the invariant is satisfied and false otherwise. The two virtual functions, Destroy and CheckInvariant, implement the semantics of Temporal Invariants. Class TemporalInvariant provides a default implementation of Destroy, which does nothing. The CheckInvariant function is purely virtual ensuring that an implementation specialized for the particular kind of temporal invariant will be provided by each derived class.

The derived classes EventuallyInvariant and AlwaysInvariant capture the *eventually valid* and *always valid* temporal properties. The AlwaysInvariant::CheckInvariant function, lines 1–4 of Figure 4, uses a macro, MAKE_ASSERTION, to validate its invariant. If the invariant is satisfied, the macro does nothing and the return statement on line 3 allows the invariant to be checked again at the next invocation. However, if the invariant is not satisfied, MAKE_ASSERTION throws an exception alerting the user that a temporal invariant has been broken. The EventuallyInvariant::CheckInvariant function also uses the base class Execute function to ensure that its invariant is satisfied.

However, unlike AlwaysInvariant, EventuallyInvariant only returns its invoking object if the invariant is not satisfied. If the invariant is satisfied, then lines 7–8 deallocate the current invariant and return a NULL pointer. Thus, once an EventuallyInvariant becomes valid it is no longer evaluated. The EventuallyInvariant::Destroy function on lines 12-14 of Figure 4 should be invoked for any EventuallyInvariant still alive at the end of the program. If the instance of EventuallyInvariant is still in existence, its invariant must not have been satisfied and the user should be alerted. Thus, the sole purpose of the Destroy function is to throw an exception.

The right side of Figure 3 illustrates policy classes with three *keystone* specific specializations. The generic TemporalInvariantPolicy class is responsible for storing and checking the Temporal Invariants for each of the objects being validated. A static map associates an object with its temporal invariants. Each public function of the policy class represents a join point where an object's invariants should be checked. The parameter to the function is an object that should have its invariants checked. The endOfProgram function is parameterless; instead, it loops through all of the objects in the static map checking invariants and calling the Destroy method. The *keystone* specific policy classes only consist of *endOfConstructor* functions. There is one function defined for each type in the hierarchy. An endOfConstructor function should create the appropriate invariants for the type being constructed and add an entry into the static map associating the newly created object with the newly created invariants.

4.2 Weaving aspects

The temporal invariant implementation is woven into *keystone* using aspects implemented as policy classes. The join points are the same as traditional class invariants: the end of the constructor, the beginning and end of each method, and the beginning of the destructor. An additional join point representing the end of the program is also required to check and destroy any remaining eventually valid invariants.

The *keystone* parser and front-end project was already well-developed [19] before we decided to use temporal invariants to validate the three most important hierarchies. To incorporate temporal invariants as policies, we use typedefs rather than the usual approach using templates. The typedef mechanism includes the advantage of templates, in that they can be easily changed at compile time, while minimizing the number of changes required for integration of invariants.

Figure 5 illustrates a function with temporal invariants woven into its join points. Policy is a type-

```
(1)  NameDeclaration*
     Scope::lookup( NameOccurrence* occur ) {
(2)      Policy::beginningOfMethod( this );
(3)      //... original method here
(4)      Policy::endOfMethod( this );
(5)  }
```

Figure 5. Example of woven aspects.

def for ScopeInvariantPolicy. The current object's address, represented in C++as the keyword this, is passed as a parameter to the policy function. For *keystone*, the weaving was done manually; however, a reengineering tool could have also been used.

5 Case Study

In this section we describe the results of our study of temporal invariant validation. The target application for our study is *keystone* [19, 24], a parser and front-end for the ISO C++ language [13]. The validator was executed on a $DellPrecision^{TM}$ 530 workstation with $Intel^{©}$ $Xeon^{TM}$ 1.7 GHz processor equipped with 512 MB of RDRAM, running the Red Hat Linux 7.1 operating system. Our implementation language is C++ [28], compiled with $GNU^{©}$ *gcc* version 2.96.

In the next section we describe the test suite for the study. In Section 5.2 we compare, for each kind of temporal invariant, the number of temporal invariant objects generated to the average number of times the invariants are checked during execution of each test case. In Section 5.3 we compare the cost of validating temporal invariants with two other approaches to invariant validation [8, 10]. In Section 5.4 we discuss the faults that were exposed through temporal invariant validation and in Section 5.5 we describe the threats that preclude generalization of our results.

5.1 The test suite

The table in Figure 6 summarizes our suite of eight test cases, listed in the rows of the table as encrypt, Clause 3, Clause 14, php2cpp, fft, graphdraw, ep matrix and vkey. The test cases in the suite were chosen because of their range and variety of application and to provide statement adequate coverage of *keystone*. The test cases are mostly listed in sorted order by number of lines of code, not including comments or blank lines, except for Clauses 3 and 14, which we discuss shortly.

Test case	lines	classes	classes w/ fns
encrypt	946	1	1
Clause 3	952	40	34
Clause 14	2,403	446	440
php2cpp	1,920	6	6
fft	2,238	51	36
graphdraw	4,354	199	76
ep matrix	4,944	78	51
vkey	8,556	279	44

Figure 6. Test suite.

Test case encrypt is an encryption program that uses the Vignere algorithm [1]. Clause 3 and Clause 14 are sequences of examples taken from Clauses 3 and 14 of the ISO C++ standard for testing name lookup and templates respectively [13, 20]. The php2cpp test case converts the PHP web publishing language to C++ [6] and fft performs fast Fourier transforms [17]. graphdraw is a drawing application that uses *IV Tools* [30], 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 [27]. vkey is a GUI application that uses the *V GUI* library [31], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications.

The columns of the table in Figure 6 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. We list both classes and classes with functions in Figure 6 to distinguish true object-oriented classes from C structures, as illustrated in the final two columns of the figure. All of the test cases are complete applications, except Clause 3 and Clause 14, and three test cases use large libraries: ep matrix, vkey and graphdraw use the *NTL*, *V GUI* and *IV Tools* libraries respectively.

Test case	Eventually: No. generated	Eventually: No. checked	Always: No. generated	Always: No. checked
encrypt	14,357	14,611	4,652	58,824
Clause 3	6,695	6,900	2,196	19,513
Clause 14	16,506	16,925	5,217	43,126
php2cpp	16,862	17,017	5,428	71,322
fft	32,939	33,229	10,414	107,778
graphdraw	68,560	70,855	20,518	240,249
ep matrix	108,077	108,729	34,233	444,999
vkey	112,088	115,365	30,971	284,072

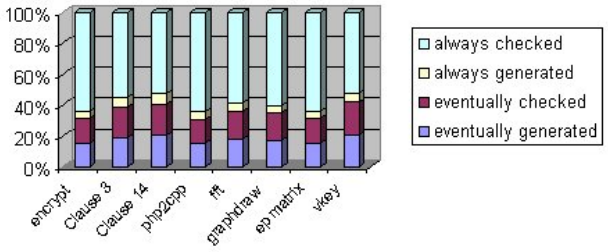


Figure 7. Temporal invariants generated and checked.

5.2 Invariant objects generated and checked

We now compare the performance of our aspect-oriented implementation of temporal invariants with two other approaches. The first part of our study is to compare the number of *eventually valid*, EventuallyInvariant, objects generated, and the number of *always valid*, AlwaysInvariant, objects generated, with the average number of times they were checked, (Section 3 provides detail about these invariants).

The results of the first part of our study are listed in Figure 7. The rows of the table in the figure list each of the test cases and the columns list the number of eventually valid invariant objects generated, Eventually: No. generated, the number of times the eventually valid invariants were checked, Eventually: No. checked, the number of always valid invariant objects generated, Always: No. generated, and the number of times always valid invariants were checked, Always: No. checked. For example, the first row of the table shows that for the encrypt test case 14,357 eventually valid invariant objects were generated, and they were checked 14,611 times; 4,652 always valid invariants were generated and they were checked 58,824 times. The number of eventually valid invariant objects checked is always slightly larger than the number generated because most are valid the first time; however, a few of the eventually valid invariants had to be checked two or more times before they became valid.

The graph in Figure 7 dramatizes the number of times always valid invariant objects were checked. Each bar in the graph corresponds to a row of the table and each subsection of a bar corresponds to a particular column within a row. For example, the first bar of the graph depicts results for the encrypt test case, where the subsection at the bottom of the bar represents the number of eventually valid invariant objects generated, the next subsection of the bar represents the number of times eventually valid invariant objects were checked, the third subsection represents the number of always valid invariant objects generated and the top subsection of the first bar represents the number of times the always valid invariant objects were checked.

The second subsection of each bar is, for each test case, only slightly larger than the first subsection, illustrating the fact that eventually valid temporal invariants are valid, for the most part, the first time they are checked. The third subsection of each bar is small, indicating that there are fewer always valid invariants generated; however, the fourth or top subsection of each bar is, by far, the largest of the four subsections, illustrating the fact that always valid invariant objects must be checked during the entire execution of the test case. For example, eventually valid invariants are checked 1.02 times during the average execution of a test case. However, the always

Test case	ATT	TI	EOP	None
encrypt	11.62	0.96	0.53	0.53
php2cpp	6.66	1.24	0.77	0.77
fft	25.84	3.32	2.37	2.33
graphdraw	54.22	7.55	5.43	5.40
ep matrix	175.56	35.04	29.42	29.29
vkey	133.22	19.01	15.53	15.53

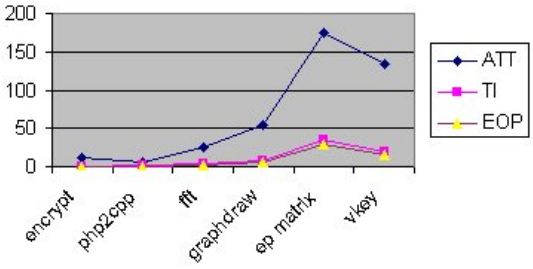


Figure 8. Efficiency results.

valid invariants were checked 11.17 times during the average execution of a test case.

5.3 Efficiency of temporal invariants

We now compare the efficiency of our aspect-oriented implementation of temporal invariant validation with two other approaches to invariant validation. Reference [10] describes the use of an acyclic visitor to validate invariants at the end of a program, **EOP**. Reference [8] describes the use of the Decorator pattern to validate invariants at the end of constructor execution, at the beginning of destructor execution and at the beginning and end of all methods, **ATT**. Figure 8 summarizes our results for comparing the efficiency of validating temporal invariants, **TI**, with the other approaches. Each experiment was measured with the time facility, with the average time for ten executions reported. In each experiment, the invariants are the same except for the TI approach, where each invariant is qualified as *eventually valid* or *always valid*.

The columns of the table of Figure 8 list experimental results for the three approaches: **ATT**, **TI**, **EOP**; the final column of the table, **None**, lists results for executing the test case with no validation. The rows of the table list results for each test case. For example, the encrypt test case required 11.62 seconds for ATT, 0.96 seconds for TI, 0.53 seconds for EOP and 0.53 seconds for None. Thus, validating at the end of the program, EOP, required virtually the same time as no validation; validating temporal invariants, TI, required 0.96 seconds, only a little more than EOP; and validating all the time, ATT, required 11.62 seconds, which is considerably longer than the other two approaches.

The graph in Figure 8 further illustrates this timing comparison where the top line in the graph represents timings for the ATT approach and the two lines at the bottom of the graph represent timings for the TI and EOP approaches. Clearly, the ATT approach required much more time, in our case study, than the TI or EOP approaches.

5.4 Errors uncovered through validation

The target application of our study, *keystone*, is an ongoing project whose initial inception was first tested using statement coverage and then validated by checking invariants at the *end of the program*, EOP, as described in reference [10]. During this initial validation of *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 and the 6 implementation errors have also been repaired [10].

Using temporal invariants we uncovered two additional errors not discovered using the EOP approach of reference [10]. The drawback of the EOP approach is that it will not uncover errors that may have existed early in the execution of the program but were corrected as execution progressed. We uncovered two such errors using our temporal invariant approach. For example, when the keystone parser encountered an expression such as $\sim x$, it always installed x into the symbol table as a destructor; however, it is also possible that $\sim x$ is an expression for bitwise complement of x. In the latter case, the original *keystone* would not have correctly identified x. Since x had been incorrectly identified as a destructor, the *keystone* front-end would have later assigned a type to x, covering the error. However, using always valid temporal invariants, the error is uncovered before *keystone* can incorrectly generate a type for x.

Using validation at the end of constructors, at the beginning of destructors and at the beginning and end of methods, ATT, uncovered the same two additional errors that the TI approach uncovered.

Recently, *keystone* was extended to include template structures and we have used the test cases described in Figure 6 together with test cases extracted from the clauses of the ISO C++ standard for regression testing [20]. During the extension of *keystone*, we discovered that invariant validation continually kept our maintenance activity on track, signaling when a fault had been introduced into the application.

5.5 Discussion

In our approach, temporal invariants are validated dynamically so that the potential for 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. For example, if none of the test cases use templates than the temporal invariants in class TemplateParameterScope will not be validated. Using *gcov* to measure coverage, our test suite provides *92%* coverage of the statements of the code and *100%* coverage of the temporal invariants. Nevertheless, there are several threats that preclude fully generalizing the results of our case study.

The first threat relates to the efficiency of our approach. We have shown in Section 5.2 that for the *keystone* parser and front-end, we generated more eventually valid invariant objects than always valid invariant objects. Since eventually valid invariants are checked less frequently than always valid invariants, the TI validation was almost as efficient as not validating invariants. However, in the worst case, all of the temporal invariants might be always valid invariants and TI validation will be as expensive as ATT.

The second threat relates to validation strength, or the ability to uncover errors, provided by the various approaches. We have shown that ATT and TI uncovered two additional errors not uncovered by EOP. However, our approach requires further investigation into a comparison of validation strength provided by ATT and TI. Nevertheless, temporal invariants clearly provide more expressivity than traditional class invariants.

6 Related Work

Design by Contract was originally advocated by Turing [12], further developed by Hoare [11], Floyd [9] and Dijkstra [7] and made popular in the Eiffel language [23]. However, Eiffel has not gained the wide popularity of other object-oriented languages such as C++ and Java. Nevertheless, the abundance of research into contracts is testimony to their importance [5, 10, 14, 18, 23, 25, 26, 29]. Due to space constraints, we review two works that relate to our paper. The first studies the effects of using aspects and exception handling to enforce invariant validation in Java [18]. The second describes OCL extensions to accommodate time-based constraints and derive test patterns to validate distributed components [25].

Lippert and Lopes describe the reengineering of an existing Java framework, JWAM, to investigate the effects of aspect-oriented programming on exception detection and handling [18]. JWAM uses the Design by Contract methodology to ensure that callers do not misuse methods and that the basic implementation preserves the original specification. In JWAM, a broken contract results in a thrown exception. The contracts in JWAM were originally specified as comments in the Java code, but the researchers reengineered the contracts as aspects using AspectJ [15]. The research demonstrated that aspects significantly reduced the amount of exception handling code (LOC) needed to validate contracts. The work in reference [18] uses AspectJ to weave traditional class invariants into Java programs. In our work we use policy classes to weave temporal invariants into a C++ application.

Ramakrishnan and McGregor describe extensions to the OCL grammar to accommodate temporal operators *always*, *never*, *eventually*, *until*, *unless* and *since* [25]. These extensions enable the developer to group class constraints in time order by concatenating individual class invariants. The work proposes to generate test patterns using the invariants; however, the authors fail to provide detail about the process of test pattern generation. Nevertheless, the extensions to OCL described in reference [25] provide a basis for the specification of our OCL based temporal invariants.

7 Conclusions and Future Work

In this paper we have described the use of temporal invariants to capture assertions about class attributes that are not easily captured using traditional class invariants [22]. We have described a reengineering technique to weave temporal invariants into an application using aspects [16] implemented as policy classes [2]. We have compared our approach to two other approaches and shown that, for our case study application, temporal invariant validation is not as expensive as validation with traditional class invariants. We have described some faults that temporal invariants uncovered that were not uncovered using invariant validation at the end of a program.

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] Boost. Boost C++. www.boost.org, May 2002.

[5] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *International Symposium on Software Testing and Analysis (ISSTA'2002)*, pages 70–80, Rome, Italy, July 2002.

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

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

[8] E. B. Duffy, J. P. Gibson, and B. A. Malloy. Applying the decorator pattern for non-intrusive profiling of object-oriented software. Technical Report CU-7, Clemson University, September 2002.

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

[10] T. H. Gibbs, B. A. Malloy, and J. F. Power. Automated Validation of Class Invariants In C++ Applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE'2002)*, pages 205–214, Edinburgh, UK, September 2002.

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

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

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

[14] J. M. Jezequel, D. Deveaux, and Y. Le Traon. Reliable objects: Lightweight testing for OO languages. *IEEE Software*, 18(4):76–83, 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *CACM*, 44:59–65, October 2001.

[16] G. Kiczales, J. Lamping, A. Mendhedar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97, LNCS 1241*, pages 220–242, Jyvskyl, Finland, 1997. Springer-Verlag.

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

[18] M. Lippert and C.V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427, Limerick, Ireland, 2000.

[19] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 2002. (to appear).

[20] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power. Testing c++ compilers for iso language conformance. *Dr. Dobbs Journal*, pages 71–80, June 2002.

[21] R. Martin. Acyclic visitor. In *Joint Pattern Languages of Programs Conferences (PLoP'96)*, number wucs-97-07, September 4–6 1996.

[22] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–52, 1992.

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

[24] 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.

[25] S. Ramakrishnan and J. D. McGregor. Modelling and testing oo distributed systems with temporal logic formalisms. In *Proceedings of the 18th International IASTED Conference Applied Informatics*, Innsbruck, Austria, September 2000.

[26] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

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

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

[29] A. Tiwari, H. Ruez, H. Saidi, and N. Shankar. A technique for invariant generation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 86–89, Paris, France, April 2001.

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

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