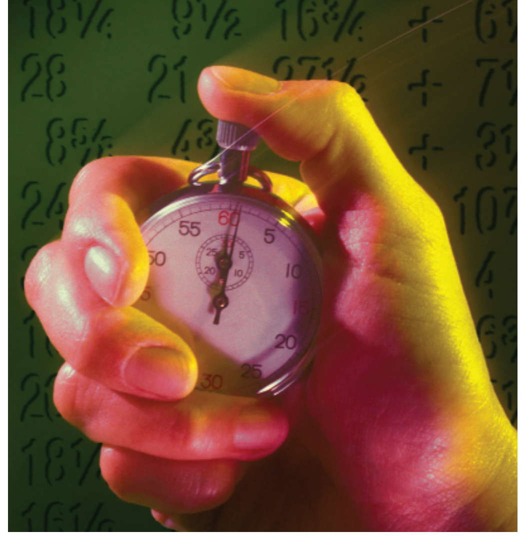


Assertions can help improve software quality. To use them effectively, it's important to determine when they are valid.

Brian A. Malloy and Jeffrey M. Voas

Programming with Assertions: A Prospectus

A recent report states that after five decades of software development, defective software is the norm, high-quality software the exception ("50 Years of Software: Key Principles for Quality," J.A. Whittaker and J.M. Voas, *IT Professional*, Nov.-Dec. 2002, pp. 28-35). This is a disappointing observation in view of the many advances that software development has realized, including the emergence of software process improvement and the adoption of object technology, including a Unified Modeling Language (UML).

All these advances facilitate the design of object-oriented systems and the development of component-based software. Clearly, to address the quality problem for software, developers need a technology that specializes in producing robust software. One widely used process that supports the construction of quality software is testing, which executes the program with input data or test cases, and then compares the output data to expected results.

However, the transfer of techniques from research to practice has been slow because many techniques do not scale to real programs. Many are also too difficult for the average practitioner to grasp. Also, inadequacies in the testing infrastructure have significant impact on software developers and industry users (*The Economic Impacts of Inadequate Infrastructure for Software Testing*, tech. report, Nat'l Inst. for Standards and Tech., May 2002). With all these hurdles, it's no wonder why practitioners are looking for alternatives to testing.

Inside

Background on Assertions

Temporal Invariants in C++

Recent Developments in Assertion Use

ONE ALTERNATIVE: ASSERTIONS

An alternative to testing that has increasing popularity and usage entails the use of assertions to monitor the data attributes of functions or classes. *Assertions* are formal constraints on the behavior of a software application; Alan Turing originally advocated their use in 1950 ("The Emperor's Old Clothes," C. Hoare, *Comm. ACM*, Feb. 1981):

> An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

Anthony Hoare, Robert Floyd, and then Edsger Dijkstra further developed the concept of assertion-directed programming. The first language to include support for programming with assertions was Algol W, a variant of Algol developed by Nicholas Wirth and Hoare. Other languages that include assertion support are Sather and Eiffel. Special techniques also incorporate assertions into

1520-9202/04/$20.00 © 2004 IEEE

existing general-purpose languages, such as Pascal, Ada, C, C++, and Java. The advantage gained in using assertions is that they provide a basis for determining if a program is correct: The program is correct if its implementation is consistent with the assertions. The "Background on Assertions" sidebar lists selected references to this development and these languages.

Here, we provide a prospectus on programming with assertions, including a summary of their important features and the prospects of exploiting assertions to improve software applications.

OVERVIEW

As defined earlier, an assertion is a formal constraint on the behavior of a software application. Programmers commonly write such a constraint as an annotation, and it usually describes what the application is supposed to do, rather than how the application should do it. An effective assertion does not restate code statements but rather states an important property of the application, succinctly and unambiguously. The use of assertions originated, of course, in the procedural realm, where programmers use them to

- specify preconditions on a function,
- specify postconditions on a function,
- place constraints on the function's return value, or
- specify a constraint on an intermediate state of a function.

Developers have deployed assertions in applications written in procedural and object-oriented languages. Because the types of assertions deployed in procedural software are, for the most part, a subset of those in object-oriented software, our main focus here is object-oriented languages.

Assertions in object-oriented applications

Eiffel's design by contract methodology describes the use of assertions. The notion of a *contract* in object-oriented programming describes the relationship between a supplier class and a client class; it is expressible as preconditions and postconditions on methods, and invariants on classes.

Figure 1 illustrates a client class, Bank (on the figure's left); and a supplier class to Bank, the Account hierarchy (shown on the figure's right). The Account hierarchy represents a simple bank account; it consists of an abstract base class and two derived classes for a specializing Account. Derived class SavingsAccount represents a simple savings account where the bank pays interest on balances that we assume remain positive. Class CheckingAccount represents a checking account with overdraft facilities, on which the bank levies an interest fee, overdraftRate.

To illustrate the notion of assertions, Figure 2 lists some class invariants that a programmer might use as constraints on the bank account of Figure 1. A *class invariant* is an assertion or formal constraint on the behavior of a data attribute of the objects in a given class. Programmers write the formal constraint as an annotation, using either a programming or a formal language. The Object Constraint Language (OCL) is a formal language used to describe expressions on models specified in the UML.

The class invariants for the Account hierarchy, illustrated in Figure 2, provide a flavor of class invariants and OCL expressions. The phrase context SavingsAccount on line 1 in the figure provides the class context in which to apply the invariants on lines 2 to 4. These invariants insist that for any SavingsAccount object, the balance must be positive, and the interest rate must be between 0 and 100. Similarly for the CheckingAccount class, we assert that the balance must not go below the overdraft limit, expressed as a negative amount.

Invariants have several interesting aspects. First, for what classes do the invariants hold; for example, do the invariants on a base class hold for the derived classes? Second, where in the program do the assertions hold?

Background on Assertions

Development of assertions

- **"An Axiomatic Basis for Computer Programming," C. Hoare, *Comm. ACM*, Feb. 1969, pp. 576-583.**
- **"Proof of Correctness of Data Representations," C. Hoare, *Acta Informatica*, vol. 1, 1972, pp. 271-281.**
- **"Assigning Meanings to Programs," R. Floyd, *Proc. Am. Mathematical Soc. Symp. Applied Mathematics*, vol. 19, 1967, pp. 19-31.**
- ***A Discipline of Programming*, E.W. Dijkstra, Prentice Hall, 1976.**

Languages incorporating assertions

- **Sather: Object-Oriented Programming, J. Feldman and colleagues, Oct. 2003; http://www.icsi.berkeley.edu/~sather/.**
- ***Object-Oriented Software Construction*, 2nd ed., B. Meyer, Prentice Hall, 1997.**

Assertions in object-oriented languages (Object Constraint Language)

- ***The Object Constraint Language: Precise Modeling with UML*, J. Warmer and A. Kleppe, Addison-Wesley, 1999.**

ASSERTIONS AND INHERITANCE: TYPE SYSTEMS

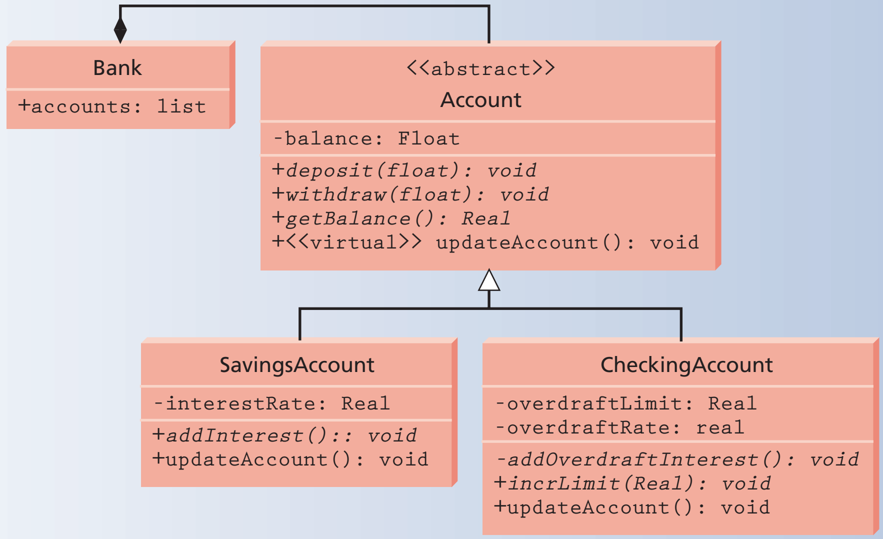
It's easier to incorporate assertions into applications if they are somehow included in the programming language used to implement the application. Some languages, such as Sather and Eiffel, include assertions as part of the language; Java has incorporated assertions since version 1.4 (*Preliminary Design of JML: A Behavioral Interface Specification Language for Java*; G.T. Leavens, A.L. Baker, and C. Ruby; tech. report 98-06i Iowa State Univ., 2000).

It is important to note here that we are not talking about the assert statement included in C, C++, and Java, although you can use this statement to facilitate the implementation of the assertions that we describe here. The assertions that we describe here are predicates that describe a fact or condition about the data attributes of a class that the members of the class must maintain. In contrast, the assert statement in C, C++, and Java is a construct that programmers can use to *check* that assertions remain inviolate.

There have been several approaches to incorporating assertions into existing general-purpose languages such as C++ and early versions of Java. In programming with assertions, it is important to consider the interplay between the assertions and the underlying type system of the implementation language. In fact, the type system forms a set of assertions guaranteeing that the application does not violate the type system's rules. Thus, assertions used in an application should work in tandem with the underlying type system. A major role for assertions is to specify properties similar to types, but that a type checker cannot check ("A Practical Approach to Programming with Assertions," D.S. Rosenblum, *IEEE Trans. Software Eng.*, Jan. 1995, pp. 19-31).

Next, we explore the interplay between the type system and assertions to demonstrate the power of assertions to augment the type system for polymorphism, dynamic binding, and method overriding.

Figure 1. Bank account.



This figure illustrates client class Bank, shown on the figure's left, and a supplier class to Bank, the Account hierarchy, on the figure's right. The Account hierarchy represents a simple bank account and consists of an abstract base class and two derived classes for specialized instances of Account.

Figure 2. Sample class invariants.

```
context SavingsAccount
  inv balance: self.getBalance() >= 0
  inv rateIsPercent: self.interestRate >= 0 and
                     self.interestRate < 100
context CheckingAccount
  inv balance: self.getBalance() >= self.overdraftLimit
  inv negativeLimit: self.overdraftLimit <= 0
```

This figure lists class invariants, expressed in OCL, for the SavingsAccount and CheckingAccount classes. OCL expressions, when evaluated, do not have side effects, so their evaluation cannot alter the state of the corresponding executing system, even though you can use an OCL expression to describe a state change. OCL expressions allow the modeler to express invariants in a language-independent manner.

Static typing and dynamic binding

Programming languages are either statically or dynamically typed. The notion of *static typing* means that the text of the system can indicate a variable's type. In *dynamic typing*, the variable's type can remain unknown until the system is running. Many researchers claim that the effective use of object technology requires static typing. Moreover, the object-oriented approach is typically based on the notion of type together with the concept of a module to produce the class construct. Inheritance hierarchies provide considerable flexibility for manipulating objects polymorphically, while retaining the safety of static typing. Virtual functions require dynamic binding to attach the invocation to the proper method. However, dynamic binding can complicate reasoning about program correctness.

To illustrate polymorphism and dynamic binding, consider again the base class Account and derived classes SavingsAccount and CheckingAccount in Figure 1. Consider also the following C++ statement: `Account *acc = new SavingsAccount`. This statement declares *acc* to be of type Account, permitting *acc* to polymorphically refer to an instance of any class derived from Account. In this case, the type of object to which *acc* refers is a SavingsAccount; however, the types of objects to which *acc* can refer are determined statically. Moreover, the following invocation of virtual method updateAccount, acc→updateAccount(), is dynamically bound to the implementation of updateAccount in SavingsAccount, since virtual method updateAccount in Account is overridden in SavingsAccount.

Assertion use has expanded to areas such as isolating the location of faults in Java programs.

Class invariants, a specific type of assertion described earlier, play an important role in inheritance. Because class invariants, like preconditions and postconditions, must be valid before and after method invocation, it's tempting to simply include invariants as part of the preconditions and postconditions. The first problem with this inclusion is that redundancy will probably make the preconditions and postconditions unnecessarily complicated. However, the second problem is more important: By including invariants as part of preconditions and postconditions, we lose the notion that there are assertions that hold for the class as a whole.

Moreover, an important rule that must apply to inheritance is that the invariants of a class are the Boolean AND of the invariants specified for the class and the invariants specified for any parents of the class. This property reflects one of inheritance's basic characteristics: Properties provable using the specification of an object's presumed type should hold even though the object is actually a member of a class derived from that type ("A Behavioral Notion of Subtyping," B.H. Liskov and J.M. Wing, *ACM Trans. Programming Languages and Systems*, Nov. 1994, pp. 1811-1841).

Returning to our savings account example, if a client of Account calls the updateAccount method, it expects the method to preserve the semantics of updateAccount, even if it is actually calling a method overridden in a derived class of Account. Assertions provide a mechanism whereby the client can remain assured that the derived class will preserve these semantics, and that the updateAccount method will work correctly, even in a derived class, because inheritance always causes assertions to pass on to descendants. Thus, you can use assertions to preserve the Liskov Substitution Principle: Clients that use pointers or references to a base class must be able to use objects of the derived class without knowing they are doing so ("Data Abstraction and Hierarchy," B.H. Liskov, *SIGPLAN Notices*, May 1988).

Overloading and overriding

In many languages, it is possible to overload a function name so that the program uses the same name for several functions; the functions are distinguishable by their parameters. This permits programmers to use the same name for functions that implement the same concept for different types. For example, the name `sqr` can be overloaded for `int sqr(int)` and `float sqr(float)` so that a programmer can reuse these names for functions that perform the same task for different types, such as `int` and `float`. You can think of dynamic binding of a member function to an invocation as a type of overloading that the program execution resolves at runtime. Object-oriented languages permit methods in a derived class to override methods in a base class, where the name, return type, and parameters of the methods are identical; that is, there is *novariance* in the parameters and return type. Novariance provides type safety; however, the problem with the novariance rule on parameters and return types is that it is overly strict and prohibits some safe method overriding. The terms covariance and contravariance describe the relaxation of the novariance rule.

Covariance means that a method in a derived class can override a method in a base class provided that the parameters and the return type in the overridden method are the same type or a subtype of those in the base class. Eiffel uses covariance together with assertions to augment the Eiffel type system and enforce type safety. C++ originally used novariance for both parameters and return types, but relaxed this novariance rule to covariance for parameters.

Contravariance refers to relaxing the novariance rule to permit overriding for methods whose parameters and return type are identical or a superclass. Sather permits contravariance for overriding methods.

TEMPORAL INVARIANTS

The usual approach for specifying the places in an object-oriented application where a class invariant must hold is to do so

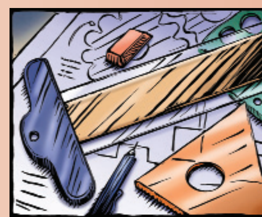
- after class instantiation (after the constructor invocation),

- before and after every method invocation by another object, and
- before every destructor invocation.

Class invariants are not required to remain valid during method execution, only before and after invocation.

However, the problem with the traditional notion of class invariants is that some important assertions about a class are initially invalid. This is because of circular dependencies or the monotonically increasing nature of the system under construction. For example, during program compilation, the compiler constructs a name object to facilitate name lookup. However, certain fields in the name object, such as the corresponding scope of the name, might be unknown until after name lookup has occurred ("Symbol Table Construction and Name Lookup in ISO C++," J.F. Power and B.A. Malloy, *Proc. Tech. Object-Oriented Languages and Systems*, IEEE CS Press, 2001, pp. 57-68).

Temporal Invariants in C++



- ***Applying the Decorator Pattern for Non-Intrusive Profiling of Object-Oriented Software*, E.B. Duffy, J.P. Gibson, and B.A. Malloy, Tech. Report CU-7, Clemson Univ., Sept. 2002.**
- **"Weaving Aspects into C++ Applications for Validation of Temporal Invariants," T.H. Gibbs and B.A. Malloy, *Proc. 7th European Conf. Software Maintenance and Reengineering*, IEEE Press, 2003, pp. 249-258: This is a detailed account of the case studies that we mention in this article.**
- **"Automated Validation of Class Invariants In C++ Applications," T.H. Gibbs, B.A. Malloy, and J.F. Power, *Proc. Int'l Conf. Automated Software Engineering*, 2002, pp. 205-214: This is the source of the EOP results.**

Moreover, assertions about a class might have an important impact on memory management. For example, during program execution the programmer might delete some heap-based objects of a class if the references to these objects will be lost; otherwise, system performance will degrade. Additionally, the operating system might delete the remaining heap-based objects after the program terminates. Thus, some assertions about class attributes might be initially invalid, but become valid as the program executes, and they might have an important impact on the system under construction.

To work around this problem, programmers can use temporal invariants, which provide more expressivity than traditional class invariants. The work listed in the "Temporal Invariants in C++" sidebar provides a basis for the following discussion.

Temporal invariants

Temporal invariants are assertions about a class that will qualify to one of four levels: eventually, always, never, or already. 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 the always-valid invariant should be checked at program termination.

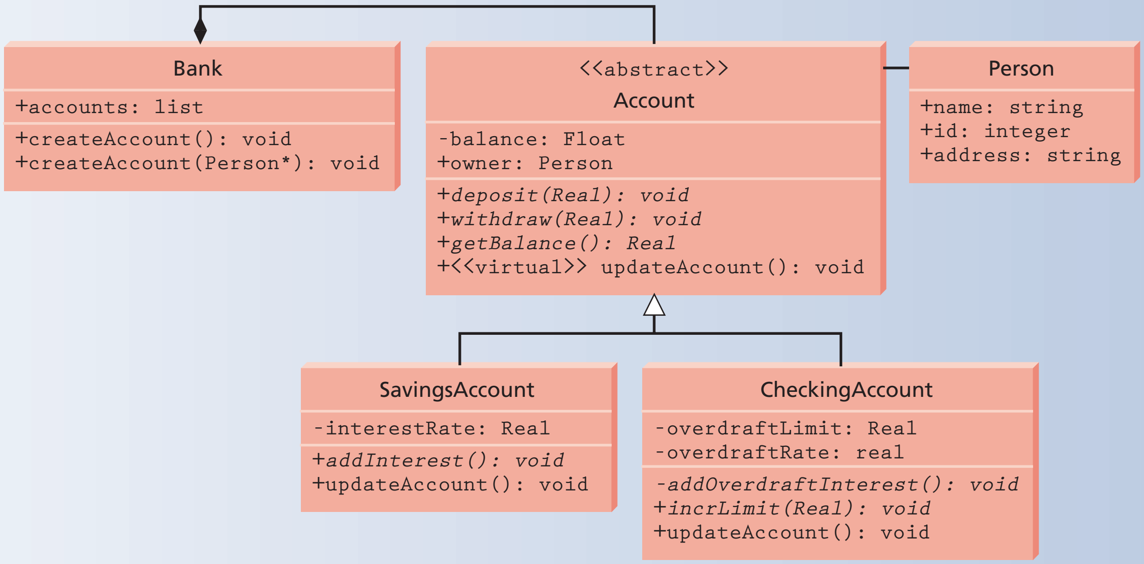
An *eventually valid* invariant is an assertion about a class that must become valid before an instance of the class reaches the end of a destructor or before program termination. A *never-valid* invariant is equivalent to an always-valid invariant with a negated assertion. Finally, an *already-valid* invariant is an assertion about a class that must be valid at the beginning of a constructor. An already-valid invariant might, for example, describe a file that must be open before the creation of a class instance.

To illustrate temporal invariants, Figure 3 shows the bank account example of Figure 1, except that we have added overloaded functions createAccount to the Bank class and to the Person class associated with the Account class through the owner attribute. Figure 4 illustrates some temporal invariants for the CheckingAccount class of Figure 3. The balance and negativeLimit invariants, lines 2 and 3, are similar to those listed in Figure 2, except that they now have the always-invariant designation. Also, on line 4 of Figure 4, we added the ownerNotNull invariant to the CheckingAccount class as an eventually valid temporal invariant. In this way, temporal invariants permit the user to create a bank account without knowing the owner details at creation time; the eventually valid invariant guarantees that the user must eventually provide these details.

Efficiency of temporal invariants

The case study of a C++ system (cited in the sidebar) used aspects to weave assertions into join points using policies (*Modern C++ Design: Generic Programming and Design Patterns Applied*, A. Alexandrescu, Addison-Wesley, 2001). It investigates the effectiveness of temporal invariants and compares the aspect-oriented implementation's performance with class invariant validation at the end of a program, EOP. The case study also includes a validation of class invariants at the end of constructors, at the beginning of destructors, and at the beginning and end of methods, which we refer to as "all the time" or ATT. Expressed in OCL, the temporal invariants have extensions to accommodate temporal operators.

Figure 3. Using temporal invariants.



In this example, the user can create an account without knowing the account owner details. Temporal invariants can provide assurance that the user will eventually provide these details.

Figure 4. Temporal invariants for bank account.

```
context CheckingAccount
 inv balance:
 always(self.getBalance() >= self.overdraftLimit)
 inv negativeLimit: always(self.overdraftLimit
       <= 0)
 inv ownerNotNull: eventually(owner <> NULL)
```

This figure lists temporal invariants, expressed in OCL, for the CheckingAccount class of Figure 3.

Recent Developments in Assertion Use

- **"Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code," L.C. Briand, Y. Labiche, and H. Sun, *Int'l Symp. Software Testing and Analysis* (ISSTA 02), ACM Press, 2002, pp. 70-80.**
- **"Korat: Automated Testing Based on Java Predicates," C. Boyapati, S. Khurshid, and D. Marinov, *Int'l Symp. Software Testing and Analysis* (ISSTA02), ACM Press, 2002, pp. 123-133.**

Figure 5 summarizes the results of comparing the efficiency of validating temporal invariants, TI, with the other approaches. The experimental measurements report the average time for 10 executions. In each experiment, the invariants are the same except for the TI approach, which qualifies each invariant as eventually valid or always valid.

The accompanying Table 1 summarizes our suite of six test cases—encrypt, php2cpp, fft, graphdraw, ep matrix, and vkey. We chose these test cases because of their range and variety of application, and to provide statement-adequate coverage of our case study application, keystone, a parser and front-end for ISO C++ ("Decorating Tokens to Facilitate Recognition of Ambiguous Language Constructs," B.A. Malloy, J.F. Power, and T.H. Gibbs, *Software: Practice and Experience*, Jan. 2003, pp. 19-39).

The columns of the table of Figure 5 list experimental results for the three approaches: TI, EOP, and ATT. The final column lists results for executing the test case with no validation. For example, on encrypt, validating invariants at the end of the program, EOP, required virtually the same time as no validation. Validating temporal invariants, TI, required 0.96 s, only a little more than EOP. Validating all the time, ATT, required 11.62 s, considerably longer than the other two approaches.

Figure 5 further illustrates this timing comparison; the top line represents timings for the ATT approach and the two lines at the bottom represent the TI and EOP approaches. Clearly, the ATT approach required more time than the other two approaches.

Thus, temporal invariants are more expressive than traditional invariants and can be more efficient. Moreover, temporal invariants uncovered errors not uncovered as part of the testing process. They also uncovered more errors than simply validating invariants at the end of the program. In our experiments, the TI approach permitted us to write eventually valid invariants. We did not include these invariants in the ATT set because they would have been initially invalid. TI also uncovered one additional error in keystone that we had not found using ATT or EOP approaches.

Figure 5. Efficiency results.

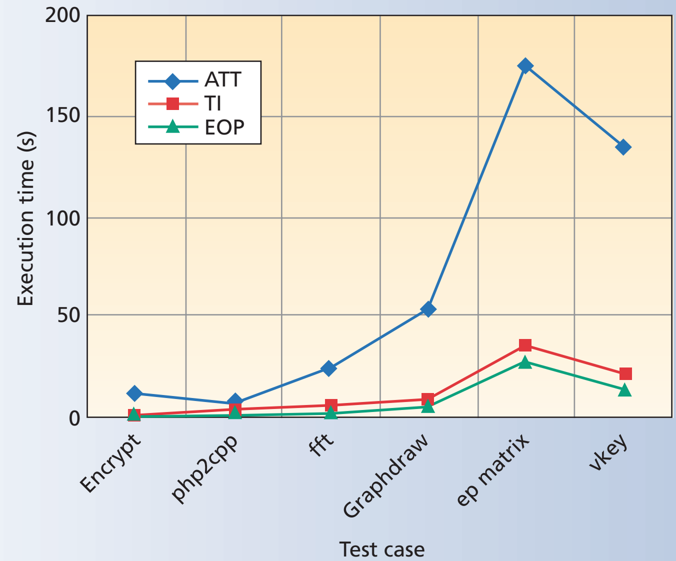


Table 1. Experimental results for various approaches to assertion application.

	Average time for 10 executions (s)			
Test case	TI	EOP	ATT	No validation
encrypt	0.96	0.53	11.62	0.53
php2cpp	1.24	0.77	6.66	0.77
fft	3.32	2.37	25.84	2.33
graphdraw	7.55	5.43	54.22	5.40
ep matrix	35.04	29.42	175.56	29.29
vkey	19.01	15.53	133.22	15.53

ASSERTIONS: USAGE AND PROSPECTS

The traditional use of assertion-directed programming is to provide a basis for determining program correctness and thereby facilitating the construction of higher-quality software systems.

However, recent developments have expanded assertion use into other aspects of the software life cycle; we list some sources in the "Recent Developments in Assertion Use" sidebar. For example, Briand and colleagues have investigated the use of assertions to isolate the location of faults in Java programs. They used OCL to express the assertions and seeded faults into an automatic teller machine system that serves as a case study.

Boyapati and colleagues developed a framework for automated testing of data structures in Java programs. Their approach uses method preconditions to construct assertions and, using the assertion, they generate inputs for which the assertion remains true. The approach uses the method postconditions as a test to check the correctness of each output.

Finally, Baudry and colleagues exploit contracts to measure the quality of component-based systems. They describe measures to compute two quality factors: robustness and diagnosability. Their experimental studies, based on applying mutation analysis to OO systems, estimate the overall quality of a system in terms of these two factors.

The use of assertions in traditional ways, together with these extended applications, promises to help address the quality problem in current software development. ■

Brian A. Malloy *is an assistant professor in the Computer Science Department of Clemson University. Contact him at malloy@cs.clemson.edu.*

Jeffrey M. Voas *is chief scientist at Cigital Labs in Dulles, Va. Contact him at jmvoas@cigital.com.*

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.