Adaptable Assertion Checking for Scientific Software Components

Tamara L. Dahlgren
Center for Applied Scientific Computing and Center for Applications Development and Software Engineering
Lawrence Livermore National Laboratory
dahlgren1@llnl.gov

Premkumar T. Devanbu
Department of Computer Science
University of California, Davis
devanbu@cs.ucdavis.edu

Abstract

We present a proposal for lowering the overhead of interface contract checking for science and engineering applications. Run-time enforcement of assertions is a well-known technique for improving the quality of software; however, the performance penalty is often too high for their retention during deployment, especially for long-running applications that depend upon iterative operations. With an efficient adaptive approach the benefits of run-time checking can continue to accrue with minimal overhead. Examples from scientific software interfaces being developed in the high performance computing research community will be used to measure the efficiency and effectiveness of this approach.

1. Introduction

We are proposing the use of a new approach to the checking of assertions in deployed software components; namely, run-time adaptation. The goal of this approach is to provide an efficient, effective mechanism that improves on the standard practice of disabling or even eliminating assertion checking prior to deployment.

For this effort, our definition of a component conforms to that given by Bertrand Meyer [15]; namely, that a component is an *independent* software unit with an interface specification describing how the component should be used. In other words, clients and components are loosely coupled through the component's interfaces. Hence, libraries or subsets of libraries with interface descriptions are considered components. Even commercial, or third-party, binaries can be made into components after-the-fact through their interface definitions.

Consequently, our instantiation of adaptable assertion checking relies on the use of software that wraps the implementation of a component. These wrappers are automatically generated from formal specifications of the interfaces that include behavioral contracts or constraints. These behavioral contracts are automatically transformed into enforcement code.

Unfortunately, the benefits of run-time assertion checking often come at the price of unacceptably high overhead. This is especially true for long-running applications using iterative operations, which are commonly found in scientific computing. One way to retain some of the benefits of assertion enforcement during deployment is to reduce the frequency of checking [13]. And one avenue that, to our knowledge, has not been explored is to achieve this reduction through adaptation.

Liblit *et al.* [13] explore the use of statistical sampling over time to debug infrequently occurring errors by amortizing the cost of assertion checking throughout the user community. In science and engineering applications, however, it is possible to iterate over a given interface hundreds to many thousands of times in a single run. This characteristic provides an opportunity to make effective use of techniques, such as statistical sampling over time, to reduce the frequency assertion checking.

Furthermore, not all assertions are equal in terms of cost or importance. That is, the cost of checking a given assertion can range from cheap (e.g., checking for a null pointer) to expensive (e.g., checking that matrix A is an inverse of matrix B if the product AB is the identity matrix). Similarly, some assertions are merely useful to check while others can be critical, such as a computed value used to drive expensive hardware. In addition, the numerically intensive nature of science and engineering applications provides an opportunity for exploring the use of hierarchies of increasingly expensive assertion implementations. For example, an assertion that requires a matrix be symmetric could be represented by an "inexpensive" check that samples a subset of the elements or by the more expensive yet accurate version that checks every element not on the diagonal.

The remainder of this paper describes the challenges, component contracts, run-time enforcement, and plans for testing the performance and effectiveness of these techniques. The examples that serve as the experimental basis employ the standard mesh interface specification

[19] being developed by the Terascale Simulation Tools and Technologies (TSTT) Center [4]. The TSTT Center is a collaborative effort between researchers from a number of U.S. DOE national laboratories and academic institutions who are focused on the development of "plug-and-play" software components for multiple meshing and discretization technologies.

2. Challenges

There are trade-offs that must be made in order to provide some level of assurance in the quality of software. The primary issue inhibiting the use of run-time assertion enforcement during deployment is the commitment of compute resources. In particular, memory and processing time are often at a premium in high performance computing applications. The trade-offs associated with the flexibility, performance, and effectiveness of our proposal are briefly discussed in this section.

Flexibility is often pitted against efficiency both in terms of memory and performance. Providing the flexibility to specify a range of assertions and to specify multiple assertions inherently requires that the code use a larger memory footprint. The key is to try to minimize that footprint whenever possible. Assertion checking itself inevitably impacts the amount of time it takes to execute a method (also referred to as a *routine* or *operation*), therefore it is very important to minimize the performance overhead. For reduced frequency checking, this is especially important when assertions are not going to be checked for a given invocation.

Reducing the frequency also introduces the issue of effectiveness. That is, decreasing the level of checking inherently reduces the opportunities to catch assertion violations. Determining the balance between an acceptable level of checking and its ability to identify bugs in the code, whether in terms of the application's use of the interface or the interface implementation, is a challenge.

Furthermore, recognizing the fact that not all assertions are equal in terms of cost or importance exacerbates the flexibility versus performance versus effectiveness concerns. Clearly there is additional memory and performance overhead that must be addressed when distinguishing cheap versus expensive as well as hierarchies of assertion checks. In addition, questions about identification and classification of assertions arise. For example, a function call used in an assertion could be cheap if the function does a comparison of a constant and a simple data type variable. Whereas, a function that performs a computation that requires iterating over all of the elements of a matrix would be expensive. The encapsulation of interface specifications basically precludes this information.

3. Component contracts

Since the advances in supercomputing systems are enabling the development of increasingly larger, more complex applications, the developers of those codes are relying more than ever on software elements developed by disparate teams and third-party contributors. Consequently, it's becoming increasingly common for applications to be composed of codes implemented in multiple programming languages. This provides an opportunity to efficiently integrate both automated language interoperability and assertion enforcement technologies. Doing so also enables the reuse of contracts across multiple implementations of an interface.

Therefore, we have integrated support for assertions into the Scientific Interface Definition Language (SIDL) developed by the Components project [5] at Lawrence Livermore National Laboratory (LLNL) [6]. SIDL has been extended to include the standard assertions applicable to interface specifications; namely, class invariants, method preconditions and method postconditions. An *invariant* is used to specify properties that are to remain unchanged throughout the life of a software element instance. A method *precondition* specifies constraints on when it is valid to invoke a method while a *postcondition* constrains its effects.

3.1 SIDL

SIDL is an interoperability specification language for scientific computing. It is used to identify the calling interface of a component being made available to clients implemented in different programming languages. Specifications written in SIDL are automatically transformed into client-server language interoperability code tailored for scientific computing through the Babel compiler [7]. Like other Interface Definition Languages, such as the Object Management Group (OMG) IDL [17] used by the Common Object Request Broker Architecture (CORBA) [16], SIDL is programming- and implementation- language neutral.

Both SIDL and OMG IDL support the modular packaging of full method definitions that specify the type (e.g., integer, float) and direction (i.e., in, out, inout) of each parameter. Both also support multiple inheritance of interfaces, enumerations and arrays.

Unlike OMG IDL, SIDL's basic types include the fundamental science and engineering data types of numeric complex and multi-dimensional, multi-strided arrays. Furthermore, the Babel compiler generates interoperability code optimized for in-process communication and implementation code that conforms to the typical scientific programming paradigms of the underlying implementation languages.

3.2 Assertion extensions

The SIDL grammar has been extended to include constructs inspired by Eiffel [14] to represent the classic interface assertions of class invariants, method preconditions and method postconditions. Like result checking techniques [18, 20], postconditions depend upon the function being computed regardless of the underlying implementation algorithm.

The syntax for the precondition and postcondition assertion clauses is illustrated by the norm method in the VectorUtilities package below. The precondition appears in the *require* clause and specifies that the caller must pass a non-null array into the norm method. The postconditions are given in the *ensure* clause and are based on the mathematical properties of the vector norm. The expression "result >= 0.0" states that the implementation of the norm method shall return a non-negative result. Furthermore, "nearEqual(result, 0,0, 1.0e-9) iff isZero(u, 1.0e-9)" states that the result shall be zero (within the computational tolerance of 1.0e-9) if-and-only-if the parameter, u, is the zero array (i.e., each of u's elements are within the same computational tolerance of 0.0). Note that, in this example, *isZero* is a method that is defined elsewhere in the interface and the tolerance value of 1.0e-9 is used for illustration purposes.

```
package VectorUtilities version 1.0 {
  class Ops {…
    static double norm (in array<double> u)
      require  u != null;
      ensure   result >= 0.0;
               nearEqual(result, 0.0, 1.0e-9)
                        iff isZero(u, 1.0e-9);
  …}}
```

In integrating conditional expressions, the basic operators available in most programming languages were added (e.g., equality, logical and), as were some that are not typically found (e.g., implies, xor, iff). The literals "true", "false", and "null" were also included as was the literal "result" for the return value of a function.

One concern with interface-level assertions is that the encapsulation of private data, used to hide implementation details, precludes their inclusion in assertion expressions. In order to deal with this situation, we have allowed the use of function calls within expressions. There are restrictions, of course, in that the functions must be either local to the interface or be one of the built-in methods automatically generated by the Babel compiler, which include existential and universal quantifiers.

4. Run-time enforcement

Since one of the key challenges to providing efficient run-time enforcement of assertion checking is minimizing the performance overhead, care has been taken to address this issue specifically when assertions are not being checked. Additionally, techniques that reduce the frequency of checking are being explored, starting with two rudimentary adaptation policies: linear and random.

4.1 Efficient execution

Since assertion enforcement is being integrated into language interoperability code, the Babel compiler automatically generates the run-time assertion checks in its Intermediate Object Representation (IOR) of classes. For reasons of portability and performance, IORs are generated in C regardless of the implementation language. Figure 1 illustrates the control flow path from the client through the interoperability code generated by Babel (i.e., the stub, IOR, and skeleton) to the implementation and back. For efficiency, the skeleton layer is generally by-passed when the implementation is also in C.

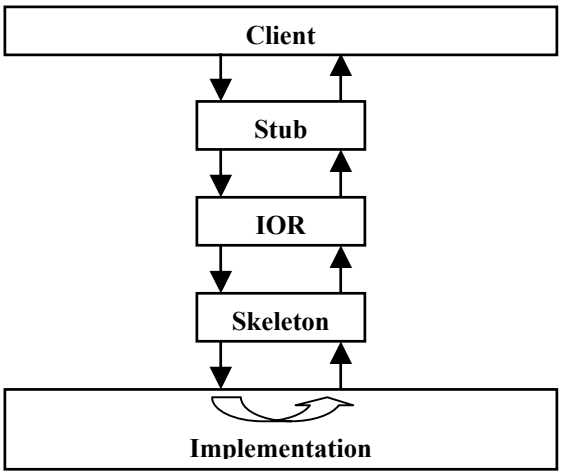

Figure 1. Interoperability control flow

Since assertion enforcement is being generated in the language interoperability wrappers, a variation of the code duplication approaches advocated in [2, 13] has been employed. This approach is illustrated for our norm example in the simplified code fragment below. By leveraging Babel's use of function pointers in the IOR, the only duplication necessary is that of the method call (e.g., call_norm()). It is this call that is wrapped in a fast, uninstrumented path and a slow, instrumented path. When the policy dictates that assertions are not checked, the fast path is taken, which performs only slightly more processing than the method invocation. Otherwise, the

slow path is taken where all preconditions and invariants are checked prior to invocation then all postconditions and invariants are checked after invocation.

```
if (countdown > 1) {
  /* Fast path: decrement countdown and call function */
  countdown = countdown - 1;
  result = call_norm(u);
} else {
  /* Slow path: wrap call with checks, reset countdown */
  okay = true;
  if (u == null) {
    okay = false;
    call_check_error(pre, name, not_null_msg);
  }
  if (okay) {
    result = call_norm(u);
  }
  if (result < 0.0) {
    okay = false;
    call_check_error(post, name, non_negative_msg);
  }
  …
  setNextCountdown(method, okay);
}
return result;
```

4.2 Adaptation strategies

Adaptable enforcement of assertions enables their application on realistic data sets, both in terms of size and values. Due to the very precise, compute-intensive nature of science and engineering applications, they are more susceptible to overflow, underflow, and round-off errors than most IT applications [12, 18, 20]. The aggregation of round-off errors over the life of an iterative computation that can take days, weeks, or months to run can result in a tremendous waste of time and compute resources. While pinpointing the exact time where the computation started to fail would be ideal, the ability to detect it in the middle of a computation could save developers hours of debugging time.

As a proof of concept, we have implemented two policies for rudimentary adaptable assertion checking: linear and random. The linear approach employs a counter-based sampling technique similar to that used by Arnold *et al.* [2] for the collection of profiling information. The random approach uses a statistical sampling technique similar to the one used by Liblit *et al.* [13] for checking assertions at the programming language level. Both approaches rely on a user-specified sampling value to determine the sampling frequency or density, respectively.

Regardless of the policy, assertions are checked on the initial call to each method. The basic assumption being that it is best to catch problems using the interface on the first call. The assertions are checked again on each subsequent call to a method that follows an assertion violation until no violations are detected. Hence, if there are no assertion violations for a method, the frequency of checking follows that specified by the policy.

This simplistic approach to adaptation has provided some of the infrastructure to enable us to pursue the more advanced adaptation policies introduced in Section 1. In particular, we are in the design phase of developing adaptation policies to take advantage of the fact that not all assertions are equal in terms of cost or importance. We will begin by integrating a user-specified importance factor into the SIDL grammar as well as modifying our framework to obtain and utilize rules for distinguishing between cheap and expensive assertions and for progressing through hierarchies of checks.

5. Experimentation plan

In order to address the two major concerns of efficiency and effectiveness, we will perform experiments based on the TSTT interface standard [19]. Several existing mesh smoothing clients and a finite element method client will be used over two different implementations of the interface. Although the TSTT interface standard is still in flux; using it enables contract reuse. As a result, additional experimentation opportunities will arise in the near future as other implementations that have or are in the process of conforming to the standard become available.

Two forms of additional instrumentation will obviously have to be added in order to obtain the necessary analysis data. To measure the performance impact, timing information will need to be collected. Fault injection will be used to randomly perturb the data associated with the assertion violations based on known fault models in related numerical computations. Information on injection and violation detection incidents will be collected for analysis of the effectiveness of our adaptable assertion enforcement approach.

6. Related Work

The software engineering literature has many examples of efforts exploring facets of ensuring component correctness. Similarly, the scientific computing community has explored mechanisms for ensuring the correctness of computational results. Due to space constraints, this section focuses on related works in the area of general run-time assertion checking with an emphasis on those based on high-level specifications.

There are two basic characteristics that differentiate run-time assertion checking efforts reported in the literature. In particular, the efforts are distinguished by

their method of integration and level of enforcement. The integration of assertion checking into an application can be instrumented directly in code, in wrappers, or through external monitors. Similarly, the level of enforcement can be an all-or-nothing approach, sometimes with the option of partial enforcement such as all preconditions only or all postconditions only, or through a reduced checking approach. The remainder of this section briefly describes related efforts within this context.

In order to ensure safe adaptation, the SAMcode model of adaptable mobile agents [1] includes the specification of adaptable procedure and method assertions. The specifications allow one precondition and one postcondition associated with each adaptable method so they appear to follow an all-or-nothing enforcement strategy, though adaptable procedures can be used for assertions. Similarly, Feather *et al.* [9] adapt their requirements constraint checking (through run-time event monitoring) in order to account for runtime behavioral deviations and changing environmental conditions. They consider constraints as soft goals, some of which can be specified by the user as "breakable assertions". This actually corresponds to our notion of useful assertions. Unlike these efforts, our work provides automated adaptation of the frequency of checking itself.

Liblit *et al.*'s [13] statistical assertion checking inspired our work, especially in terms of implementation efficiency (i.e., fast versus slow paths) and reduced frequency checking. However, we are integrating our checks into language interoperability code to maximize contract reuse; whereas theirs is instrumented in the software itself. Additionally, each of their assertions has equal probability of being checked on a given execution while all assertions for a method are currently treated as a single entity with the same probability in our work.

There are a number of high-level specification efforts that map constraints, or contracts, into executable code. The Architectural Specification Language (ASL) [3] is mapped into OMG IDL. Hamie [10] added assertions to the Object Constraint Language (OCL), which is used for modeling the design of software, and integrated them into specifications for C++ and Java. Similarly, Verheecke and Van Der Straeten [21] developed a framework that translates OCL into executable constraints though their approach employs constraint classes. Edwards *et al.* [8] also uses the automatic generation of instrumented wrappers from specifications. Heineman [11], however, employs a Run-time Interface Specification Checker (RISC) for contract enforcement. None of these efforts, however, utilize adaptation to address reduced frequency checking.

7. Conclusions

When developing software for use by others, long-term success mandates that the code attain an acceptable level of quality. One technique for ensuring the quality of software involves the use of assertions. Full assertion enforcement is too costly for most applications, so we propose the use of adaptive assertion checking as a new approach to run-time enforcement in deployed components.

Adaptable enforcement of assertions facilitates debugging software that would otherwise run without assertion checking enabled. Using this approach has the potential of saving developers hours, even days, of debugging, especially for long-running applications that depend upon iterative operations.

We have implemented two rudimentary adaptation policies and are in the process of designing experiments to test their performance and efficiency. We are also in the preliminary design phase of developing more advanced policies that factor in the cost and importance of individual assertions.

8. Acknowledgements

Our appreciation goes to Tom Epperly and Gary Kumfert for their support of the modifications to SIDL and Babel. We would also like to thank Bjarne Stroustrup for his suggestions regarding assertion importance and user-defined violation handling. This effort was funded under the auspices of the U.S. Department of Energy's Center for Component Technology for Terascale Simulation Software (CCTTSS) of the Scientific Discovery through Advanced Computing (SciDAC) program.

Funding for the examples being leveraged in our study was provided by the TSTT SciDAC. We want to thank Kyle Chand for his suggestion to implement the mesh smoothing examples as part of a TSTT interface performance study and for helping debug aspects of their use of the Overture implementation. Thanks also go to Lori Diachin for her finite element methods example.

Our thanks also go to those who reviewed and provided feedback on the initial draft of this paper. In particular, we'd like to thank Tom Epperly, Alan Laub, and Bronis de Supinski.

This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

9. References

[1] N. Amano and T. Watanabe, "A Software Model for Flexible and Safe Adaptation of Mobile Code Programs", In *Proceedings of the International Workshop on Principles of Software Evolution*, Orlando, FL, May 2002, pp. 57-61.

[2] M. Arnold and B.G. Ryder, "A Framework for Reducing the Cost of Instrumented Code", *ACM SIGPLAN Notices,* V. 36, May 2001, pp. 168-179.

[3] F. Bronsard, D. Bryan, D., W.(V.) Kozaczynski, E.S. Liongosari, J.Q. Ning, A. Olafsson, and J.W. Wetterstrand, "Toward Software Plug-and-Play", In *Proceedings of 1997 Symposium on Software Reusability (SSR '97),* Boston, MA, May 17-20, 1997, pp. 19-29.

[4] D. Brown, L. Freitag, and J. Glimm, "Creating Interoperable Meshing and Discretization Technology: The Terascale Simulation Tools and Technologies Centre", In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computation Field Simulations*, Honolulu, HI, June 3-6, 2002. Also available as Technical Report UCRL-PRES-151494, Lawrence Livermore National Laboratory, Livermore, CA, 2002.

[5] Components Project, www.llnl.gov/CASC/components/, 2004.

[6] T. Dahlgren, T. Epperly, and G. Kumfert, "Babel User's Guide", Technical Report UCRL-MA-145991, Lawrence Livermore National Laboratory, Livermore, CA, 2004.

[7] T. Dahlgren, T. Epperly, and G. Kumfert, "Babel Tutorial - Introduction to Babel Technologies," Technical Report UCRL-PRES-200001, August 2003.

[8] S. H. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth, "A Framework for Detecting Interface Violations in Component-Based Software", In *Proceedings of the 5th International Conference on Software Reuse*, June 2-5, 1998, pp. 46-55.

[9] M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior", In *Proceedings of the 9th International Workshop on Software Specification and Design*, April 1998, pp. 50-59.

[10] A. Hamie, "Enhancing the Object Constraint Language for More Expressive Specifications", In *Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC '99)* Takamatsu, Japan, December 7-10, 1999, 376-383.

[11] G.T. Heineman, "Integrating Interface Assertion Checkers into Component Models", In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, OR, May 3-4, 2003.

[12] T.E. Hull, M.S. Cohen, J.T.M. Sawchuk, and D.B. Wortman, "Exception Handling in Scientific Computing", *ACM Transactions on Mathematical Software*, V. 14, N. 3, September 1988, pp. 201-217.

[13] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling", In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 9-11, 2003, pp. 141-154.

[14] B. Meyer, *Object-Oriented Software Construction, 2nd Edition,* Prentice Hall, Upper Saddle River, NJ, 1997.

[15] B. Meyer, "The Grand Challenge of Trusted Components", In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, Portland, OR, May 3-10, 2003, pp. 660-667.

[16] Object Management Group, "CORBA Basics", www.omg.org/gettingstarted/corbafaq.htm, 2004.

[17] Object Management Group, "OMG IDL: Details", www.omg.org/gettingstarted/omg_idl.htm, 2004.

[18] P. Prata and J.G. Silva, "Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations", In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, June 15-18, 1999, pp. 4-11.

[19] Terascale Simulation Tools and Technologies Specification (V. 0.5.1), www.tstt-scidac.org/software/TSTT.sidl, 2004.

[20] M. Turmon, R. Granat, D.S. Katz, and J.Z. Lou, "Tests and Tolerances for High-Performance Software-Implemented Fault Detection", *IEEE Transactions on Computers*, V. 52, N. 5, May 2003, pp. 579-591.

[21] B. Verheecke and R. Van Der Straeten, "Specifying and Implementing the Operational Use of Constraints in Object-Oriented Applications", In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (Tools Pacific 2002)*, Sydney, Australia, February 2002, pp. 23-32.