Gaining Confidence in Scientific Applications Through Executable Interface Contracts

Tamara Dahlgren,[1] David Bernholdt,[2] Lois Curfman McInnes[3]

[1] Lawrence Livermore National Laboratory, Livermore, CA
[2] Computer Science and Mathematics, Oak Ridge National Laboratory, Oak Ridge, TN
[3] Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL

E-mail: dahlgren1@llnl.gov, bernholdtde@ornl.gov, mcinnes@mcs.anl.gov

Abstract. Interface contract enforcement is intended to help scientists gain confidence in software built from third-party components. Unfamiliar components present increased risk of incorrect or unanticipated usage patterns and unexpected component behavior. Executable interface contracts can address these issues but may incur unacceptable overhead. Research into techniques for performance-driven contract enforcement pursues practical solutions to adapting the level of contract enforcement to performance constraints.

1. Introduction

Many aspects of modern high-performance scientific computing challenge the levels of confidence scientists and engineers have in software. Scientific applications continue to grow ever larger and more complex due to the desire to simulate more complex phenomena through higher fidelity models, evolving high-end computer architectures, and new algorithms to effectively utilize these systems. These goals are of particular concern to applications under development by SciDAC teams working in combustion [1], quantum chemistry [2], core-edge fusion modeling [3], and accelerator modeling [4]. Executable interface contracts — in the form of concise, human-readable, machine-processable specifications of behavioral constraints — can be used to gain confidence in applications built from components.

Verification and *validation* (V&V) are terms for the general processes used to gain confidence in software. Developers *verify* that their software conforms to its specifications — in terms of performing its computations. Researchers *validate* results from their computational models against appropriate external results through analytical models, experiments, and/or observations. Scientific computing software environments traditionally provide few facilities for aiding the V&V process, leading developers to build and rely on sets of tests. Thorough test suites can be quite challenging and time consuming to develop. In practice, errors often arise because developers are not able to anticipate every possible usage scenario and input data set.

Interface contracts offer a complementary approach to software verification. Current practice in programming defines only the *syntax* of the interface through specifications of methods and their parameters. Interface contracts provide a means of specifying some of the semantics associated with component interfaces — in terms of the behaviors associated with the methods. Contracts are formed by precondition, postcondition, and/or class invariant clauses belonging to the interface specification, not the underlying implementation(s). *Precondition* clauses consist

of assertions on properties that must hold prior to method execution. *Postcondition* clauses contain assertions that must hold upon method completion. *Class invariants* apply before and after execution of all defined methods. Because this approach specifies properties of the interface itself in a form amenable to automated enforcement, it allows *every* use of the interface to be treated as a test case, thereby enabling dynamic validation of program behavior under actual operating conditions. The result is improved testing, debugging, and runtime monitoring of software quality, thereby providing scientific software developers with a powerful tool for catching errors early and ensuring correct software usage during deployment.

Several techniques developed in the community focus on verifying computational results [5, 6, 7]. The most prominent approaches are *Algorithm-Based Fault Tolerance* (ABFT) and *Result Checking* (RC). ABFT can take the form of any one of a number of techniques taking advantage of the underlying mathematical computations to accumulate information about the correctness of the results. One example is the use of a check sum. RC, on the other hand, is akin to postconditions. It can involve a similar *a posteriori* computation used to compare with results. Although both ABFT and RC are useful for gaining confidence in computational results, they can be expensive and do not address the more general issue of using software correctly.

The enforcement of interface contracts also results in computational overhead. Verifying that a scalar parameter is non-zero is a fast operation, but confirming a vector or matrix has a non-negative norm scales with the size of the data. If the overhead of contract checking is too high, enforcement will be limited to selected test cases. However, the greatest chance of exposing errors and failures comes when the code is being used in "extreme" situations, such as in the largest-scale simulation runs. These are also the conditions under which users tend to be most sensitive to performance overheads. Standard practice in communities adopting contracts is to eliminate all checks or manually disable at least the more complex or expensive ones. Consequences of completely disabling contract enforcement range from spending days to weeks reproducing and debugging errors to making decisions or reporting findings based on erroneous information.

This research was undertaken as part of the SciDAC Center for Technology for Advanced Scientific Component Software (TASCS) [8] as part of the Common Component Architecture (CCA) [9] Forum's pursuit of scientific software component technologies. Components are encapsulated units of software interacting only through well-defined interfaces. The specification of contracts is a first step in providing automated tools utilizing semantic information to increase confidence in the software and resulting scientific data.

This paper highlights some of the work and findings to date. Results are presented from experiments primarily using mesh components developed by the ITAPS [10] SciDAC project, which is providing meshing and discretization tools for a variety of SciDAC applications. Mesh management is a critical part of many scientific applications so the benefits of interface contracts on mesh components can help improve the quality of many applications.

2. Interface Contracts in Babel

One of the key reasons executable interface contracts are not ubiquitous in this domain is the inability to define them in common scientific programming languages. This research aims to make contracts broadly available through the Babel language interoperability toolkit [11]. Babel provides middleware tailored for high-performance scientific programming for the following languages: C, C++, Fortran, Java, and Python. Support for the specification and enforcement of interface contracts has been added to the Babel toolkit. Contracts are specified through the addition of clauses to the interfaces defined in the Scientific Interface Definition Language (SIDL). Statements to enforce contracts are automatically generated by the Babel compiler along with the language interoperability middleware. This approach allows interface contracts to be defined once in the specification and applied to all associated implementations in any of

```
double norm(in double tol)
  throws     /* Exceptions */
    sidl.PreViolation, NegativeValueException, sidl.PostViolation;

  require    /* Precondition clause */
    non_negative_tolerance: tol >= 0.0;

  ensure     /* Postcondition clause */
    non_negative_result: result >= 0.0;
    nearEqual(result, 0.0, tol) iff isZero(tol);
```

Figure 1. Vector norm contracts example.

the supported languages.

Figure 1 shows the SIDL specification of a *norm* method on a vector of doubles. In this case, `norm` is a function taking `tol` as an argument and returning a double `result`. The precondition clause consists of an assertion requiring the caller to pass a non-negative tolerance, `tol`, value. If the preconditions are satisfied, all implementations of the method must then ensure the return of a non-negative `result` within the specified tolerance of zero if-and-only-if the values of all elements in the vector are zero (i.e., it is the *zero* vector).

Contract enforcement is automatically managed by the Babel runtime library. Built-in, contract clause-specific exceptions are raised whenever a violation is detected. Since SIDL and Babel are central to the widely-used reference implementation of the CCA, interface contracts implemented in this manner can easily be used with CCA components.

3. Experiments

Research thus far focuses on identifying and developing interface contract enforcement sampling techniques driven by execution time overhead limits [12, 13, 14, 15]. Experiments are conducted to investigate the associated impacts. Data are collected to establish metrics for comparing enforcement strategies. This section summarizes basic strategies and metrics used in this research. Selected results and an overview of findings are also presented. More information and details can be found in [14, 15].

3.1. Enforcement Policies

Interface contract enforcement policies reflect different strategies for checking contract clauses. Although this research involves the implementation of many policies, this paper focuses on three: *Never*, *Always*, and *Adaptive fit*. The first two policies — *Never* and *Always* — represent "all-or-nothing" strategies providing baseline data. The *Never* policy disables interface contract enforcement. The resulting execution times are used to establish enforcement overhead metrics. The *Always* policy enables full contract enforcement. That is, all interface contracts encountered during execution are checked. Consequently, the policy provides baselines for the total number of checked (or enforced) contracts and detected violations for each trial. *Adaptive fit* is one of three performance-driven sampling approaches. The policy checks contract clauses only if the accumulated enforcement execution time estimate is within the user-specified limit of the cumulative estimates outside the contracts.

3.2. Metrics

Three metrics are used to compare the effects of policies under study: enforcement overhead, contracts enforced, and violations detected. **Enforcement overhead** is the percentage

Table 1. Summary of data collected in the latest study [14]. Experiments did not detect any interface contract violations for the program labeled **MA**.

Program	Software Component (Source)	Policy	Mean (over all trials)		
			Enforcement Overhead	Contracts Enforced	Violations Detected
A	Simplicial Mesh (ITAPS)	*Always*	11%	100%	100%
		Adaptive fit	17%	65%	67%
AA	Simplicial Mesh (ITAPS)	*Always*	35%	100%	100%
		Adaptive fit	14%	58%	67%
MA	Simplicial Mesh (ITAPS)	*Always*	27%	100%	0%
		Adaptive fit	11%	0.27%	0%
MT	Volume Mesh (ITAPS)	*Always*	3%	100%	100%
		Adaptive fit	2%	0.35%	6%
VT	Vector (Babel)	*Always*	9%	100%	100%
		Adaptive fit	0%	59%	64%

difference in the average execution time of a policy above the cost of conducting the experiment without contract checking. That is, the overhead of a given policy is relative to the average execution time for experiments using the *Never* policy. **Contracts enforced** is computed based on the number checked for a policy relative to the total number checked using the *Always* policy. **Violations detected** for a policy is also relative to the total number of violations detected with the *Always* policy. Hence, the effects of sampling contracts are considered in terms of execution time, contract checks, and contract clause violations.

3.3. Findings

Table 1 summarizes data obtained from experiments using the policies described in Section 3.1. Experiments involve two mesh components from the ITAPS [10] SciDAC project and a vector component from Babel [11]. Details, based on results by trial, can be found in [14]. Performance-driven enforcement automatically adjusts the level of contract clause checking and tends to have better overall control of enforcement overhead than alternatives strategies considered in the study. As illustrated by program **A**, performance-driven enforcement is inappropriate for programs exercising contracts composed solely of scalar checks. However, performance-driven enforcement appears to be most suited to programs enforcing contract clauses whose checking is fast-to-moderately expensive relative to the amount of time spent in component methods.

Analysis of data across three studies [15] further indicates the quality of execution time estimates impacts checking and overhead. While coarse estimates enable performance-driven policies to keep the overhead closer to the target level, they preclude the detection of violations in trials of the first study. The most refined estimates in the final study enable performance-driven policies to detect significant numbers of violations in roughly half of the trials.

4. Summary

The goal of this research is to help scientists gain confidence in software built from third-party components using specialized interface contract enforcement strategies. The vision of scientists developing insights into and predictions about physical phenomena through component technologies is based on the idea of a repository containing multiple components conforming to the same interface specification. Behavioral specifications in the form of executable interface contracts are a well-known mechanism for ensuring compliance at runtime. However,

performance overhead concerns can be roadblocks to their adoption.

This research proposes the use and investigates the effects of performance-driven sampling as a means of controlling the impact of interface contract enforcement on program execution time. Target applications are those making numerous calls to methods with contracts that are also expected to incur unacceptable overhead from enforcing all contracts. Experiments leveraging two ITAPS mesh components and a Babel vector component are described. Findings indicate performance-driven interface contract enforcement is a viable alternative to the traditional strategy of disabling enforcement for applications exhibiting target characteristics. Programs executing methods whose contracts contain assertions which are fast-to-moderate to check relative to the time spent executing their methods appear to benefit most from performance-driven strategies.

Future work includes further testing and enhancement of these capabilities in SciDAC applications, as well as incorporating these techniques into tools under development by the TASCS project for computational quality of service (CQoS) [16], or the automatic selection and configuration of components to suit a particular computational purpose and environment.

Acknowledgments

We thank those who contributed software and/or interface contracts to the studies: Lori Freitag Diachin (simplicial mesh), Carl Ollivier-Gooch (volume mesh, program **MT**), and Kyle Chand. This work was performed under the auspices of the U. S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program's Center for Technology for Advanced Scientific Component Software (TASCS). Lawrence Livermore National Laboratory is managed by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344. Oak Ridge National Laboratory is managed by UT-Battelle, LLC under Contract DE-AC-05-00OR22725. Argonne National Laboratory operates under Contract DE-AC02-06CH11357.

References

[1] Najm H (PI) Computational Facility for Reacting Flow Science (CFRFS) `http://cfrfs.ca.sandia.gov`
[2] Gordon M (PI) Chemistry Framework using the CCA `http://www.scidac.gov/matchem/better.html`
[3] Cary J (PI) Framework Application for Core-Edge Transport Simulations `http://www.facetsproject.org`
[4] Spentzouris P (PI) Community Petascale Project for Accelerator Science and Simulation (COMPASS) DOE SciDAC project, 2007
[5] Hull T E, Cohen M S, Sawchuk J T M and Wortman D B 1988 *ACM Transactions on Mathematical Software* **14** 201–217
[6] Prata P and Silva J G 1999 *Proc. Intl. Symp. on Fault-Tolerant Computing* pp 4–11
[7] Turmon M, Granat R, Katz D S and Lou J Z 2003 *IEEE Transactions on Computers* **52** 579–591
[8] Center for technology for advanced scientific component software (tascs) `http://tascs-scidac.org`
[9] Allan B A, Armstrong R, Bernholdt D E, Bertrand F, Chiu K, Dahlgren T L, Damevski K, Elwasif W R, Epperly T G W, Govindaraju M, Katz D S, Kohl J A, Krishnan M, Kumfert G, Larson J W, Lefantzi S, Lewis M J, Malony A D, McInnes L C, Nieplocha J, Norris B, Parker S G, Ray J, Shende S, Windus T L and Zhou S 2006 *Intl. J. High-Perf. Computing Appl.* **20** 163–202 URL `http://hpc.sagepub.com/cgi/reprint/20/2/163`
[10] Diachin L (PI) Center for Interoperable Technologies for Advanced Petascale Simulations (ITAPS) `http://www.scidac.gov/math/ITAPS.html`
[11] Kumfert G (PI) Babel `http://www.llnl.gov/CASC/components/babel.html`
[12] Dahlgren T L and Devanbu P T 2004 *Proc. Intl. Workshop on Software Engineering for High Perf. Computing System Appl.* (Edinburgh, Scotland) pp 64–69
[13] Dahlgren T L and Devanbu P T 2005 *Proc. 2nd Intl. Workshop on Software Engineering for High Perf. Computing System Appl.* (St. Louis, Missouri) pp 73–77
[14] Dahlgren T L 2007 *Proc. Intl. Symp. on Component-Based Software Engineering* (Medford, MA USA)
[15] Dahlgren T L 2008 *Performance-Driven Interface Contract Enforcement for Scientific Components* Ph.D. thesis University of California, Davis One Shields Avenue, Davis, CA, 95616
[16] Norris B, Ray J, Armstrong R, McInnes L C, Bernholdt D E, Elwasif W R, Malony A D and Shende S 2004 *Proc. Intl. Symp. on Component-Based Software Engineering* (Edinburgh, Scotland)