UCRL-CONF-228332

LAWRENCE LIVERMORE NATIONAL LABORATORY

Performance-Driven Interface Contract Enforcement for Scientific Components

T. Dahlgren

February 26, 2007

The 10th International Symposium on Component-Based Software Engineering
Medford, MA, United States
July 9, 2007 through July 11, 2007

Performance-Driven Interface Contract Enforcement for Scientific Components

Tamara L. Dahlgren

Lawrence Livermore National Laboratory, Livermore, CA 94550 USA
dahlgren1@llnl.gov [⋆]

Abstract. Several performance-driven approaches to selectively enforce interface contracts for scientific components are investigated. The goal is to facilitate debugging deployed applications built from plug-and-play components while keeping the cost of enforcement within acceptable overhead limits.
This paper describes a study of global enforcement using *a priori* execution cost estimates obtained from traces. Thirteen trials are formed from five, single-component programs. Enforcement experiments conducted using twenty-three enforcement policies are used to determine the nature of exercised contracts and the impact of a variety of sampling strategies. Performance-driven enforcement appears to be best suited to programs that exercise moderately expensive contracts.

1 Introduction

Selective, performance-driven interface contract enforcement is intended to help scientists gain confidence in software built from plug-and-play components while retaining their code's high performance. This work is a natural extension of decades of research in component technology and software quality. For the purposes of this work, a *component* is defined as an independent software unit with an interface specification describing how it should be used [3]. Hence, caller and callee are loosely coupled through the callee's interfaces. Thus, logical feature groupings within existing scientific libraries can be wrapped as components.

Interchangeable components based on varying characteristics such as the underlying model, precision, and reliability were key features of the vision published in McIlroy's 1968 seminal paper on software components [4]. Grassroots efforts were begun in the late 1990's by the Common Component Architecture (CCA) Forum [5–8] to bring component-based software engineering to the high-performance scientific computing community. At present, eleven institutions — consisting of national laboratories, universities, and research-based companies —

[⋆] This research was funded under the auspices of the U.S. Department of Energy's Center for Technology for Advanced Scientific Component Software (TASCS) [1] of the Scientific Discovery through Advanced Computing (SciDAC) [2] program by the University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. UCRL-CONF-228332 are actively involved in collaborative efforts to further facets of the organization's goals. This research is part of the CCA's software quality initiative.

The Institute of Electrical and Electronics Engineers (IEEE) [9] defines *quality* as "the degree to which a system, component, or process meets" its specifications, needs, or expectations. Interface contracts are specifications that take the form of preconditions, postconditions, and or invariants that belong to the interface, not the the underlying implementation(s). *Preconditions* are assertions on properties that must hold prior to method execution. *Postconditions* are assertions that must hold upon method completion. *Invariants* apply before and after method execution. Hence, interface contracts are specifications that are amenable to automated enforcement.

Interface contracts are related to a practice traditionally referred to as "defensive programming". Conscientious developers have long relied on assertions at the top of their routines to protect their software from bad inputs. The basic intent is to catch potential input-related problems before they cause the program to unexpectedly crash. These kinds of checks should always be retained since contracts may not be enforced during deployment. However, interface contracts are broader in scope since they can include constraints on other properties of the input as well as properties of the output, the component, and its state.

Since scientific components are developed by people with different backgrounds and training, it is not safe to assume that everyone uses the same level of rigor in their software development practices — especially in the case of research software. This fact does not preclude the potential advantages for scientists to experiment with different research components providing similar computational services. Defining those services with a common interface specification facilitates the use of different implementations. Executable interface contracts then provide some assurances that interface failures can be caught regardless of the programming discipline used by component implementors.

However, the community's performance concerns could become a roadblock to the adoption of contract enforcement during deployment; hence, the pursuit of performance-driven enforcement policies. Section 2 describes the trade-offs faced by the community. The enforcement infrastructure is summarized in Section 3. Section 4 elaborates on the methodology and subjects used in the study before highlighting key findings. An overview of the most relevant related works is given in Section 5 before the summary of future work in Section 6.

2 Motivation

There is growing interest in leveraging component-based software engineering (CBSE) for the re-use of legacy software as plug-and-play components in multi-scale, multi-physics models. The resulting complexity of these applications — especially when components are implemented in different programming languages — makes testing and debugging difficult. The ability to swap components at runtime increases debugging challenges. At the same time, developers of scientific applications are very concerned with the performance implications of new technologies since computational scientists are typically willing to incur no more than ten percent additional overhead. Effectively balancing these competing demands is a significant challenge.

Applications composed in a plug-and-play manner depend on components implemented and wrapped in accordance with claimed services. However, when using unfamiliar components, there is increased risk of incorrect or unanticipated usage patterns. Furthermore, such applications have the potential of relying on input data set combinations that lead to unexpected component behavior.

Interface contracts can provide clear documentation of service constraints. When specified in an implementation-neutral language, contracts can also serve as a basis for the consistent instrumentation and enforcement of interface constraints, regardless of the underlying implementation language. Pinpointing the exact statement or module in which the computation failed would be ideal; however, the ability to detect violations in the middle of execution can still save many hours to weeks of debugging.

While interface contracts can facilitate testing and debugging applications built of components, contract enforcement is generally perceived as too expensive for deployment. This may be an extension of the idea that programming language-level assertions can have a negative impact on performance. Intuitively, assertions in frequently executed code and tight loops are most likely to be too costly. Consequently, standard practice — specifically in domains and projects that rely on assertions — involves eliminating all checks or disabling at least the more complex or expensive ones. The result, however, is exposing software to unchecked violations. Risks range from spending days to weeks reproducing and debugging errors to making decisions or reporting findings based on erroneous information.

With the growing interest in CBSE for building multi-scale, multi-physics models from legacy software comes the challenge of providing mechanisms to facilitate debugging with minimal performance impact. Hence, this research pursues a compromise solution of performance-driven enforcement within a user-specified overhead tolerance. The basic idea is to throttle enforcement at runtime if and when the limit is reached.

3 Enforcement Infrastructure

The Babel [10] toolkit developed at Lawrence Livermore National Laboratory forms the basis for the enforcement infrastructure. Specifications in the Scientific Interface Definition Language (SIDL) are automatically translated into language interoperability middleware using the Babel compiler. Contracts are supported through optional SIDL annotations, which are mapped to runtime checks embedded in the middleware. An example of an annotated SIDL specification for the vector *norm* method is given below. The remainder of this section describes the toolkit with an emphasis on changes since preliminary investigations [11, 12].

```
package vector version 1.0 {
    class Utils {
                        ⋮
      static double norm(in array⟨double⟩ u,
                              in double tol, in int badLevel)
      throws      /* Exceptions */
        sidl.PreViolation, NegativeValueException, sidl.PostViolation;

      require      /* Preconditions */
        not_null: u != null;
        u_is_1d: dimen(u) == 1;
        non_negative_tolerance: tol ≤ 0.0;

      ensure      /* Postconditions */
        non_negative_result: result ≥ 0.0;
        nearEqual(result, 0.0, tol) iff isZero(u, tol);
                        ⋮
    }
}
```

Enforcement decisions are centralized in the experimental Babel toolkit to better control overhead across multiple components. In addition, decisions are made on a finer basis by grouping contracts by locality. For example, the three expressions in the *norm* method's preconditions are treated as a single group while the two postcondition expressions form a second group. Splitting contracts in this manner allows for a wider variety of enforcement options. Previously [11, 12] only three options were supported: *Periodic*, *Random*, and *Adaptive Timing (AT)*. The first two classic strategies were compared to *AT*, which sought to limit the overhead of contract enforcement using runtime timing instrumentation.

Enforcement policies are now based on two parameters: enforcement frequency and contract type. Enforcement frequency determines how often contracts are checked. Contract types further constrain checks to classes of contracts, thereby providing a mechanism for measuring the properties of contracts actually exercised during program execution.

Enforcement frequency can be one of: *Never*, *Always*, *Periodic*, *Random*, *Adaptive Fit (AF)*, *Adaptive Timing (AT)*, or *Simulated Annealing (SA)*. With *Never*, the middleware completely by-passes the enforcement instrumentation. Hence, the software operates as if contracts had never been added to the specification. All contracts (of the specified type) are enforced with the *Always* option. *Periodic* and *Random* support the classic sampling strategies. Enforcement decisions for the remaining three options are based on estimated execution times of methods and their associated contracts. *AF* enforces contracts only if their estimated time will not result in exceeding the user's limit on the cumulative total of program and method cost estimates. *AT* enforces contracts when their estimated times will not exceed the user-specified overhead limit for the method. Finally, *SA* operates like *AF* but allows the overhead to exceed the user-specified limit with decreasing probability over time. Hence, a total of seven enforcement frequencies are supported, three of which are performance-driven.

Contract types can be one of: *All*, *Constant*, *Linear*, *Method Calls*, *Simple Expressions*, *Preconditions*, *Postconditions*, *Invariants*, *Preconditions-Postconditions*, *Preconditions-Invariants*, *Postconditions-Invariants*, and *Results*. All types are checked at the specified frequency with the *All* option. For historical and built-in assertion function reasons, complexity options are currently limited to *Constant*- and *Linear*-time, where contracts for a method are considered to be at the level of the highest complexity assertion expression. The *Method Calls* option enforces only contracts containing at least one method call — built-in or user-defined — while the *Simple Expressions* option is used for contracts wherein no method calls appear. *Preconditions*, *Postconditions*, *Invariants*, and their combinations enforce contracts conforming to those classical distinctions. Finally, *Results* enforces (postcondition) contracts only when at least one expression contains a result or output argument. When combined with *Always*, statistics using these options can serve as baselines for performance-driven counterparts.

Another new feature is enforcement tracing. When enabled by the program, special instrumentation in the middleware determines the amount of time spent in the program, enforcing preconditions, enforcing invariants before the method call, executing the annotated method, enforcing its postconditions, and enforcing invariants after the method call. The resulting timing data is currently dumped to a file after each invocation before control is returned to the caller. Hence, trace results provide the basis for *a priori* execution cost estimates needed for performance-driven enforcement.

The experimental version of the Babel toolkit automatically translates contract annotations in the SIDL specification into runtime checks embedded in the generated language interoperability middleware. During program execution, enforcement decisions are made globally using the chosen frequency and contract type options that form the enforcement policy. One of seven frequency options — including *Never* and three performance-driven strategies — together with one of twelve contract type options can be active at a time. For simplicity, when any frequency option is combined with *All* contract types, "*All*" is dropped from the name.

4 Experiments

Experiments are conducted on a total of thirteen trials formed from five, single-component programs. Enforcement traces are produced to obtain program, method, and contract execution times for use in enforcement experiments. A variety of sampling strategies are employed for each trial in order to study and compare their effects. Analysis of experiment results reveal several interesting patterns. Before presenting results for performance-driven policies, it is useful to consider the impact of full contract enforcement.

4.1 Subjects

Five, single-component programs along with several input array sizes are used as the basis for thirteen trials. Table 1 describes the programs and selected input array sizes. The first four programs rely on components implementing a community-developed mesh interface standard that defines interfaces supporting multiple mesh access patterns. The specification was established by the Terascale Simulation Tools and Technologies (TSTT) Center [13, 14], which is now called the Interoperable Tools for Advanced Petascale Simulation (ITAPS) Center. A single, readily available input file was used with each program. The fifth program is a Babel regression test specifically developed to exercise basic contract enforcement features.

Table 1. Descriptions of the five programs that form the basis for thirteen trials.

	Program	
Component	**Abbrev.**	**Description**
Simplicial Mesh	**MA**	Retrieve all faces from the mesh then, for each face, retrieve the adjacent vertices.
	A	Retrieve all faces from the mesh in sets based on the size of the input array. Sizes 1, 14587 (10%), and 145870 (100%) were used to reproduce the violation and vary processing.
	AA	Retrieve faces in the same manner as program **A** plus, for each set of faces, retrieve their corresponding adjacent vertices. The same input array sizes were used.
Volume Mesh	**MT**	Exercise and check consistency of five mesh interfaces: core mesh capabilities, single entity query and traversal, entity array query and traversal, single entity mesh modification, and entity array mesh modification.
Vector Utilities	**VT**	Exercise all supported functions to include successful execution; one or more precondition violations; and one or more postcondition violations. Sizes 6 (original), 10, 100, 1000, and 10000 were used to vary processing.

Much as one would expect in the real world, the programs involve predominantly constant-time contracts in a variety of settings. Program **A** exercises only constant-time contracts. Varying the input array size in this case corresponds to different amounts of processing within the method and numbers of loop iterations (to vary sampling opportunities). Program **AA** builds on **A** by adding the adjacency retrieval method and its linear-time postconditions to the loop to vary contract processing times as well. (Input array sizes for both programs were selected to induce a violation discovered in previous work [12].) That same method is the only one invoked within program **MA**'s loop. So the three programs vary not only method and contract processing times with constant- and, in two cases, linear-time contracts, they also provide meager to ample sampling opportunities.

The last two programs — **MT** and **VT** — are test programs that serve other purposes in this study. Program **MT** exercises 1,909,129 contracts using a small input file readily available with the GRUMMP [15] software. **VT**, on the other hand, checks 146 contracts every run regardless of the input array size. Since the program builds multiple vectors, varying input array sizes has a significant impact on the amount of execution time attributed to the program. So the two programs expand on the variety of sampling opportunities or execution cost distributions demonstrated with the first three programs.

Hence, different input arrays sizes for several of the five programs were used to define thirteen trials. The trials varied in terms of the amount of work done in the methods and programs. The numbers of sampling opportunities also varied, ranging from six (with **A-145870**) to 1,909,129 (with **MT**). Finally, although linear-time contracts are checked to some degree in nearly all trials, checked contracts are predominantly constant-time.

4.2 Methodology

The experimental process consists of essentially three phases. In the first phase, execution cost estimates are established in order to guide performance-driven enforcement decisions. The second phase involves conducting the actual experiments. Finally, experiment results are analyzed and compared.

Once annotated contracts are translated by Babel into enforcement checks in the middleware and the software re-built, *a priori* execution time estimates are needed for each annotated method and its associated contracts. The enforcement tracing feature, described in Section 3, is used to estimate those costs. Due to the sizes of the corresponding trace files, each trial is executed five times with tracing and full contract enforcement enabled. Mean execution times are then computed from the traces to obtain trial-specific estimates.

Figure 1 illustrates the resulting enforcement trace results. Preconditions dominate contract costs for trials **MA**, **A-1**, and **AA-1**. The total costs of annotated methods far exceed the times attributed to the other categories for trials **A-14587**, **A-145870**, **AA-14587**, and **AA-145870**; however, contract execution times are dominated by postconditions for the latter two as a result of the linear-time contracts. Only 15-20% of the execution times of trials **VT-6**, **VT-10**, and **VT-100** are attributed to contracts where even less time is spent in the methods. Finally, execution times for trials **VT-1000** and **VT-10000** are almost exclusively spent in the programs. Nearly every trial illustrates a different pattern, or execution profile.

Experiments are then performed by executing each trial multiple times using each enforcement policy under consideration. A total of twenty-three different policies were used to gather data for this study. Seven policies combined the *All* contract types option with each of the seven enforcement frequency options to capture frequency-specific data. Eight more policies combined the *Always* option with basic contract types to provide data on the nature of checked contracts.

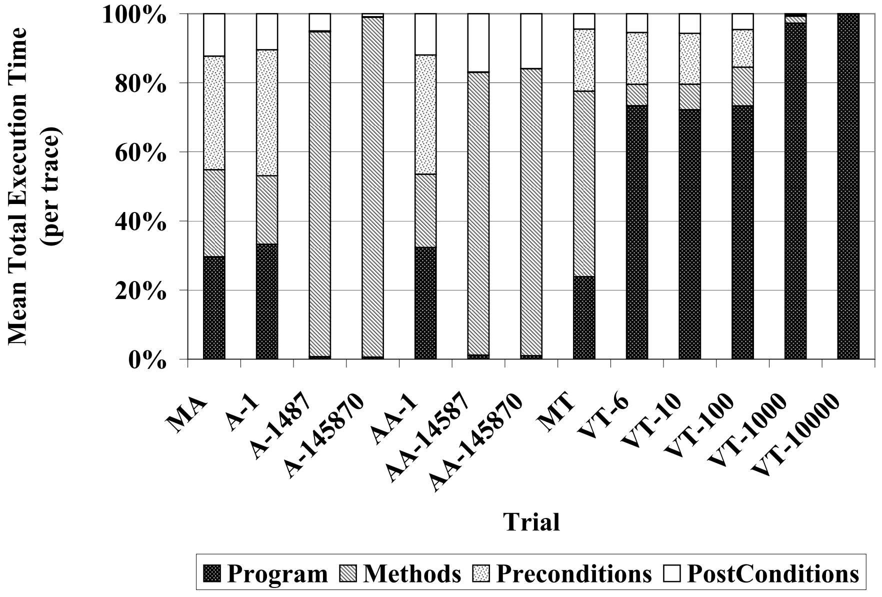

Fig. 1. Trace execution profiles

Finally, the *AF* option was combined with the basic contract types for the last eight policies. The goal was to investigate the impact of performance-driven variants. A 5% overhead limit was used on all performance-driven enforcement policies. Between ten and thirty repetitions of each experiment were performed to mitigate the inherent variations in execution times. Every effort was made to perform experiments, which were executed on a networked machine running Red Hat Linux 7.3, when the machine was lightly loaded.

Baseline data are derived from the results of policies using the *All* types option. Three metrics are computed based on experiment results: enforcement overhead, contract coverage, and violation detection effectiveness. Since the *Never* policy measures the execution time when the instrumentation is by-passed, it serves as the basis for computing overhead. That is, **enforcement overhead** is calculated as the percentage of execution time above that of *Never*. Running trials with the *Always* policy provides baselines for both the total number of contracts checked and total number of detectable violations. **Contract coverage** is then computed as the percentage of the number of checks for a policy versus the number with *Always*. Similarly, **violation detection effectiveness** is the fraction of violations detected with a policy versus with *Always*. Combining the *Always* frequency option with specific contract types provide similar coverage and violation detection baseline metrics for their performance-driven counterparts.

Hence, tracing is used to obtain the program, method, and contract execution cost estimates needed by the middleware to guide performance-driven policies.

Enforcement experiments are run by executing each trial numerous times for each enforcement policy to mitigate the inherent variability in execution times. Baseline metrics are collected using basic policies like *Never* (for overhead) and *Always* (for contract coverage and detected violations).

4.3 Full Enforcement Results

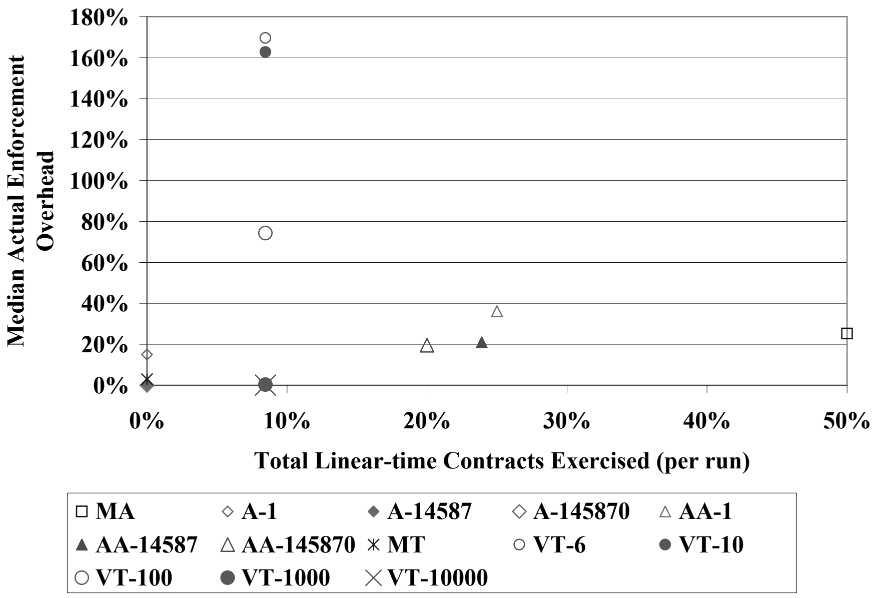

Fig. 2. Median enforcement overhead with *Always* by percentage of linear-time contracts.

Figure 2 illustrates the overhead with *Always* by percentage of linear-time contracts. Trials **A-14587**, **A-145870**, **VT-1000**, and **VT-10000** incur negligible overhead. Contract enforcement opportunities range from 6 to 146 per run with at least 92% being constant-time contracts. Trial **MT** incurs only 3% overhead despite checking significantly more contracts than any other trial. According to results using the *Linear* policy, only 89 of **MT**'s contracts are linear-time. Trials **A-1**, **AA-1**, and **MA** reflect between 291,740 and 583,484 contract checks per run. While the latter two trials include linear-time contracts, the corresponding output arrays contain only a few entries due to their use of single-element input arrays. Furthermore, trial **AA-1** exercises only constant-time contracts in the first method in its loop so the instrumentation overhead on those invocations is not ameliorated. Trials **AA-14587** and **AA-145870** checked 44 and 8 contracts per execution, respectively, with their linear-time contracts working on several times the number of elements as in their input arrays. Even though all trials with program **VT** check the same number of contracts, trials **VT-6**, **VT-10**, and **VT-100** incur significantly more overhead than any other trial. Judging from the data in the trace profiles shown in Figure 1, this may be attributable to the relatively small amount of time spent in the methods.

Trials formed using different input array sizes were deliberately chosen to detect the same violations for each program. Those violations are described in Table 2. Mesh program violations reflect the fact that both the programs and component implementations were initially developed prior to contract definition. Program **VT**, on the other hand, exhibits characteristics of non-compliant programs and implementations as a result of deliberately triggering precondition and postcondition violations.

Table 2. Contract violations detected with *Always*, where the same violations occur regardless of input array size.

Program Abbrev.	**Description**
MA	No contract violations.
A	Final (extra) call returns a null array pointer when no more faces left to retrieve from the mesh. The postcondition (set) is constant-time.
AA	Same violation as in **A**.
MT	Four precondition violations occur in constant-time contracts as a result of the program not pre-allocating two classes of input arrays. The remaining 43 violations, which occur in linear-time postconditions, result from the implementation not properly setting output array values for adjacencies.
VT	A total of 78 violations per run are deliberately triggered with *Always*, where postcondition failures are emulated. In all, 94% of the violations are triggered in constant-time preconditions.

So, with the *Always* policy, only seven of the thirteen trials incur more than 3% overhead. It appears these results can be attributed either to lots of relatively inexpensive contracts in tight loops or to moderately expensive contracts. The trials also illustrate a range of between one and seventy-eight contract violations per run. The numbers of violations are tied to the programs not the trials.

4.4 Performance-Driven Enforcement Results

An analysis of the results indicate performance-driven policies generally perform well relative to *Always* — in terms of performance and detected violations — in 83% of the trials with detectable violations. In half of those trials, at least 43% of the performance-driven policies detect all of the violations with negligible overhead — even in trial **VT-145870** where the overhead of *Always* is 20%.

Unfortunately, only *SA* is able to check more than two contracts in trials **A-1** and **AA-1**. The remainder of this section focuses on results using *AF*, *AT*, and *SA* for the six trials where performance-driven enforcement show an improvement over *Always*.

With only 3% overhead for checking all of **MT**'s contracts, all but *SA* cut the overhead by a third while checking only a small fraction of the contracts and generally detecting no more five of the forty-seven violations. Surprisingly, *AT* detects 94% of the violations while covering only 0.04% of the contracts. Given the algorithm used by the policy, these results indicate those violations occur in relatively cheap contracts.

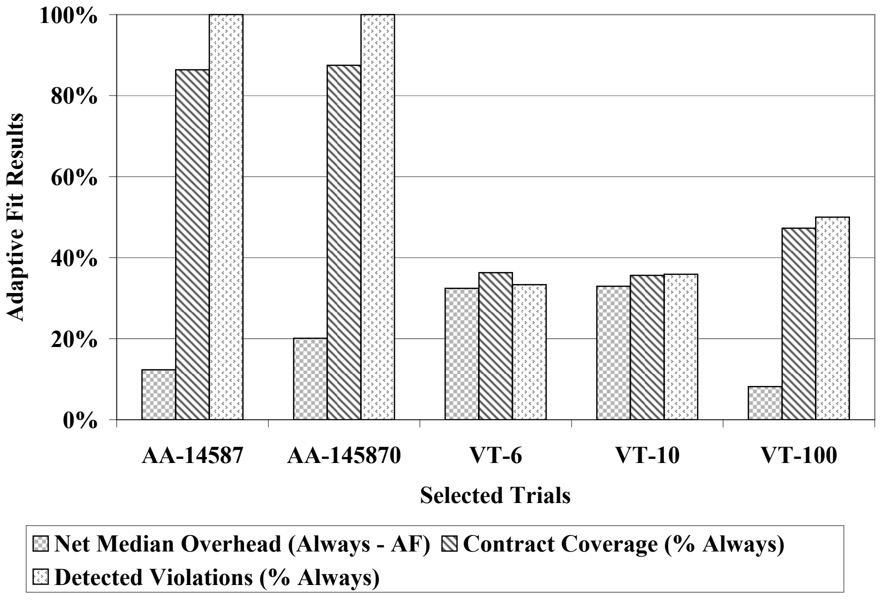

Fig. 3. Enforcement results for *Adaptive Fit (AF)*.

Figure 3 illustrates the results for *AF* on the remaining five trials. The corresponding chart for *SA* is very similar. While *AT* checks fewer contracts for the first two trials shown, it always detects the violation at negligible overhead in those cases. However, it is unable to check any contracts for the last three trials, indicating their execution times exceed 5% of their methods'. This information is useful when considering the results for *AF* and *SA*.

Both *AF* and *SA* cover 86% or more of the contracts for trials **AA-14587** and **AA-145870** and always detect the violation while incurring less than 43% of the overhead of *Always*. The numbers of contracts checked in those trials are forty-four and eight, respectively. So there is less overhead savings for the trial with more contracts.

As shown in Figure 2, the remaining trials incur the most overhead with *Always*. However, the estimated execution time and results of AT indicate all of the contracts exceed 5% of their method's execution times. So it is not surprising that the overhead savings with AF and SA are relatively modest. However, AF is able to detect a third (for **VT-6**) to half (for **VT-100**) of the violations with at least an 8% savings in overhead. SA detects 29% to 36% of the violations in the same trials with 3-4% more savings in overhead in the first two cases. In these cases, the larger the input array size, the lower the overhead and the lower the savings with performance-driven enforcement. However, the larger the input array size, the higher the coverage for a given policy and the more violations it detects.

Three patterns in the results emerged during analysis. First, the instrumentation appears to be too costly for trials that exercise lots of relatively inexpensive contracts; namely, those within tight loops. This is likely attributable to insufficient work performed in the methods to offset those costs. That does not appear to be the case for trials enforcing under two hundred, inexpensive contracts where the overhead is negligible. In general, performance-driven enforcement seems better suited to trials whose traces indicated between 15% and 22% estimated enforcement overhead.

5 Related Work

Associating assertions with software dates to the 1950's [16–18]. Applied researchers recognized the value of executable assertions for testing and debugging in the mid-1970's [19–21]. Meyer's [22] Design-by-Contract methodology was built on this foundation. Component-level extensions of Meyer's work began appearing in the mid-1990's. This section briefly summarizes seven technologies supporting component contracts and three using or proposing sampling of assertions during deployment.

The Architectural Specification Language (ASL) [23, 24] encompasses a family of design languages for CBSE. Its Interface Specification Language (ISL) extends CORBA [25] Interface Definition Language (IDL) with preconditions, postconditions, invariants, and protocol (or states). The Assertion Definition Language (ADL) [26] extends CORBA IDL with postconditions. The goal of ADL is to facilitate formal specification and testing of software components. Hamie [27] advocates extending the Object Constraint Language (OCL), which is a textual language for expressing modeling constraints. He proposes adding invariants to class diagrams. Hamie also proposes adding preconditions, postconditions, and guards to state transition diagrams. The extensions are integrated into specifications for C++ and Java. Similarly, Verheecke and van Der Straeten [28] developed a framework that translates OCL into executable constraints (for Java) using constraint classes. The *ConFract* [29] system adds internal and external composition to the classic contracts, using a rule-based, event-driven approach to runtime verification. The Java Modeling Language (JML) [30] is another example of a language pursuing component contracts — in the form of preconditions and postconditions in comments. Edwards *et al.* [31] also automatically generate wrappers from specifications, with the goal being to separate enforcement from the client and implementation. Their "one-way" wrappers are used to check preconditions. They also have "two-way" wrappers to check preconditions and postconditions, but those are not automatically generated. Heineman [32] employs a Run-time Interface Specification Checker (RISC) for enforcement of preconditions and postconditions.

While the aforementioned technologies pursue component contracts, the most relevant related research efforts identified so far involve sampling assertions. In two efforts, assertions in program bodies are sampled during deployment to reduce enforcement overhead. Liblit *et al.*'s [33, 34] statistical debugging relies on (uniform) random sampling of assertions in remotely deployed applications. This facilitates remote application profiling and debugging of arbitrary code using automated instrumentation. Similarly, Chilimbi and Hauswirth [35] focus on rarely occurring errors but within the context of their SWAT memory leak detection tool. Three pre-defined staleness predicates automatically inserted into program bodies are sampled during deployment. Checking is based on tracing infrequently executed code while frequently executed code is sampled at a very low rate to reduce overhead. The sampling rate starts at 100% but decreases — to a minimum — with each check. Leak reports are then generated from trace files after the program terminates. Collet and Rousseau [36] advocate random sampling limited to universal quantification for recently modified classes and their dependents.

Like the first seven technologies, this work leverages component contracts to improve the quality of software. Programming language-neutral SIDL contracts are automatically instrumented for use by implementations in a variety of languages employed in scientific computing; namely, C, C++, Fortran 77/90, Java, and Python since they are supported by Babel. Using implementations in different programming languages can vary the effects on performance; hence, another motivation for pursuing performance-driven heuristics. Sampling of enforcement decisions is similar to the approach taken by Liblit *et al.* and Chilimbi and Hauswirth. However, while they employ basic sampling strategies, this research advocates automatically tuning the sampling level at runtime based on performance-driven heuristics.

6 Future Work

This research lays a foundation for further investigation of both the nature of interface contracts needed for scientific applications and their impact on performance. Additional studies, involving collaborations with component developers and scientists, should yield insights that can be used to refine the current set of techniques as well as develop others. Better techniques are also needed to improve the accuracy of enforcement decisions. In the meantime, the toolkit is being revised to more readily support multi-component contract enforcement and to integrate these new features into the official Babel source code repository. The work is being done in preparation for conducting a study using small, multi-component example programs as part of a CCA collaboration.

7 Summary

This paper presents results from an investigation of the impact of performance-driven policies supported in an experimental version of the Babel language interoperability toolkit. Enforcement decisions are made on a global basis using *a priori* execution costs obtained from enforcement traces.

Results for five single-component programs are presented based on three logical phases. The first phase involves trace experiments to obtain execution times attributable to programs, invoked methods, preconditions, postconditions, and invariants. Baseline enforcement experiments are then used to obtain execution costs when enforcement is by-passed (with the *Never* policy); total numbers of contracts checked and violations detected during normal contract enforcement (with the *Always* policy); and characteristics of the contracts and violations (with basic contract type options). Finally, the impacts of performance-driven enforcement policies are compared to baselines.

In general, performance-driven policies performed as well or better than *Always* while catching significant numbers of violations in 83% of the trials with violations. Performance-driven policies tended to incur at most a few percent overhead in trials where the overhead of *Always* was negligible. The policies were not able to overcome instrumentation overhead issues in trials representing tight loops. However, the more general-purpose, performance-driven policies were able to detect significant numbers of violations at a saving of at least 8% overhead compared to *Always* in trials involving moderately expensive contracts.

Acknowledgments. Thanks go to those who contributed software and or financial support for this work. In particular, Lori Diachin provided the simplicial mesh component and Carl Ollivier-Gooch the GRUMMP volume mesh component. Carl was also responsible for the Mesh Unit Test program. Contracts for the mesh interface specification were defined in collaboration with Lori, Carl, and Kyle Chand. As leads of the project responsible for the Babel toolkit, Tom Epperly and Gary Kumfert supported this work. Many thanks for the useful feedback also go to those who reviewed this paper: Tom Epperly, Prem Devanbu, Steve Dahlgren, and the anonymous reviewers.

References

1. United States Department of Energy: TASCS Initiative. `http://www.scidac.gov/compsci/TASCS.html`
2. United States Department of Energy: SciDAC Initiative. `http://www.osti.gov/scidac/`
3. Meyer, B.: The grand challenge of trusted components. In: Proceedings of the 25th International Conference on Software Engineering (ICSE '03), Portland, OR (May 3–10, 2003) 660–667

4. McIlroy, M.D.: Mass produced software components. In: Proceedings of the NATO Software Engineering Conference. (October 1968) 138–155 Also available at `http://cm.bell-labs.com/cm/who/doug/components.txt`.
5. Alexeev, Y., *et al.*: Component-based software for high-performance scientific computing. In: Proceedings of Scientific Discovery through Advanced Computing (SciDAC 2005), San Francisco, CA, USA (June 26-30, 2005)
6. Armstrong, R., Beholden, D.E., Dahlgren, T., Elswasif, W.R., Kumfert, G., McInnes, L.C., Nieplocha, J., Norris, B.: High end computing component technology (white paper). In: Workshop on the Road Map for the Revitalization of High End Computing, Washington, DC, USA (2003)
7. Bernholdt, D.E., *et al.*: A component architecture for high-performance scientific computing. International Journal of High-Performance Computing Applications, ACTS Collection special issue (November 2005)
8. Common Component Architecture (CCA) Forum: Cca. `http://www.cca-forum.org/`
9. 610.12-1990, I.S.: IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA. (September 1990)
10. Lawrence Livermore National Laboratory: Babel. `http://www.llnl.gov/CASC/components/babel.html`
11. Dahlgren, T.L., Devanbu, P.T.: Adaptable assertion checking for scientific software components. In: Proceedings of the Workshop on Software Engineering for High Performance Computing System Applications, Edinburgh, Scotland (May 24, 2004) 64–69
12. Dahlgren, T.L., Devanbu, P.T.: Improving scientific software component quality through assertions. In: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, St. Louis, Missouri (May 2005) 73–77
13. Brown, D., Freitag, L., Glimm, J.: Creating interoperable meshing and discretization technology: The terascale simulation tools and technologies center. In: Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations, Honolulu, HI (June 3–6, 2002) 57–61
14. Ollivier-Gooch, C., Chand, K., Dahlgren, T., Diachin, L.F., Fix, B., Kraftcheck, J., Li, X., Seol, E., Shephard, M., Tautges, T., Trease, H.: The TSTT mesh interface. In: Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV (January 2006)
15. GRUMMP — Generation and Refinement of Unstructured, Mixed-Element Meshes in Parallel. `http://tetra.mech.ubc.ca/GRUMMP/`.
16. Hoare, C.A.R.: The emperor's old clothes. Communications of the ACM **24**(2) (February 1981) 75–83
17. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of the Symposia in Applied Mathematics, Mathematical aspects of Computer Science. Volume 19., American Mathematical Society (1967) 19–32
18. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (October 1969) 576–580, 583
19. Adams, J.M., Armstrong, J., Smartt, M.: Assertional checking and symbolic execution: An effective combination for debugging. In: Proceedings of the 1979 annual conference. (1979) 152–156
20. Chen, W.T., Ho, J.P., Wen, C.H.: Dynamic validation of programs using assertion checking facilities. In: The IEEE Computer Society's 2nd International Computer Software and Applications Conference. (November 13–16, 1978) 533–538

21. Saib, S.H.: Executable assertions — an aid to reliable software. In: Proceedings of the 11th Asilomar Conference on Circuits, Systems and Computers. (November 7–9, 1977) 277–281
22. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Upper Saddle River, NJ (1997) Second Edition.
23. Bronsard, F., Bryan, D., Kozaczynski, W.V., Liongosari, E.S., Ning, J.Q., Ólafsson, A., Wetterstrand, J.W.: Toward software plug-and-play. In: Proceedings of the 1997 Symposium on Software Reusability (SSR '97), Boston, MA (May 17–20, 1997) 19–29
24. Kozaczynski, W.V., Ning, J.D.: Concern-driven design for a specification language supporting component-based software engineering. In: Proceedings of the 8th International Workshop on Software Specification and Design. (1996) 150–154
25. Object Management Group: CORBA basics. `http://www.omg.org/gettingstarted/corbafaq.htm`
26. Sankar, S., Hayes, R.: ADL — an interface definition language for specifying and testing software. ACM SIGPLAN Notices, IDL Workshop **29**(8) (August 1994) 13–21
27. Hamie, A.: Enhancing the object constraint language for more expressive specifications. In: Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC '99). (December 7–10, 1999) 376–383
28. Verheecke, B., Straeten, R.V.D.: 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) 23–32
29. Collet, P., Ozanne, A., Rivierre, N.: Enforcing different contracts in hierarchical component-based systems. In: Proceedings of the 5th International Symposium on Software Composition (SC '06), Vienna, Austria (March 25–26, 2006) 50–65
30. Leavens, G.T., Rustan, K., Leino, M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. Technical Report TR 00-15, Iowa State University, Ames, Iowa (August 2000)
31. Edwards, S.H.: Making the case for assertion checking wrappers. In: Proceedings of the RESOLVE Workshop. (June 2002) Also available as Virgina Tech Technical Report TR-02-11.
32. Heineman, G.T.: 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)
33. Liblit, B., Aiken, A., Zen, A.X., Jordan, M.I.: 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) 141–154
34. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Sampling user executions for bug isolation. In: Proceedings of the 1st Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '03), Portland, OR (May 2003) 3–6
35. Chilimbi, T.M., Hauswirth, M.: Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA (October 9–13, 2004) 156–164
36. Collet, P., Rousseau, R.: Towards efficient support for executing the object contraint language. In: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 30), Santa Barbara, CA (August 1–5, 1999) 399–408