LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TH-235341

Performance-Driven Interface Contract Enforcement for Scientific Components

T. L. Dahlgren

May 9, 2008

To appear on the June 2008 degree list.

Performance-Driven Interface Contract Enforcement for Scientific Components

By

TAMARA LYNN DAHLGREN
B.S./B.A. (California State University, Stanislaus) 1985
M.S. (California State University, Sacramento) 1993

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in charge

2008

Performance-Driven Interface Contract Enforcement for Scientific Components

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
UCRL-TH-235341.

To my husband, Stephen, and our children, Kathryn and Shaun, and to those who hesitate to pursue their potential.

Acknowledgments

Many people provided support, encouragement and guidance as I navigated the challenging waters of graduate study combined with full-time employment. First and foremost, I thank my husband, Stephen, and children, Kathryn and Shaun. Their encouragement, patience, and many sacrifices during this long journey have made the completion of this dissertation possible.

I am grateful to my adviser, Prem Devanbu, for pushing me to keep looking for a relevant, cutting-edge research topic. In the process, I read and absorbed a great breadth of related and nearly related literature — the most relevant of which appear in the bibliography. Also, in pointing me to Ben Liblit's remote debugging work, Prem supplied the inspiration that allowed me to effectively combine my dissertation research with my work assignment for most of this endeavor.

Members of my Qualifying (Q) and Doctoral (D) committees — Zhaojun Bai (D), Prem Devanbu (Q, D), Paul Dubois (Q), Alan Laub (Q), Raju Pandey (Q, D), and Zhendong Su (Q) — provided useful feedback and suggestions. Special thanks go to Paul and Zhaojun for their appreciation of my unusual situation and much needed encouragement and support.

Thanks also go to those who reviewed versions of this thesis. Paul Dubois, Pete Eltgroth, and Tom Epperly critiqued early drafts. Raju Pandey's questions and feedback were especially insightful. And my husband, Stephen, provided many editorial improvements.

Several people involved with the UC Davis College of Engineering Instructional Television (ITV) Program and its link with Lawrence Livermore National Laboratory (LLNL) provided outstanding encouragement and support. In giving his presentation at LLNL on the program, the late Harry Brandt — ITV Program founder and UC Davis Professor Emeritus — re-awakened my interest in pursuing this degree. He also provided essential encouragement and advice as I contemplated starting down this path and again as I neared completion. I also want to thank Kathy Zobel for her kindness, encouragement, and advice. Thanks go to Claire Daughtry for her support as well.

This research is built on software developed by the LLNL Components Project, so

I first thank Scott Kohn for spearheading the project in its formative years and for giving me the opportunity to find my research niche. Thanks also go to Tom Epperly and Gary Kumfert for their continued support after Scott's departure.

Projects whose components were used in the experiments are especially deserving of my gratitude for without their software the studies would not have been as relevant. First, I thank Lori Freitag Diachin (LLNL) for giving me the opportunity to collaborate with the Terascale Simulation Tools and Technologies (TSTT) Center and for donating her simplicial mesh implementation. I also appreciate Carl Ollivier-Gooch's (University of British Columbia) enthusiasm for my research and his donation of an early version of his GRUMMP component and the TSTT mesh test suite. Thanks also go to Lori, Kyle Chand (LLNL), and Carl for defining the TSTT contracts.

For their interest in and encouragement of my research from its fledgling stages, I'd like to express my appreciation to members of the Common Component Architecture (CCA) Forum — especially Randy Bramley (Indiana University-Bloomington), David Bernholdt (Oak Ridge National Laboratory), Rob Armstrong (Sandia National Laboratory), and Lois Curfman McInnes (Argonne National Laboratories) — and to Dan Quinlan (LLNL).

Finally, I would like to thank current and former LLNL managers for their words of encouragement, support, and/or advice: Steve Ashby, Dona Crawford, Trish Damkroger, Paul Dubois, Pete Eltgroth, John May, and Jim McGraw. I am especially indebted to Paul, Pete, and John for their advice and encouragement during critical junctures.

Most of this research was funded under the auspices of the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) programs Center for Component Technology for Terascale Simulation Software (CCTTSS) and Center for Technology for Advanced Scientific Component Software (TASCS).

Contents

List of Figures

List of Tables

Abstract

Performance-driven interface contract enforcement research aims to improve the quality of programs built from plug-and-play scientific components. Interface contracts make the obligations on the caller and all implementations of the specified methods explicit. Runtime contract enforcement is a well-known technique for enhancing testing and debugging. However, checking all of the associated constraints during deployment is generally considered too costly from a performance stand point. Previous solutions enforced subsets of constraints without explicit consideration of their performance implications. Hence, this research measures the impacts of different interface contract sampling strategies and compares results with new techniques driven by execution time estimates. Results from three studies indicate automatically adjusting the level of checking based on performance constraints improves the likelihood of detecting contract violations under certain circumstances. Specifically, performance-driven enforcement is better suited to programs exercising constraints whose costs are at most moderately expensive relative to normal program execution.

Chapter 1

Introduction

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 extends decades of research in component technology and software quality. A *component* is an independent software unit with an interface specification describing how it should be used [129]. Hence, caller and callee are loosely coupled through the methods, or routines, defined in the callee's interfaces (and classes), thereby enabling componentization of libraries as well as commercial and third-party binaries. Specifications, in terms of executable contracts on callers and callees, identify properties which must hold at key points during invocations of defined methods. This research pursues practical solutions to adapting the level of contract enforcement to performance constraints.

Interchangeable components guided by varying characteristics, such as the underlying model, precision, and reliability, are key features of the vision published in McIlroy's 1968 seminal paper on software components [125]. Grassroots efforts begun in the late 1990's by the Common Component Architecture (CCA) Forum [4, 7, 16, 31] seek to bring component-based software engineering to the high-performance scientific computing community. The CCA Forum has since adopted the Scientific Interface Definition Language (SIDL) for its interface specifications and the rest of the Babel toolkit — discussed in Chapter 3 — for generating programming language interoperability middleware [111].

The Institute of Electrical and Electronics Engineers (IEEE) [1] defines *quality*

```
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 1.1: Vector norm contracts example.

as the degree to which a system, component, or process meets its specifications, needs, or expectations. Interface contracts consist of precondition, postcondition, and/or class invariant clauses belonging to the interface specification, not the underlying implementation(s). *Precondition* clauses consist of assertions on properties which must hold prior to method execution. *Postcondition* clauses contain assertions which must hold upon method completion. *Class invariants* apply before and after execution of all defined methods. Hence, interface contracts are specifications amenable to automated enforcement.

Figure 1.1 shows the contract specification for a *norm* method on a vector of doubles. In this case, `norm` is a function taking `tol` as an argument and returns 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).

Many conscientious developers validate inputs to protect their methods (or routines) from bad data during deployment. This practice, sometimes referred to as "bullet-proofing" or "defensive programming", is related to interface contracts in that both involve conditions that must hold at specific points during program execution. The basic intent of these validations is to catch potential input-related problems before they cause the program to unexpectedly crash. These checks should always be retained to support graceful program termination. Contracts, on the other hand, may not be enforced during deployment.

Furthermore, interface contracts are broader in scope in that they may include constraints on properties of the input, output, component, and component state.

Since scientific components are developed by people with different backgrounds and training, it is not safe to assume 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 provide some assurances for catching failures at the interface regardless of the programming discipline used by component implementers.

However, the community's performance concerns can be a roadblock to the adoption of contract enforcement during deployment; hence, the pursuit of enforcement sampling using performance constraints. Therefore, performance-driven enforcement of interface contracts is pursued as a mechanism for helping scientists gain confidence in software built from plug-and-play software components.

This chapter summarizes the purpose, goals, and results of this research. The motivation behind performance-driven interface contract enforcement is discussed further. Research goals are presented before key challenges and results are described.

1.1 Motivation

There is a growing interest in component-based software engineering (CBSE) to facilitate the re-use of legacy software as plug-and-play components in multi-scale, multi-physics models. The resulting complexity of these applications — especially using components 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 performance implications of new technologies [29, 102]. According to Jaideep Ray, the Common Component Architecture (CCA) Forum conducted an informal poll to determine the amount of overhead that can be tolerated. Their findings indicate computational scientists are typically willing to incur no more than about ten percent performance 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, there is increased risk of incorrect or unanticipated usage patterns when using unfamiliar components. Furthermore, such applications rely on input data set combinations with the potential of leading to unexpected component behavior.

Interface contracts can provide clear documentation of service constraints. When specified in an implementation-neutral language, interface contracts can also serve as a basis for their consistent instrumentation and enforcement. Pinpointing the exact statement or module in which the computation fails would be ideal; however, the ability to detect contract violations during execution can still save on debugging efforts.

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

Re-use of legacy software as plug-and-play components in multi-scale, multi-physics models increases the risk of incorrect or unanticipated usage patterns and input data set combinations leading to erroneous component behavior. Executable interface contracts can facilitate testing and debugging but performance concerns must be addressed to gain their acceptance within the scientific computing community. Consequently, this research focuses on using performance criteria to drive contract sampling for enforcement purposes.

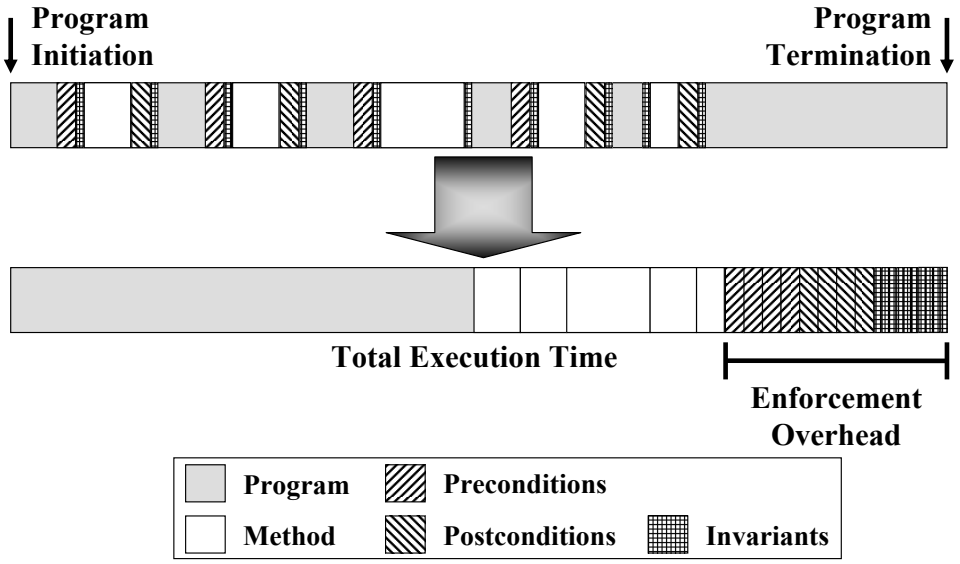

Figure 1.2: Contract enforcement overhead, where *Program* is the time spent in the application or caller; *Method* is the time spent in methods defined in the interface specification; *Preconditions* is the time attributed to precondition clauses; *Postconditions* is the time attributed to postcondition clauses; and *Invariants* is the time attributed to invariant clauses.

1.2 Problem

With the growing interest in CBSE for building multi-scale, multi-physics models using legacy software comes the challenge of providing mechanisms to facilitate debugging with minimal performance impact. Figure 1.2 illustrates the source of enforcement overhead for interface contract checking. A sequence of execution times spent on program statements, component methods, and individual contract clauses is illustrated in the top bar of the figure. The resulting overhead, shown in the bottom bar, consists of the amount of time spent checking preconditions, postconditions, and invariants. The sequence of times forms an *execution profile*.

The execution profile of a given program can vary significantly depending on the input set [22, 23, 156, 183]. This is clearly the case when execution time is a function of input size. Similarly, some implementations may terminate, that is short-circuit, routines early upon encountering certain values. In addition, different computing hardware and environments impact execution time [183, 187]. Architectural and system configuration factors — such as pipelining, cache, and memory — as well as programming language

factors — such as level of abstraction above the machine and compiler settings — all contribute to execution costs. The load on the system can also change within and across executions.

Another factor to consider is component usage — in terms of invocations of different combinations of interfaces — which can vary significantly from one application and environment to another [23, 25, 187]. Potential variability in an execution profile also occurs when one component is replaced, on-the-fly, with another conforming to the same interface but encapsulating a different algorithm and/or implemented in a different programming language.

Hence, a number of factors must be considered to facilitate debugging deployed software with minimal performance impact. External factors — such as system architecture and configuration — can affect program and contract execution times. Inputs, component interfaces actually used by a program, and the potential for swapping components at runtime also present problems for this research. Therefore, experiments need to consider each combination of program, component, input set, and execution environment separately when investigating interface contract enforcement overhead.

1.3 Research Goals

The vision of this research is to provide a mechanism for improving the quality of scientific applications built from deployed, third-party, plug-and-play components while addressing the community's performance concerns. The fundamental goal of the work reported here is to investigate the feasibility of performance-driven sampling as a means of reducing the impact of interface contract enforcement on program execution time. This is accomplished through practical solutions to enforcement based on contract sampling driven by performance constraints.

Performance-driven enforcement makes contract sampling decisions on a temporal basis, applying a user-specified overhead limit to execution times up to and including those of the current method. Each time a method with contracts is encountered, a decision must be made as to whether the contracts are to be enforced. Figure 1.3 illustrates the basic principle. Before the contracts of the first method, m_1, are checked, an estimate is made of

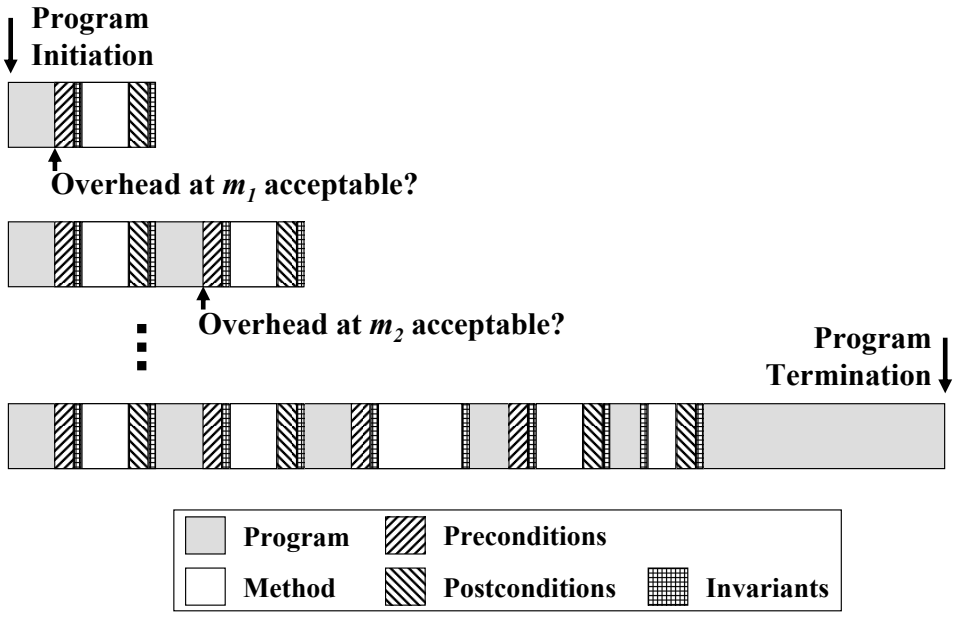

Figure 1.3: Performance-driven enforcement, where *Program* is the time spent in the application or caller; *Method* is the time spent in methods defined in the interface specification; *Preconditions* is the time attributed to precondition clauses; *Postconditions* is the time attributed to postcondition clauses; and *Invariants* is the time attributed to invariant clauses.

the overhead up through m_1's invocation. If the estimated overhead is within the tolerance limit, the contracts are checked. This decision process is repeated for m_2, m_3, and etcetera until the program terminates. Hence, this research constrains the amount of time spent checking contract clauses to a user-specified percentage of the total time spent executing statements in the program and within component methods.

This approach hinges on two basic precepts. First, the amount of time spent enforcing all interface contracts associated with the methods called by a program exceeds the user's performance overhead tolerance. Second, a sufficient number of calls are made to methods with relatively moderate contract execution costs to make sampling worthwhile.

Three studies are conducted to empirically evaluate the performance overhead, checked contracts, and detected violations using a variety of sampling strategies. Each study involves at least one performance-driven enforcement technique. All such policies make enforcement sampling decisions based on accumulated execution time estimates and a user-specified enforcement overhead limit. The policies are intended for programs incurring unacceptable overhead when enforcing all contracts while providing sufficient contract

sampling opportunities.

1.4 Challenges

CBSE and plug-and-play component computing are cutting edge technologies in scientific computing. Current efforts are underway in the CCA Forum to build the requisite technologies [177]. Babel provides foundational support in terms of programming language-neutral specifications — through the Scientific Interface Definition Language (SIDL) — and language interoperability middleware. As a result of the relatively limited use of the technologies at the time of these studies, the research described herein is inherently exploratory. Data are gathered for the comparison of different enforcement strategies based on three metrics. The experimental process attempts to mitigate issues interfering with the validity of results.

1.4.1 Subjects

The intent of this research is to help improve the quality of multi-language, multi-component, plug-and-play applications. However, a number of factors relating to the maturity of scientific component technologies and availability of suitable applications impact the selection of subjects.

First, the mainstream scientific computing research community is still in the early stages of investigating software component technologies. Issues of performance and the complexity of manually componentizing legacy software has led researchers in the CCA Forum to advocate starting with a few methods performing large computations. While this is a practical approach for those adopting component technologies, it is unfortunate for this research as it delays finding real applications with properties making sampling contracts worthwhile.

The programs readily available for these studies are simple examples, most of which make only a few calls to component methods with basic arguments. More importantly, the components generally lack contracts since enforceable interface contract specifications are not part of mainstream programming — especially in the scientific computing research community. Consequently, most of those programs present insufficient interface

contract sampling opportunities. So, even when real libraries are available, they must be componentized and have their contracts defined. A critical step in the componentization process involves the manual integration of library calls into the implementation layer of the generated language interoperability middleware.

Identifying and defining meaningful interface contracts using results, relationships between arguments, and component state requires an in-depth understanding of the underlying algorithms [6]. This level of expertise can typically only be provided by the original developer(s). In some cases, other developers having extensive experience using a component or library may also have the required knowledge. Hence, meaningful interface contracts must be defined by component experts.

Prototype interface contract enforcement features are currently integrated only in the Babel C and C++ bindings. The need for existing C/C++ programs and components already wrapped with Babel middleware limit the available subjects suitable for this research to relatively simple, single-component programs. To overcome the resulting limitations, different methods and/or input set combinations offering a wider range of execution profiles are employed. Hence, the resulting trials provide some insights with the potential of encouraging future collaborations on more sophisticated programs and applications.

1.4.2 Metrics

Three metrics are relevant for this research: enforcement overhead, number of interface contracts checked, and number of violations detected. Enforcement overhead is the driving metric behind this research since performance is a critical issue for the scientific computing community. The number of checked contracts for a given trial is important for comparing the level of enforcement associated with different sampling strategies. Finally, the number of detected violations provides insights into the effectiveness of each sampling strategy in terms of catching interface errors.

Capturing data for these metrics requires runtime instrumentation, applied consistently across trials, to track the numbers of method calls, contract checks, and etcetera. This is accomplished primarily through the Babel-generated language interoperability middleware. Although the real-time collection process affects execution time, it is necessary

to collect accurate data. As discussed in Section 1.2, elapsed time is still unstable since it varies from one run to another even without the added instrumentation. Fortunately, the effects can be mitigated to some extent by following a consistent experimental process.

1.4.3 Experimental Process

The goal of the experimental process is to control conditions as much as possible to minimize potential bias. As discussed in Section 1.4.1, a wider variability in execution profiles is obtained through combinations of programs and input sets. As a result, ninety-five trials are conducted in each of the first two studies, while the third study involves a total of thirteen trials. Experimenter expectancy bias is mitigated by the use of the same processes applied to each trial within a study. Every enforcement strategy, including those providing baseline metrics data, is executed on the same software. Outliers in the data are not eliminated from computations of average enforcement overheads to avoid the potential of biasing the data. Hence, the trials, measures, and enforcement policies are well-defined and consistently applied on a trial basis with every effort made to ensure conditions are as similar as possible across runs.

1.5 Results

This research investigates the impacts of using execution time overhead limits to drive interface contract enforcement. Three studies collect data for baseline and sampling policies. Each study uses a different approach to obtain execution time estimates for performance-driven enforcement. Results indicate enforcing contracts based on performance constraints is appropriate in the target conditions described in Section 1.3. Another important factor appears to be the execution time estimates of the contracts relative to the time spent executing statements within component methods. Performance-driven enforcement is more appropriate for checking contracts whose execution times are no more than moderately expensive relative to the time spent in the method. The accuracy of execution time estimates affects the ability of the system to adapt the level of enforcement to the program. In addition, estimates impact the system's ability to manage the actual level of enforcement overhead.

1.6 Summary

Performance-driven interface contract enforcement aims to improve the quality of applications built at runtime from scientific components by partially checking the explicit obligations on component callers and implementers. This research focuses on sampling based upon performance criteria to encourage the specification of interface contracts by scientific software developers and the enforcement of those contracts during deployment by scientists. The emphasis on performance is necessary since the community typically requires evidence indicating a new technology will not add excessive overhead to their applications.

Since scientific plug-and-play component technologies are still in their infancy and since there is a lack of executable interface contracts in existing component specifications, this research is inherently exploratory in nature. Consequently, challenges include: locating suitable subjects; gathering metrics data with as little impact on results as possible; and ensuring a consistent process is followed for all experiments within a study.

The challenges are addressed in several ways. Relatively simple, single-component programs are employed as subjects in the studies. A wider range of execution profiles, as described in Section 1.2, are obtained by varying input set combinations, when possible, to form separate trials. Metrics are collected using the same program-specific instrumentation for all enforcement strategies. The same basic experimental process is followed for all trials in a given study. While the conclusions of the studies cannot be generalized to plug-and-play scientific component computing applications, the infrastructure developed and insights gained through the studies should encourage follow-on collaborations.

Chapter 2 provides background information on key issues such as software quality, component contracts, and related work. Due to the many technologies supporting assertions and interface contracts, the related work section focuses on those efforts actually employing sampling techniques for enforcement during deployment. Interface contract instrumentation and enforcement rely on prototype versions of the Babel language interoperability toolkit tailored to the needs of scientific computing. Chapter 3 summarizes extensions to Babel's specification language, compiler, and runtime library that apply across the reported studies.

An overview of three studies is given in Chapter 4. Chapter 5 presents the approach and results of the first study, which relies on runtime timing for a performance-driven variation of the countdown-based work of Liblit *et al.* [117, 118]. In order to address the impact of the additional runtime cost and to gain a better understanding of characteristics of interface contracts in scientific computing, the remaining studies, described in Chapter 6, use *a priori* execution times for global enforcement decisions. The first of the two studies relies on data from simple timing experiments. As a result of the corresponding violation detection results, enforcement tracing is introduced into the toolkit to provide more tailored execution times for the final study. Finally, Appendix A provides abbreviations, acronyms, and definitions used throughout this document.

Chapter 2

Background

To quickly recap the motivation for this research, a critical requirement for component plug-and-play is compliance with a common interface specification. However, it is not enough to share the same method signatures. Different implementations must also behave in a consistent manner. The specification of behavioral constraints on interfaces is one approach to ensuring consistency. These constraints are commonly referred to as interface contracts.

While components developed from formal specifications tend to be of higher overall quality [150], the addition of executable interface contracts to mature software has also been shown to improve quality [157]. Unfortunately, the inherent overhead in runtime checking can be unacceptably high for some applications. Different strategies are therefore used to check subsets of contracts or their assertions at runtime. More relevant to this research are recent efforts to use sampling techniques to drive assertion enforcement.

2.1 Executable Interface Contracts

In 1971, Parnas [145] advocated machine testable, implementation-neutral component specifications. Thirty years later, Baudry *et al.* [13] found components with high encapsulation and well-defined, contractually-specified interfaces to be more effective at improving the quality of systems than implementation-dependent assertions used for defensive programming. Known as *component contracts*, assertions at this level typically

constrain functional aspects of component behavior. This emphasis has the advantage of enabling the application of contracts across the interchangeable components that comprise McIlroy's [125] 1968 vision. The last decade has witnessed numerous efforts to address executable interface contracts at a level of abstraction above the implementation language. This section summarizes seven such technologies.

The Architectural Specification Language (ASL) [27, 106] encompasses a family of design languages for Component-Based Software Engineering (CBSE). Their most relevant design language is called Interface Specification Language (ISL). ISL extends the basic method signatures of CORBA [136] Interface Definition Language (IDL) with preconditions, postconditions, invariants, and protocol (or states). The ASL also supports languages for component composition and configuration.

The Assertion Definition Language (ADL) [160] is another technology that extends CORBA IDL. In this case, however, only postcondition clauses are supported. The focus of ADL is to facilitate formal specification and testing of software components so an additional language is supplied for describing test data.

Hamie [80, 81] advocates extending the Object Constraint Language (OCL). OCL is a textual language for expressing modeling constraints. The proposal involves the addition of invariants to class diagrams and preconditions, postconditions, and guards to state transition diagrams.

Another extension to OCL is pursued by Verheecke and van Der Straeten [180]. They developed a framework that translates OCL constraints into executable constraints for Java. The assertions are implemented as separate methods in constraint classes.

Although it is specific to the Java language, the Java Modeling Language (JML) [115] is another example of the definition of contracts through a modeling language. In this case, precondition and postcondition clauses are specified as Java comments. Their Extended Static Checker (ESC) translates the specified clauses into HTML documentation and runtime debugging statements.

Edwards *et al.* [59] automatically generate wrappers from specifications. Their goal is to separate enforcement from the client and implementation. "One-way" wrappers are used to check preconditions. While they also have "two-way" wrappers to check preconditions and postconditions, the wrappers are not automatically generated.

Table 2.1: Rosenblum's [157] classification of interface assertions.

Method Interface Assertion Classification
Consistency Between Arguments
Dependency of Return Value on Arguments
Effect on Global State
Context in Which Function Is Called
Frame Specifications
Subrange Membership of Data
Enumeration Membership of Data
Non-Null Pointers

Finally, Heineman [89] employs Run-time Interface Specification Checker (RISC) for the enforcement of preconditions and postconditions. The goal is to enforce contracts for all methods, not just those in the public interface. So instead of using wrappers, they automatically instrument source code with hooks at the before and after phases of method calls.

The seven technologies described in this section rely on high-level specifications of interface contracts to establish behavioral constraints on components. Each translates the specifications into runtime checks. Some technologies implement the checks through wrappers, while others integrate them into the components.

2.2 Interface Assertion Classification

Interface contracts are formed from assertions that constrain arguments, return values, and component state visible at the interface. A classification of the types of assertions appropriate for interface contracts is provided by Rosenblum [157]. Table 2.1 lists the eight classes based on a study of systems written in the C programming language. Assertion arguments, return values, global state, and calling context are inherent within the context of interfaces. The *calling context* identifies relationships between arguments and global variables for valid invocations. *Frame specifications* identify (by reference) arguments, global variables, and state to remain unchanged by the invocation. Finally, subrange, enumeration, and non-null pointers are argument-specific assertions. Hence, five of the eight classes are assertions relating to arguments, one reflects assertions on state,

and two represent assertions on arguments and state.

2.3 Interface Contract Effectiveness

Executable interface contracts have been the subject of several studies investigating their effectiveness at detecting faults in software. Each study considers different aspects of contract effectiveness. Two of the three studies summarized in this section consider initial and improved contracts. The third study compares contracts versus oracles for detecting seeded faults.

Rosenblum [157] describes a study he conducted on a variety of 10K to 20K SLOC systems over a 5 year period using APPC (an Assertion Preprocessor for C). Interface assertions were mapped to faults in the software. Assertions were then identified that could catch the faults had the assertions been supported. In all, 58% of the faults are tied to interface assertions supported by APPC while another 16% require more powerful assertion features, such as sequencing constraints. Of the eight classes of interface assertions described in Section 2.2, Rosenblum found the most valuable assertions are those reflecting dependencies of the return value on the method's arguments. In fact, that class was tied to 32% of the faults, or nearly three times the faults associated with the next best class of assertions. His findings indicate even mature software can benefit from interface contracts.

Baudry *et al.* [13] report on the use and improvements of object-oriented contracts on a total of 233 classes in telecommunications and compilers. Initial contracts were found to be 58.5% effective on average at detecting faults. When improved, which included adding class invariants, contract effectiveness rose to a range of 50% to 84%.

Finally, Briand *et al.* [24] compare the use of contracts versus oracles with a small, 21 class, 2200 SLOC Automated Teller Machine application written in Java. Procedural and object-oriented faults were seeded with mutation operators to generate 69 variants of the application, only 47 of which demonstrated faults detectable by contracts. Their findings reveal an order of magnitude increase in their ability to diagnose faults using contracts versus test oracles.

The findings of these studies indicate interface contracts are capable of detecting significant numbers of faults in software systems. Clearly interface contracts are limited to

detecting violations triggered by faults visible at the interface. However, empirical evidence does suggest interface contracts can significantly increase the quality of software.

2.4 Traditional Enforcement Strategies

Historically contracts are enforced during testing and either disabled or removed prior to deployment. Different contract technologies provide a variety of options for selectively enforcing contracts. The most common strategies are: contract clause, software package, and software class. Less common alternatives are contracts on a method or severity basis.

Eiffel [128] supports the runtime selection of contract clause enforcement levels as follows: enforcement disabled, preconditions-only; preconditions and postconditions; preconditions, postconditions, and invariants. Java with Assertions (Jass) [99] and Jcontract [24] preprocessors also support enforcement on a contract clause basis. Similarly, Edwards [59] supports clause-based enforcement for Java but through a "bag"-like approach. That is, contract enforcement is through clause-specific enforcement wrappers. jContractor [100] supports contract clause, software class, and software package enforcement options.

Though less common, some technologies selectively enforce contracts or their assertions on a method or (assertion) severity basis. For example, in addition to package and class selectivity, iContract [107] allows enabling and disabling contract enforcement by method. The Annotation PreProcessor for C (APPC) [157] is reported to include support for explicit severity levels on an assertion basis.

Hence, traditional enforcement strategies span a broad range from "all-or-nothing" to individual assertion checks. All of these options depend on either the developer or user to identify the nature of the desired contracts or assertions to be enforced. And none of the strategies explicitly addresses the actual performance impact of enforcement that is often an important consideration for deployed software.

2.5 Related Work

Only two other efforts are known to use sampling during deployment to address the performance overhead of assertion checking. Both use sampling as an alternative to the traditional enforcement strategies described in Section 2.4. While sampling is typically used to characterize the behavior of systems [8], it has also been used for other purposes such as scheduling [75, 146], network tuning [123], and operating system tuning [155].

Liblit *et al.* [117, 118] concentrate on remote application profiling and statistical debugging of arbitrary code using automated instrumentation of user-defined assertions. They rely on uniform random sampling to detect infrequently occurring errors as a means to ensure fairness across assertions. Their countdown-based code duplication approach — adapted from Arnold and Ryder [8] — is used to address performance concerns. Finally, they use very low sampling rates to amortize the cost of assertion checking throughout the (remote) user community.

Like the work of Liblit *et al.*, the first study of this research uses automatic instrumentation of a countdown-down based code duplication approach for enforcement decisions. In this research, the countdowns enforce all assertions of the method's contract instead of individual assertions within a program body. In addition, performance-driven enforcement uses runtime timing and a user-specified overhead limit to initialize and reset the countdown.

Chilimbi and Hauswirth [34] focus on rarely occurring errors but within the context of their SWAT memory leak detection tool. They define three staleness predicates: never accessing; idle for a constant amount of time; and idle for an inactive period. Their tool automatically inserts these predicates — and only these predicates — into program bodies for sampling during deployment. Assertion 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.

Performance-driven enforcement also tends to favor earlier assertions. In the first study, all assertions within the contract are checked on the first call to each component method. Sampling rates for all of the studies in this research are based on execution time

estimates and a user-defined overhead limit.

Only two other efforts are reported to actually use sampling to guide assertion enforcement during deployment. Like this research, Liblit *et al.* handle arbitrary assertions. However, Chilimbi and Hauswirth target only three pre-defined assertions for memory leak detection. Both efforts sample assertions — not the interface contracts of this research — as a means to reduce the impact on program performance. Liblit *et al.* and Chilimbi and Hauswirth also automatically inject assertions and their sampling instrumentation into program bodies, not the language interoperability middleware of this research. Both related works also focus on detecting infrequently occurring errors in programs; whereas, this research is focused on automatically adjusting the checking level based on a user-defined performance limit.

2.6 Summary

This research is based on the premise that traditional testing and debugging mechanisms are inadequate for software components built for plug-and-play environments — especially when components may be implemented in different programming languages. At issue is ensuring components are used and implemented correctly within an arbitrary application context. *Correctness* is defined as "the degree to which a system or component is free from faults in its specification, design, and implementation" [1]. Executable interface contracts provide a mechanism for demonstrating correctness in terms of compliance with a specification. Eight classes of assertions are identified as relevant for use in interface contracts. Findings from several studies indicate interface contracts are capable of detecting significant numbers of software faults. However, performance implications of checking contracts during deployment are often considered unacceptable. Traditional strategies for selectively enforcing contracts — typically used during testing and debugging — include checking them on a package, class, or clause basis. Recent research into sampling assertions provides a general-purpose alternative for reducing the impact on performance during deployment. This research also uses sampling for enforcement purposes though the focus here is on using performance criteria to guide sampling.

Chapter 3

Common Babel Toolkit Extensions

The vision of this research is to help improve the quality of scientific applications built from third-party, plug-and-play components potentially implemented in different programming languages. Since the Common Component Architecture (CCA) Forum relies on the Babel toolkit [111] for the specification, generation, and support of its language interoperability middleware, Babel is leveraged in this research.

The Babel toolkit consists of a programming language-neutral interface specification language, compiler, and runtime library. The original purpose of the toolkit was to facilitate efficient, single-processor interoperability between programming languages commonly used in scientific computing. The specification language, called the Scientific Interface Definition Language (SIDL), is based on an object-oriented foundation of base classes, interfaces, exceptions, and built-in types. The Babel compiler translates SIDL specifications into wrappers mapping programming language-specific types to and from the common representation layer. Those wrappers rely on features of the SIDL runtime library to support basic operations and capabilities associated with SIDL types. Interface contract enforcement and data collection instrumentation has been integrated into all three parts of the toolkit. This chapter describes the extensions common to the versions of the prototype used in each of the three studies.

3.1 Scientific Interface Definition Language

The Scientific Interface Definition Language (SIDL), like the Interface Definition Language (IDL) provided by the Object Management Group (OMG) [138], is programming language-neutral. Both IDLs support the modular packaging of full method definitions specifying the type (e.g., integer, float) and mode (i.e., in, out, inout) of each parameter. Both also support enumerations, arrays, and multiple inheritance of interfaces. Unlike OMG IDL, SIDL provides basic type support for numeric complex and multi-dimensional, multi-strided arrays. The SIDL grammar was extended to include contract clauses and a rich set of expressions.

SIDL originally supported a combination of five main elements: packages, interfaces, classes, methods, and types [44]. This research required the addition of a sixth element; namely, assertions. *Packages* provide a mechanism for specifying name space hierarchies. Every SIDL file must specify *at least* one package. Packages consist of a combination of types, primarily in the form of interfaces and classes. *Interfaces* define a set of methods a caller can invoke on an object, or instance, of a class implementing the methods. *Classes* define a set of methods a caller can invoke on an object. SIDL supports multiple inheritance of interfaces but only single inheritance of classes. *Methods* define routines available for invocation by a caller. *Types* constrain parameter values, exceptions, and return values associated with methods. Finally, *assertions* are used within contract clauses to support constraints on properties of objects as well as argument and return values. For example, Figure 1.1 includes a property constraint in its postcondition clause. Specifically, the function `isZero()` is used to check that all elements of the vector object are zero. Contracts specified in interfaces and ancestor classes are inherited.

Extensions to the SIDL grammar include the interface contract productions listed in Figure 3.1. These productions support classic Eiffel [128] constructs. The `invariant` clause production applies to classes and interfaces, though it is generally referred to as a class invariant. A *class invariant* specifies constraints on a class. Therefore it identifies properties expected to hold true for all instances of the class both immediately before and after execution of each method. Those properties are expected to hold both before and after the execution of every method defined for or inherited by the class. The *precondition*

```
<class>          ::= [ abstract ] class <name>  [ extends <scoped-class-name> ]
                     [ implements-all <scoped-interface-name-list> ]
                     '{'  [ <invariants> ] (<class-method>)* '}' [';']
<interface>      ::= interface <name> [ extends <scoped-interface-name-list> ]
                     '{' [ <invariants> ] ( <method> )* '}' [';']
<class-method>   ::= [ ( abstract | final | static ) ] <method>
<method>         ::= (void | [ copy] <type>) <name> '[' <extension> ']'
                     '(' <argument-list> ')' [ local | oneway ]
                     [ throws <scoped-exception-list> ] ';'
                     [ <preconditions> ] [ <postconditions> ]
<invariants>     ::= invariant ( <assertion> )+
<preconditions>  ::= require ( <assertion> )+
<postconditions> ::= ensure ( <assertion> )+
<assertion>      ::= [ <label> ':' ] <assertion-expr> ';'
<assertion-expr> ::= <cond-expr> [ ( implies | iff ) <cond-expr> ]
<cond-expr>      ::= <or-expr>
<or-expr>        ::= <xor-expr> [ or <xor-expr> ]
<xor-expr>       ::= <and-expr> [ xor <and-expr> ]
<and-expr>       ::= <eq-expr> [ and <eq-expr> ]
<eq-expr>        ::= <rel-expr> [ ( == | != ) <rel-expr> ]
<rel-expr>       ::= <add-expr> [ ( < | > | <= | >= ) <add-expr> ]
<add-expr>       ::= <mult-expr> [ ( + | - ) <mult-expr> ]
<mult-expr>      ::= <pow-expr> [ ( * | / | mod | rem ) <pow-expr> ]
<pow-expr>       ::= <unary-expr> [ ** <unary-expr> ]
<unary-expr>     ::= <postfix-expr> [ ( not | is ) <postfix-expr> ]
<postfix-expr>   ::= <primary-expr> | <identifier>  '(' [ <arg-expr-list> ] ')'
<primary-expr>   ::= <reserved> | <identifier> | <integer> | <float>
                     | '(' <cond-expr> ')'
<reserved>       ::= null | result | true | false | pure
<arg-expr-list>  ::= <cond-expr> ( ',' <cond-expr> )*
```

Figure 3.1: SIDL contract grammar productions.

and *postcondition* clause productions, `require` and `ensure` respectively, apply to methods. A *precondition* declares constraints on invocation of a method while a *postcondition* constrains its effects. The number of productions shows the richness of the assertion expression grammar. Specification of these boolean expressions requires the addition of basic operators found in most common programming languages. The conditional operators `iff` (i.e., if-and-only-if) and `implies` enable more expressive contracts. The `implies` operator is useful for ensuring arguments, such as a null pointer, are not passed to a method incapable of supporting them. For example, in `(outHandle != null) implies (size(outHandle) >= 0)` the check for null is needed because the built-in `size()` function does not gracefully handle a null array argument. One of the clause-specific exceptions — described in Section 3.3 — is raised when contract checking detects a violated assertion.

In order to support assertions on object properties, the new grammar allows function invocations. For convenience, twenty built-in functions — listed in Table 3.1 — are provided for numeric constants and built-in SIDL arrays. Constant-time functions consist of built-in array accessors and simple numeric value comparators while linear-time functions include existential and universal quantifiers. The relational expression, *expr*, in the functions *all*, *any*, *count*, and *none* can take one of three forms. Those forms are: $u\ r\ v$, $u\ r\ n$, and $n\ r\ v$, where $u, v \in$ *SIDL arrays*, $n \in$ *Numbers*, and $r \in$ *Relational operator* (i.e., $\{<, >, <=, >=, ==, !=\}$). The first, $u\ r\ v$, is an assertion on the values of the corresponding elements in the two arrays indicating the values satisfy the specified relationship. That is, $u_i\ r\ v_i$ is applied $\forall i \in 0\ ..\ (size(u) - 1)$. Similarly, $u\ r\ n$ and $n\ r\ v$ are constraints requiring the values of every element in the array satisfy the relationship with the numeric value, n. The contracts of functions invoked during the enforcement of another method's contracts are by-passed by the middleware.

Significant extensions in the SIDL grammar are needed to support interface contracts. Over twenty productions support the specification of Eiffel-inspired class invariant, precondition, and postcondition clauses. In addition, twenty-two operators plus numeric and boolean literals can be used in assertion expressions. For example, method calls are allowed in assertion expressions to support checking encapsulated object properties. Finally, twenty built-in, convenience methods support a variety of functions primarily in terms of accessing SIDL array contents, such as universal and existential quantifiers. Overall, an

Table 3.1: Built-in functions added to SIDL/Babel for interface contracts.

Function	**Returns**	**O()**
dimen(u)	Dimension of array u.	1
irange(x, n_{low}, n_{high})	$True$ if x falls within the integer range of $n_{low}..n_{high}$.	1
lower(u, d)	Lower index of the d^{th} dimension of array u.	1
nearEqual(x, y, t)	$True$ if real values x and y are equal within the specified tolerance, t.	1
range(x, r_{low}, r_{high}, t)	$True$ if the real value x falls within the specified tolerance, t, of the range $r_{low}..r_{high}$.	1
size(u)	Allocated size of array u.	1
stride(u, d)	Stride of the d^{th} dimension of array u.	1
upper(u, d)	Upper index of the d^{th} dimension of array u.	1
all($expr$)	$True$ if the expression $expr$ evaluates to $true$ for each element in the specified array(s). For example, `all(`$u < v$`)` returns $true$ if the value of each element in array u is less than the value of the corresponding element in array v.	n
any($expr$)	$True$ if at least one element in the specified array(s) satisfies the expression $expr$. For example, `any(`$u = 0$`)` returns $true$ upon encountering the first element in array u whose value equals zero but returns $false$ if none of the elements have values equal zero.	n
count($expr$)	The total number of array elements satisfying the expression $expr$.	n
irange(u, n_{low}, n_{high})	$True$ if all elements in array u fall within the integer range $n_{low}..n_{high}$.	n
max(u)	The element in array u of maximum value.	n
min(u)	The element in array u of minimum value.	n
nearEqual(u, v, t)	$True$ if the corresponding elements in arrays u and v are equal within the specified tolerance, t.	n
none($expr$)	$True$ if none of the elements in the specified array(s) satisfies the expression $expr$. For example, `none(`$u >= 0.0$`)` returns $true$ if the value of none of the elements in array u are greater than or equal to 0.0.	n
nonDecr(u)	$True$ if the elements in array u are in order by non-decreasing value.	n
nonIncr(u)	$True$ if the elements in array u are in order by non-increasing value.	n
range(u, r_{low}, r_{high}, t)	$True$ if all elements in array u fall within the specified tolerance, t, of $r_{low}..r_{high}$.	n
sum(u)	Returns the total of the values of all of the elements in array u.	n

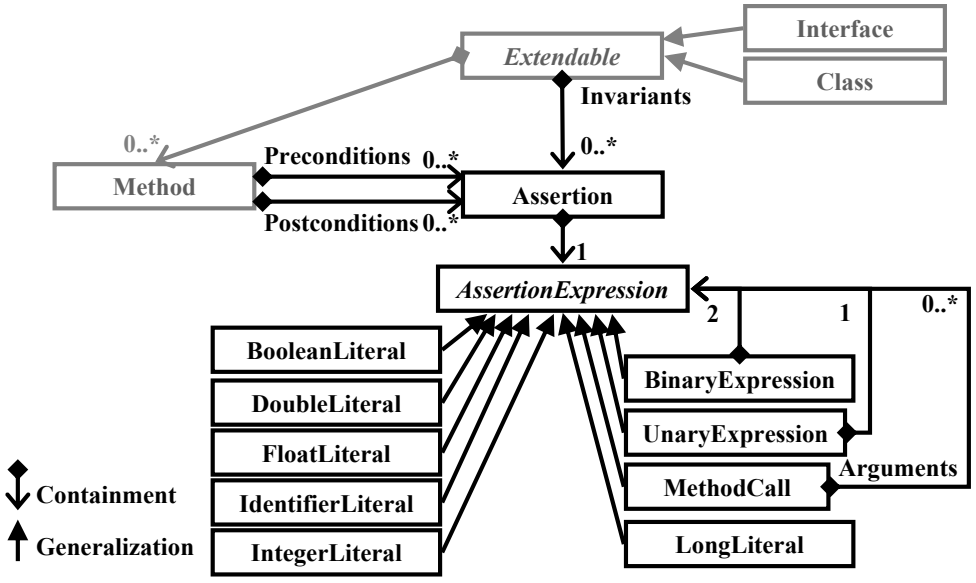

Figure 3.2: Extensions to the Babel compiler's abstract syntax tree. Names of interfaces (versus classes) appear in italics.

extensive collection of features support a rich variety of assertion expressions within SIDL interface contracts.

3.2 Babel Compiler

The infrastructure required to support SIDL elements is obtained through the automatic transformation of specifications into client-server language interoperability source code using the Babel compiler. As part of the process, SIDL elements are mapped onto an abstract syntax tree. The resulting nodes are then used to generate four layers of interoperability code.

Extensions to the abstract syntax tree consist of *Assertion*, *AssertionExpression*, and all subclasses of *AssertionExpression* shown in Figure 3.2. The base constructs from Figure 3.1 are mapped to nodes in the tree. That is, interfaces and classes — which inherit from `Extendable` — contain 0 or more `Assertion`s forming the (class) invariants clause. Similarly, methods contain 0 or more `Assertion`s in each of the precondition and postcon-

dition clauses. Each `Assertion` supports a single (unary or binary) `AssertionExpression`, which is required to evaluate to a boolean value. Each `AssertionExpression` can contain `result`, method calls, arguments, and literals within the immediate scope of the containing method.

The semantic analysis phase of interface contract processing, wherein the Babel compiler primarily checks for type conflicts, is deferred until after the tree is built. The delay is necessary, in large part, due to the need to resolve aspects of user-defined functions. For instance, function calls within assertion expressions must contain the proper number of compatible arguments in the correct order.

Additional semantic checks are performed at this stage as well. Functions called within assertions must be either built-in — as described in Section 3.1 — or annotated as side-effect "free". The designation `is pure` in the postcondition clause of a user-defined function is required to indicate the latter case. `IdentifierLiteral`s must be either built-in literals — such as `true` or `false` — or arguments from the method's signature. While literals and method calls are allowed, the `AssertionExpression` associated with each `Assertion` is required to return a boolean result. Hence, the Babel compiler performs the appropriate analyses of contracts prior to code generation.

Once the semantic analysis phase is complete, Babel generates language interoperability source code. These interoperability wrappers consist of the four layers shown in Figure 3.3. The top layer is the *stub*, which provides routines invoked by the caller. The stub layer performs any necessary argument and return value translations between the calling programming language and the intermediate layer. The next layer down the call stack is the *Intermediate Object Representation (IOR)*. The IOR provides the necessary object-oriented services, such as private object data holder and function pointer tables. The *skeleton* layer performs any necessary argument and return value translations between the interoperability middleware data types and those of the callee's programming language. Finally, the *implementation* layer provides hooks for each of the specified and several built-in methods for object management. For example, Babel automatically generates constructor and destructor method hooks. The component implementer is responsible for providing behavior within the implementation layer hooks. For efficiency and portability, the stub, IOR, and skeleton are all generated in ANSI C. The implementation layer,

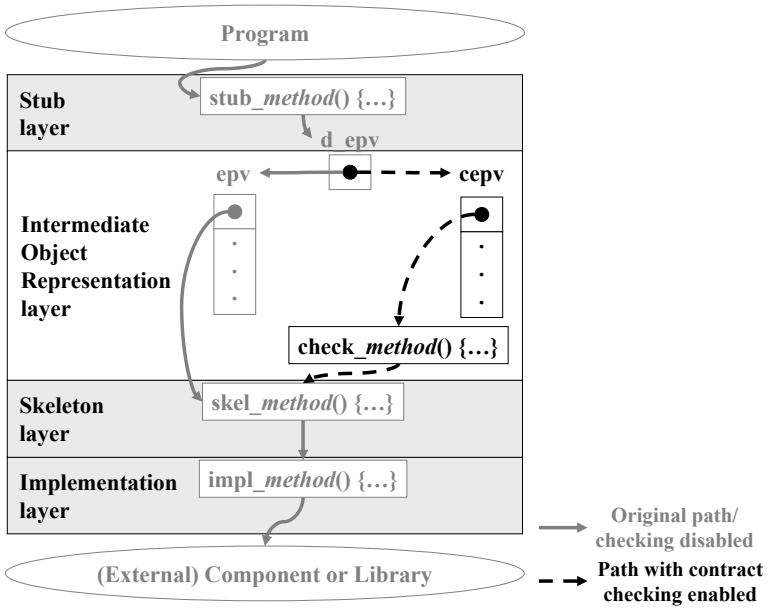

Figure 3.3: Babel-generated interface contract enforcement middleware.

on the other hand, is generated in the target language.

The Babel compiler adds `check_method` routines to the IOR, shown in Figure 3.3, to support interface contract enforcement. Individual assertions within a contract clause results in the generation of a corresponding check in the routine. Preconditions, if any, are grouped together under a single if-statement to ensure they are only executed when allowed by the current policy. Any invariants are grouped in the same manner before the call to the `skel_method`. Finally, postcondition and invariant checks are generated. Hence, all specified contracts are translated into enforcement checks within `check_method` routines. The implementation of the enforcement decision process within the routine, however, differs for each study.

A key challenge to providing efficient runtime enforcement is minimizing performance overhead. This is accomplished in part by the generation of two function pointer tables in the IOR. The primary table, called the *entry point vector* or *epv*, contains pointers to functions defined in the *skeleton* layer. The second table, denoted *cepv* in Figure 3.3, contains pointers to the new `check_method` routines. Each instance of an object maintains a pointer, *d_epv*, to the current static function pointer table. So the *d_epv* points to the *epv* table when interface contract enforcement is disabled or to the *cepv* table when enforcement

is enabled. The path taken when contracts are disabled is represented in the figure by solid grey lines. When contracts are enforced, the solid black lines show the alternative path taken within the IOR. Hence, programs only incur the overhead of enforcement routines when contract checking is enabled.

Since enforcement routines may include assertions using the built-in functions listed in Table 3.1, those functions are actually implemented as C macros. This approach is taken primarily to improve performance but the use of macros has an added benefit of reducing the number of built-in functions required to support various Babel types. That is, Babel's type safety and need for accessor methods to obtain contents of and metadata on SIDL arrays would normally require a separate built-in function for each possible combination of array argument types. Instead, a single macro corresponds to each built-in assertion function, where macro substitution ensures proper accessor methods are called.

Significant extensions added to the Babel compiler support interface contracts. Eleven classes added to the syntax tree reflect key elements of the new constructs. In all, the changes more than doubled the number of syntax tree-related source lines of code (SLOCs). Although some semantic analysis is performed during construction, most semantic checks are deferred since more context information is needed. For example, user-defined function calls may appear in contracts before their method signatures are defined. Once the semantic analysis phase is complete, Babel generates four layers of source code to wrap the implementation behavior with support for language interoperability. Contract enforcement is embedded in the intermediate object representation layer with care taken to address efficiency issues.

3.3 SIDL Runtime Library

The addition of contract enforcement raises the need to handle violations. Therefore, the SIDL runtime library was extended to include exceptions for each contract clause, as illustrated in Figure 3.4. Violations in the assertions within a contract clause result in a clause-specific exception being raised at runtime. That is, an assertion evaluating to false within a precondition clause results in a `sidl.PreViolation` exception. Similarly

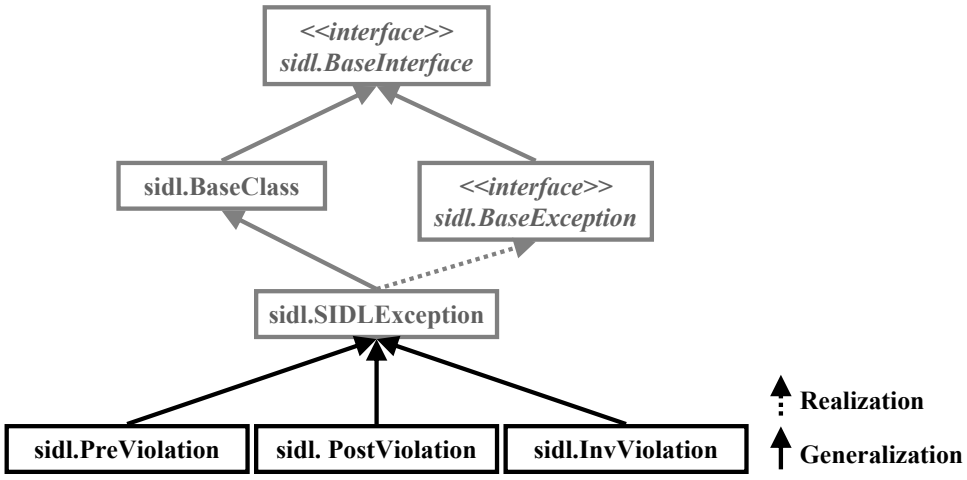

Figure 3.4: Contract violation exceptions.

violations within invariant and postcondition clauses result in a `sidl.InvViolation` and `sidl.PostViolation` exceptions, respectively. Hence, violations of assertions result in the automatic raising of an exception identifying the containing clause type.

3.4 Summary

This chapter describes common extensions to the Babel toolkit, which forms the proof-of-concept implementation of the enforcement features required for each of the studies. Over twenty productions are added to the SIDL grammar in support of Eiffel-inspired constructs for class invariant, precondition, and postcondition clauses. The new productions map to eleven new classes in the Babel compiler's abstract syntax tree. The new syntax objects perform semantic validations and generate the corresponding checks in new routines in the intermediate object representation layer of the programming language interoperability wrappers. Finally, three new, built-in exceptions are added to the SIDL runtime library identifying the type of the clause containing an assertion violated at runtime.

Chapter 4

Enforcement Studies Overview

The goal of this research is to determine the feasibility of performance-driven sampling as a means of controlling the impact of interface contract enforcement on program execution time. The guiding principle is to automatically reduce the level of enforcement when the costs are considered too high and increase it to the extent possible when the costs are below a given tolerance. That is, interface contract enforcement is automatically adjusted — as the program executes — in an attempt to meet performance constraints.

Three studies are performed to compare the effects of one or more performance-driven enforcement strategies. The same basic three-phase process establishes the core work flow. Experiments are conducted using each enforcement strategy under study. Data gathered during experiments are used to compute three metrics for comparing the relative effects of each policy. Each study relies on a different technique to estimate the execution times of the methods called and contracts checked in their trials.

4.1 Work Flow

A three-phase experimental process, illustrated in Figure 4.1, forms the basis of the work flow for each program in all of the studies. Phase one involves preparing the program. The program is repeatedly executed for each trial and interface contract enforcement policy in phase two. An input set consists of a single input file and/or a single input array size. Finally, phase three involves an analysis and comparison of the results on

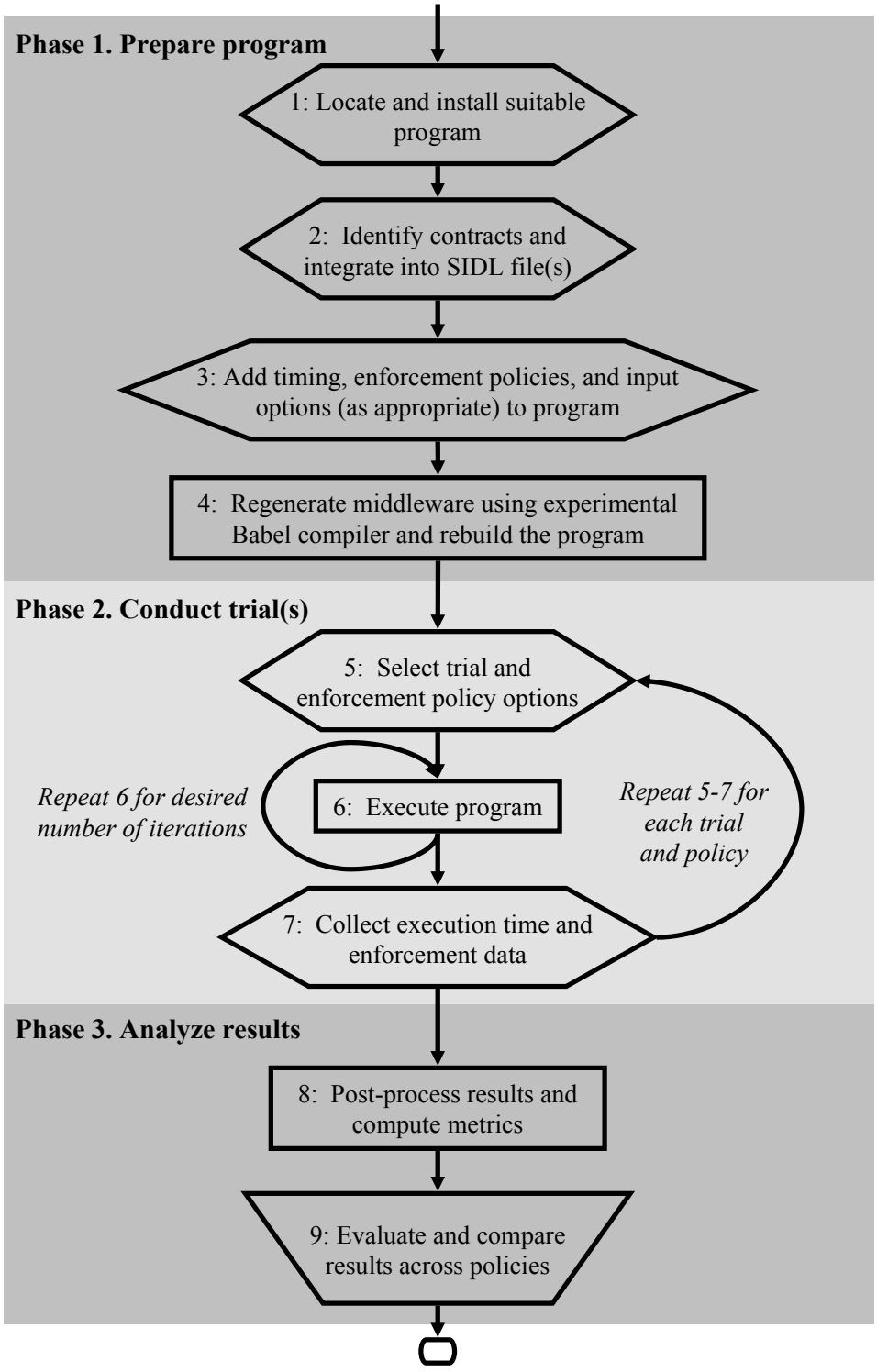

Figure 4.1: Basic experimental work flow for a single program.

a trial basis. Since each study involves experiments on a single computer, trials are formed from a combination of program, component, and input set.

As discussed in Section 1.4.1, this research is expected to be most relevant for programs making sufficient numbers of method calls for interface contract sampling purposes. So the first step in phase one is to locate and install such a program. Ideally, the program and/or its contracts provide characteristics not already represented by other programs or trials. Tests provided with the software are also run to ensure the installation was successful.

Collaboration with responsible developers yields contracts in step two. Those contracts are integrated into the SIDL file(s). This tends to be an iterative process as the developers gain a better understanding of features available for contracts.

In steps three and four, execution time instrumentation is added to the program. The full range of enforcement policy options and any input set combinations under study are also added to the program to facilitate experiments. Then the middleware is regenerated and the program compiled and linked.

Phase two focuses on conducting the trials through experiments executing the software. All data known about the contracts, methods, and programs are fixed for all experiments on a given trial. For example, the global enforcement studies require data on the complexity of assertions within contract clauses and the amount of time taken to execute individual methods and contract clauses.

Every effort is made to run all timing-related experiments at step six under the same operating conditions; that is, with the same networked computer as lightly loaded as possible. The instability of execution time measures, discussed in Section 1.4.2, is mitigated by averaging the execution times obtained from the iterations at step six.

Experiments performed with contracts disabled provide data for establishing execution time baseline measures for a trial. Another set of experiments are run with all contracts enforced to gather data on trial-specific totals for contracts checked and violations detected. Lastly, a set of experiments are performed for each enforcement sampling policy.

Consequently, scripts are created to automate steps five through seven for each trial to ensure all enforcement policy options and iterations are performed in a consis-

tent manner. They also facilitate the collection of execution timing and corresponding enforcement statistics data.

The results are analyzed in phase three. Post-processing is performed on the data collected in phase two to compute metrics for each interface contract enforcement policy. The data for all experiments are compared for each trial or, when the numbers of trials are large, for sets of trials. An assessment is then made of the relative value of basing enforcement decisions on performance constraints.

A preliminary pass over steps five through eight is made for the global enforcement studies to obtain execution time estimates. The first of those studies performs timing experiments with contract enforcement disabled and again with it fully enabled. The final study conducts enforcement tracing experiments checking all contracts during the preliminary pass.

Hence, the basic work flow consists of three phases for: preparing the program, conducting experiments on the trials, and analyzing the results. The phases are broken down into a total of nine steps repeated for each program under study. The second and third studies each perform a preliminary pass over steps in phases two and three using one or both baseline policies to obtain execution time estimates.

4.2 Enforcement Policies

Interface contract enforcement policies reflect different strategies for checking contracts. The two baseline policies — *Never* and *Always* — represent "all-or-nothing" strategies. Basic sampling techniques, like those in the related works, utilize simple sampling strategies to selectively enforce contracts. Recall from Section 1.3, this research introduces policies for making enforcement decisions using execution time estimates and performance constraints.

Figure 4.2 illustrates enforcement decisions for the two baseline policies. The *Never* policy disables interface contract enforcement. As discussed in Section 3.2, the instrumented enforcement routines are by-passed when the *Never* policy is in affect. The resulting execution time data are used to establish enforcement overhead metrics. The *Always* policy enables full contract enforcement. That is, all interface contracts encountered

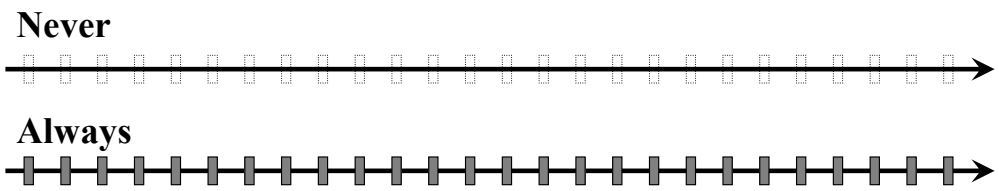

Figure 4.2: Examples of baseline enforcement policies. Hollow rectangles represent contracts not checked when the method is called while solid ones represent checked contracts.

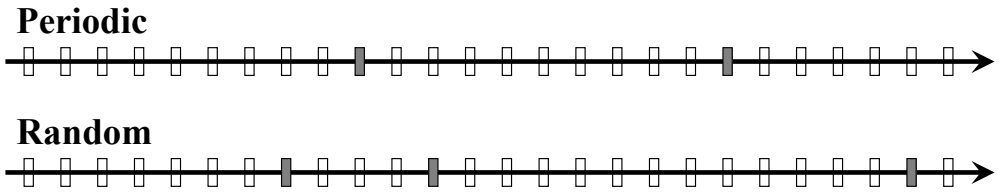

Figure 4.3: Examples of basic sampling policies, where a 10% sampling rate is shown. Hollow rectangles represent contracts not checked when the method is called while solid ones represent checked contracts.

during execution are checked. Consequently, the policy provides baseline data for the total number of checked contracts and detected violations for each trial.

All of the studies include experiments using two simple sampling techniques: *Periodic* and *Random*. Examples of contracts checked with a 10% sampling rate are shown in Figure 4.3. Although the application of the strategies varies between studies, both policies are grounded in the notion of enforcing a subset of contracts sampled on an interval or window basis. Given an interval, n, the *Periodic* policy checks the n^{th} contract encountered during execution. The *Random* policy checks a random contract within a user-specified window of size n. These techniques, therefore, serve as strategies that control the sampling rate but ignore the execution time impact of the contracts they enforce.

Similarly, different traditional contract enforcement policies, discussed in Section 2.4, selectively check contract clauses regardless of execution time effects. The Babel infrastructure changes for the global studies support similar enforcement strategies, whereby only preconditions, postcondition, or invariants may be checked. The Babel compiler is also leveraged to infer characteristics of the assertions within the clauses to establish additional strategies. As discussed further in Section 6.1.1, these characteristics-based policies check clauses based on complexity or the presence of method calls. The purpose for these policies is to provide data on the type of work being performed within contract clauses

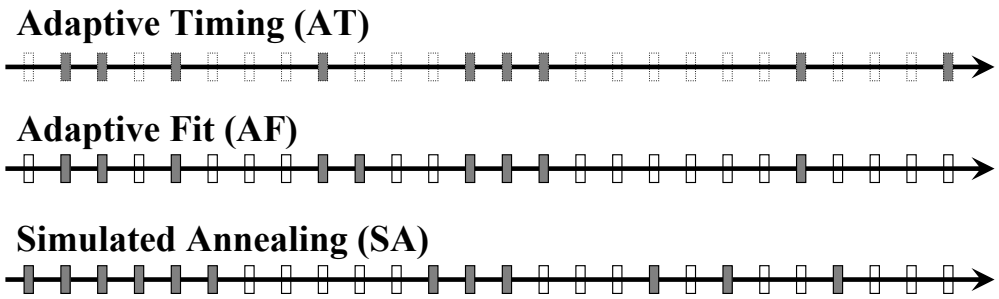

Figure 4.4: Examples of performance-driven enforcement policies. Hollow rectangles represent contracts not checked when the method is called while solid ones represent checked contracts.

exercised by a program.

This research introduces the three performance-driven enforcement policies illustrated in Figure 4.4. *Adaptive timing*, introduced in the first study, checks only those contracts whose execution time estimates are within a user-specified overhead limit. *Adaptive fit* and *Simulated annealing*, introduced in the global enforcement studies, conform more closely with enforcement approach discussed in Section 1.3. The *Adaptive fit* 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. The *Simulated annealing* strategy is essentially *Adaptive fit* with the accumulated enforcement estimate allowed to exceed the user-specified limit with decreasing probability over time. Each study, therefore, includes at least one performance-driven enforcement.

Enforcement policies reflect different strategies for checking interface contracts at run-time. "All or nothing" policies provide baseline data for the metrics used to compare the effects of contract sampling strategies. The two basic sampling approaches control the sampling rate but ignore the execution time costs of contract checks. Similarly, traditional and characteristics-based contract clause enforcement policies selectively check clauses regardless of execution impacts. Performance-driven enforcement policies, on the other hand, are guided by the relative execution time costs of checking contracts.

4.3 Metrics

As introduced in Section 1.4.2, three metrics are used to compare the effects of the policies under study: enforcement overhead, number of interface contracts checked, and number of interface contract violations detected. *Enforcement overhead* is the percentage 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. The metric for total interface contracts checked is computed based on the number checked for a policy relative to the total number checked using the *Always* policy. The number of detected violations for a policy is relative to the total number of violations detected with the *Always* policy. Hence, the effects of sampling contracts are considered in terms of relative execution time, checked contracts, and detected violations.

4.4 Interface Contract Enforcement Studies

This research involves three studies investigating the impacts of performance-driven interface contract enforcement on execution time, contracts checked, and violations detected. A different approach is taken in each case to obtain execution time estimates. The first study relies on enforcement decisions made on a local, or per-method, basis using runtime timing. The remaining two studies make enforcement decisions on a global, or program, basis using *a priori* execution times obtained through different techniques.

4.4.1 Local Study

The initial focus of this research is to determine if performance-driven enforcement can check more interface contracts and, therefore, detect more violations than simple sampling strategies. Enforcement experiments are conducted for each trial using all policies under study. Data gathered during experiments are used to compute the three metrics of interest.

This study conducts enforcement experiments using five policies. As discussed in Section 4.2, policies are grouped into sets of baselines, basic sampling, and performance-

driven sampling. Both baseline policies — *Always* and *Never* — provide measures for computing relative metrics. The simple sampling strategies — *Random* and *Periodic* — are intended to reflect related work. Finally, the *Adaptive timing* policy is introduced as the first performance-driven sampling strategy.

Measures for each policy are taken to facilitate making or tracking enforcement decisions. As discussed in Section 3.2, only experiments using the *Never* policy by-pass the instrumented middleware. Consequently, timestamps taken in the programs provide the only source of data for that policy; namely, baseline execution time measures. The instrumented middleware maintains enforcement measures for the remaining policies. Counters track, for each method, the numbers of contracts encountered, checked, and violated during execution. While execution times are taken in each program, the instrumented middleware tracks enforcement measures on a per-method basis.

Experiments are conducted using a combination of programs and input sets. Three mesh traversal programs are run with each of five input files and up to nine input array sizes. Combining programs and input sets provides a variety of processing in terms of numbers of contracts available for sampling. Also, varying input sets result in different times spent executing statements within methods and checking some assertions.

Results indicate performance-driven enforcement based on runtime timing instrumentation can be used to automatically tailor the enforcement level to the program. Further, the increased levels of contract checking in these trials enable increased detection of violations over the other sampling techniques. However, the impact of the timing instrumentation on enforcement overhead is a concern. The ability of per-method enforcement decisions to effectively manage enforcement overhead across interfaces is also an issue. In addition, questions arise about the nature of the contracts actually checked for each trial.

4.4.2 Global Simple Study

The second study addresses the runtime timing issue by taking a two step approach. Baseline timing runs are used to obtain execution time data in the first step. Then enforcement experiments are performed using execution time estimates derived from the initial baseline data. The primary focus is on determining if an approach making global

decisions using *a priori* execution costs is able to better tune the level of contract (clause) checking to the program. Characteristics of clauses actually checked are also measured. Experiments are repeated with each policy on the same ninety-five trials used in the first study.

The number of policies expands considerably, in large part due to the shift, discussed in Section 4.2, from making decisions on a contract to a clause basis. Traditional interface contract enforcement strategies, such as *Preconditions* and *Postconditions* only, are used. Two additional performance-driven enforcement strategies — *Adaptive fit* and *Simulated annealing* — are added to consider alternative approaches based on execution costs. Characteristics of contract clauses are also measured using policies focused on the nature of their assertions. The associated policies are: *Constant-* versus *Linear*-time and *Simple expressions* versus *Method calls*. Results are reported for experiments using each of thirteen policies.

Data are collected for making and tracking enforcement decisions. Execution times are obtained from simple timing experiments using only baseline enforcement policies. The resulting times are used to establish the *a priori* execution time estimates used by all three performance-driven enforcement policies. Program execution times, interface contract clauses encountered, clauses checked, and violations detected are then measured for every policy and trial. The three metrics — enforcement overhead, contract clauses checked, and violations detected — are computed from the resulting data.

Experiments are repeated with each policy on the same ninety-five trials used in the first study. That is, three mesh traversal programs use input sets from a combination of five input files and up to nine input array sizes. Switching to making enforcement decisions on a contract clause basis results in essentially doubling the number of enforcement opportunities for each program.

As in the first study, the level of contract (clause) checking appears better tuned to each program, though not each trial. Unfortunately, performance-driven enforcement policies are unable to detect any violations using the *a priori* execution times. This result raises concerns about the simple technique for obtaining those estimates.

4.4.3 Global Trace Study

The final study investigates the use of execution time estimates obtained from enforcement tracing to establish the *a priori* cost estimates used by the performance-driven enforcement policies. The goal is to determine if refined estimates allow the policies to detect violations. The policies used are the same as those of the previous study. Thirteen trials form the basis for enforcement tracing and execution experiments.

Enforcement experiments are conducted using baseline, traditional interface contract, characteristics-based, simple sampling, and performance-driven sampling policies. Baseline strategies are: *Always* and *Never*. Traditional interface contract enforcement policies are: *Preconditions* and *Postconditions* only. Characteristics-based strategies check clauses based on characteristics of the contained assertions: *Constant*-time, *Linear*-time, *Simple expressions*, and *Method calls*. Basic sampling policies are: *Random* and *Periodic*. Finally, experiments are conducted using the performance-driven sampling strategies: *Adaptive fit*, *Adaptive timing*, and *Simulated annealing*.

Data are collected both during tracing and enforcement experiments. Tracing runs are made using the *Always* policy and are used to collect elapsed time measures between trace initiation (at the beginning of the program) to trace termination (at the end of the program). Measures are taken to collect data on the time spent on program statements, preconditions, method execution, and postconditions. Enforcement experiments accumulate, for each policy: execution time estimates attributed to contracts versus the program and methods; interface contract clauses encountered during execution; clauses checked; and violations detected.

Experiments rely on a total of thirteen trials formed from five programs and up to five input sets, when appropriate. The three programs from the first two studies are re-used in this study but using only the largest input file and, when appropriate, three input array sizes. In this case, input array sizes are chosen to induce the violation detected in the first study. Two test programs are added, one of which uses five different input array sizes to form the final five trials.

Results indicate performance-driven policies based on refined execution time estimates are better able to adjust their level of enforcement to and detect violations in

the trials. General-purpose, performance-driven (global) policies perform as well or better than *Always* while catching significant numbers of violations in 83% of the trials with violations. A savings of at least 8% overhead is achieved in trials involving moderately expensive contracts. Hence, performance-driven policies tend to perform better in trials whose time spent enforcing contracts is at most moderately expensive relative to the time attributed to the programs and methods.

4.5 Summary

This research investigates the impact of performance-driven sampling as a means of reducing the execution time overhead of interface contract enforcement during deployment. Three studies are conducted using the same core nine-step process for: preparing a program; conducting trial experiments with different enforcement strategies; and analyzing and comparing results. Enforcement experiments are performed using baseline, basic sampling, and performance-driven policies. Metrics are computed for each policy relative to their corresponding measures obtained from the baseline policies. Each study uses a different technique for obtaining execution time estimates for guiding enforcement decisions made by one or more performance-driven policy.

Chapter 5

Local Enforcement

This chapter describes a study investigating the impact of performance-driven interface contract sampling based on per-method enforcement decisions and a fast- versus slow-path implementation akin to that of Liblit *et al.* [117, 118]. The study is generally referred to as the *Local Enforcement* study, or simply *Local* study, since decisions are made on a per-method basis. Insights into the performance overhead of three sampling strategies are gained by varying contract and method execution times through trials formed from different input sets on three programs. Results are summarized from two perspectives — input file and then input array size — to illustrate and compare the effects of the sampling strategies, or policies, under study. Trial sets formed by aggregating results for each program and input file keeps the number of entities processed per execution of the program the same for all runs within the set. Aggregating results by program and input array size creates sets of trials with the same numbers of entities processed per method call across all runs. Inclusion of both perspectives provides concise representations of results while facilitating identification of trials associated with patterns in the metrics, especially patterns relating to the detection of contract violations.

5.1 Babel Extensions for Local Enforcement

As discussed in Chapter 3, an experimental version of the Babel language interoperability toolkit is used to generate enforcement routines from interface contracts added

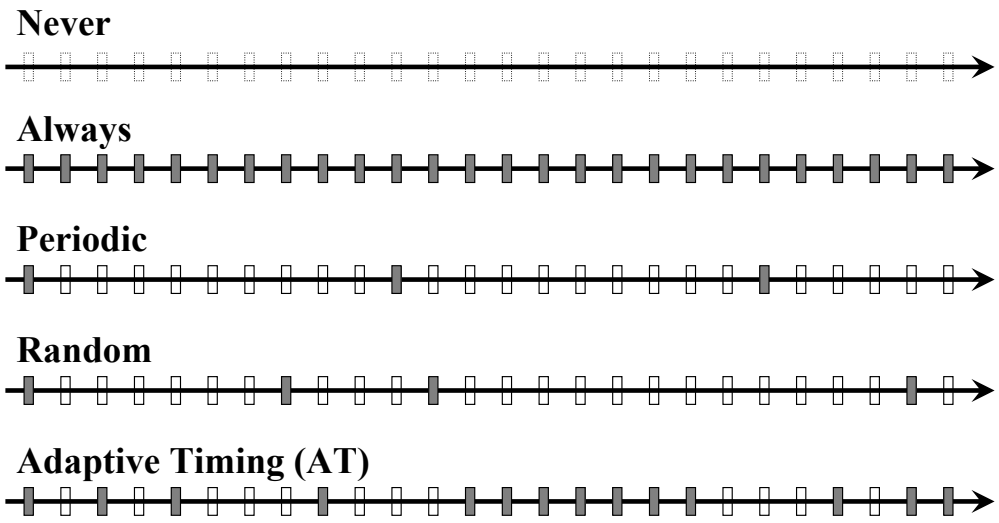

Figure 5.1: Local enforcement policies. Hollow rectangles represent contracts not checked when the method is called while solid ones represent checked contracts.

to each SIDL file. Extensions to Babel for this study consist of modifications to the SIDL runtime library and generated middleware to support enforcement policies.

5.1.1 Enforcement Policies

Five interface contract enforcement policies — illustrated in Figure 5.1 — are implemented for this study, four of which follow the enforcement path shown in Figure 3.3. The policies are: *Never*, *Always*, *Periodic*, *Random*, and *Adaptive timing (AT)*. As discussed in Section 4.2, *Never* and *Always* are baseline policies for disabling enforcement and checking all contracts, respectively. *Periodic* and *Random* reflect simple sampling strategies. Finally, *Adaptive timing* uses performance overhead to drive contract sampling and, therefore, enforcement.

Each sampling strategy requires a user-supplied option. The *Periodic* and *Random* policies are based on an explicit sampling rate. Sampling with the *Adaptive timing* policy, however, is based on an enforcement overhead limit specified as a percentage of the amount of time spent executing methods. *Adaptive timing* relies on time stamps surrounding contract clauses to obtain execution time data for calculating the amount of time spent checking contracts versus executing the method call. Time stamps are used instead of faster, platform-specific counters, for example, due to portability issues. That is, the generated Babel middleware needs to use features available on a variety of hardware platforms

```
if (countdown > 1) {      /* Fast path skips enforcement */
    decrement countdown;
    invoke the method's skel_method;
} else {                  /* Slow path enforces contracts */
    check preconditions;
    check invariants;
    invoke the method's skel_method;
    check postconditions;
    check invariants;
    reset the countdown per the enforcement policy;
}
```

Figure 5.2: Pseudocode for fast and slow paths within enforcement routines. Checks are generated only when the corresponding clause is present in the specification. Contract clauses are enforced and the `skel_method` invoked as long as exceptions are not raised.

of interest to the Scientific Computing community.

All sampling policies check interface contracts on the first call to a method. Contract checking on subsequent calls is determined based on the value of the option associated with the policy. All policies default to the behavior of *Always* once an error occurs. The *Adaptive timing* policy also defaults to the *Always* policy behavior if the amount of time to check the method's contracts is found to be under one microsecond. The rationale behind this policy-specific adaptation is the amount of time attributed to checking the contract is so small that it is not worth the overhead of additional runtime timing checks on subsequent calls.

This study investigates the impacts of interface contract enforcement using two baseline policies and three sampling strategies. Two of the sampling policies rely on traditional periodic and random sampling techniques. The final sampling policy, *Adaptive timing*, is the only performance-driven policy in this study.

5.1.2 Enforcement Routines

As discussed in Section 3.2, the experimental Babel compiler generates separate enforcement routines for each method in a specification containing interface contracts. The enforcement routines use a countdown and code duplication of the invocation of the equivalent method in the skeleton layer to reduce the impact of contract sampling on performance. As shown in the pseudocode in Figure 5.2, the result is a fast path through

the routine when contracts are not enforced and a slow path when they are checked. The method-specific countdown is decremented on each call until the value reaches zero, whereupon the countdown is reset after the contracts are enforced. It is important to point out the value used to reset the countdown for the *Adaptive timing* policy depends on the execution times computed from the time stamps. For example, if a method executes in 10 microseconds, its contracts checked in 5 microseconds, and the specified overhead limit is 5%, then the method's countdown is set to 10 so 5 microseconds out of every 105 microseconds are spent checking contracts. As a result of this fast- versus slow-path approach, the program only incurs the overhead of a countdown check to determine if contracts should be enforced.

5.1.3 Review

In addition to the common extensions to the Babel toolkit described in Chapter 3, study-specific modifications consist of the set of enforcement policies and the contents of generated enforcement routines. Three of the five enforcement policies involve sampling of interface contracts. One of the three policies — *Adaptive timing* — attempts to keep the overhead of interface contract enforcement below a user-specified limit using method and contract execution times obtained through runtime timing instrumentation. Babel-generated enforcement routines consist of a slow execution path essentially wrapping the implementation of a method with its contract checks, and a fast path skipping contract checks. The path taken is based on a method-specific countdown whose value is determined by the policy in affect.

5.2 Subjects

The subjects of this study are three mesh entity retrieval programs. All three programs share the same component, which is a partially compliant implementation of a community-developed mesh data management standard. This section starts with background information on meshes before elaborating on the component and programs.

A mesh is a data structure representing a geometric model. The basic building blocks of a mesh are entities, which are vertices (0-dimensional), edges (1-dimensional),

Table 5.1: Subjects for the *Local* study consist of three programs utilizing different mesh entity query methods. The pseudocode describes operations on mesh data loaded from an input file. Work sets are created based on a specified (input array) size.

Component	Program	Pseudocode

Simplicial Mesh	**A**	initialize the work set iterator over faces;
		while not done {
		  get next work set;
		}
		destroy the work set iterator;

	AA	initialize the work set iterator over faces;
		while not done {
		  get next work set;
		  get vertices adjacent to the faces in the work set;
		}
		destroy the work set iterator;

	MA	get all faces in the mesh;
		for each face {
		  get vertices adjacent to the face;
		}

faces (2-dimensional), and regions (3-dimensional). Entities may be arranged topologically as points, line segments, polygons, triangles, quadrilaterals, polyhedrons, tetrahedrons, hexahedrons, prisms, pyramids, or septahedrons. For example, a triangular face is defined by its three vertices.

The component is a simple, two-dimensional, simplicial mesh implementation providing basic data management methods. A simplicial mesh is an unstructured grid consisting of simplex elements. In this case, the elements are triangles formed from three points in the two-dimensional space. Data sets defining the structure of mesh elements are loaded from input files. The experiments focus on query operations to primarily retrieve the triangular faces — individually as well as through arrays.

The (original) purpose of the component is to serve as an early demonstration of and basis for evaluating the performance costs of mesh management components adhering to a common interface standard defined in SIDL [126]. Version 0.5.1 of the Terascale Simulation Tools and Technologies (TSTT) Center's specification defines a wide range of data management methods. The specification identifies methods for operations on the full mesh, arrays (or work sets) of mesh entities, and individual entities. More information on the specification can be found in [174].

Table 5.1 lists the three programs and provides the pseudocode describing the operations taken after mesh data is loaded from an input file. Program **A** simply "traverses" mesh faces and serves as a "worst-case" scenario for constant-time assertions. Program **AA** expands on program **A** by adding the retrieval of adjacent faces within the loop. In doing so, the additional query adds a contract containing additional constant-time plus two linear-time assertions. (The assertions are linear in the output array from the first call in the loop.) So this program enforces contracts containing both constant- and linear-time assertions within the loop. Finally, program **MA** retains the adjacency retrieval call of program **AA** and its associated linear-time contract. However, all face entities are retrieved in a single call prior to the start of the loop. Each adjacency call is then made to retrieve the associated vertices, in turn, for each face entity. So the tight loop in this case enforces a contract with linear-time assertions.

Hence, three simple mesh traversal programs are used as subjects in this study. Each program retrieves different combinations of entities from a simple, two-dimensional, simplicial mesh component that implements a community-developed mesh data management standard. The programs support investigation of performance scenarios involving tight loops on combinations of constant- and/or linear-time assertions.

5.3 Trials

The criteria used to form trials for this study are illustrated in Figure 5.3. The three dimensions are: contract complexity, input file size, and input array size. Contract complexity is based on arguments referenced by method calls within contract clauses. A combination of programs involving different interface contract characteristics and input sets are used to satisfy the criteria.

Contract complexity is addressed through the use of three programs iterating over mesh entities. Specifications of the methods called by one or more of the programs are given in Figure 5.4. Names of methods whose execution times are linear in the size of an input argument are italicized when they appear within a contract clause. Assertions violated during the execution of one or more programs are indicated using boldface type.

Input sets are formed from different combinations of input files and, when possible,

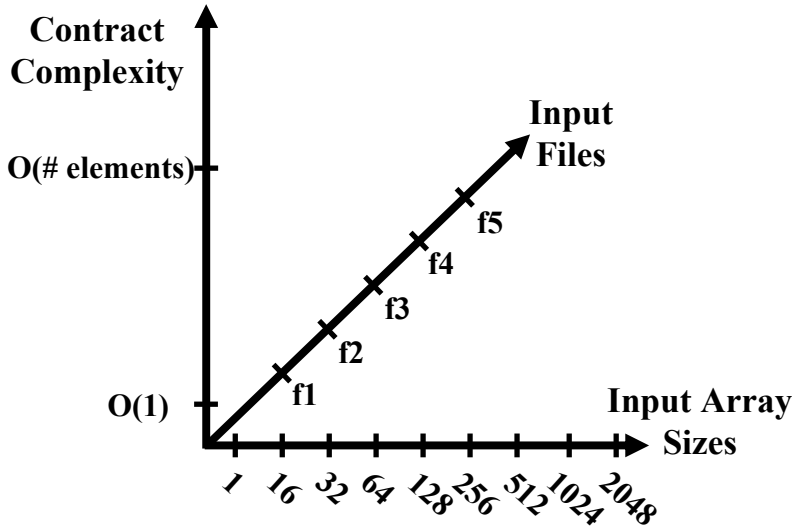

Figure 5.3: Criteria for establishing trials. Trials are formed from a combination of programs – with varying contract complexity – and input sets. Contract complexity refers to the complexity of assertions on arguments to methods. Combinations of increasingly larger input files and, when appropriate, input array sizes establish input sets for programs.

input array sizes. Five increasingly larger input files, ranging in size from 13,464 to 145,870 face entities, allow the number of mesh retrievals and total amount of processing time to vary. Nine input array sizes indirectly impact the size of output arguments checked in the linear-time assertions, affect the number of retrieval method calls performed, and vary the amount of work done within methods.

These factors have several impacts on interface contract enforcement. The more mesh retrieval calls made during program execution, the more contracts are available for sampling and the more time is expended making enforcement decisions. Increasing the amount of work within methods generally increases the amount of time spent in the methods, which helps offset the time spent checking the method's contracts. Finally, larger input arrays can translate into more time spent checking contracts linear in the size of the associated output array arguments.

In all, ninety-five trials are formed by combining each program with an input file and, when applicable, nine input array sizes. The complexities of contracts checked by a program depend on the methods it invokes. Input file sizes are varied to increase the numbers of method calls for each program. Different input array sizes are also used for two of the programs to impact the number of method calls and the amount of time spent within

```
void entitysetInitializeWorksetIterator (in opaque entity_set_handle,
      in EntityType requested_entity_type, in EntityTopology
      requested_entity_topology, in int requested_workset_size,
      out opaque workset_iterator) throws Error;
   require
      requested_workset_size > 0;
      validTypeNTopo(requested_entity_type, requested_entity_topology);
   ensure
      workset_iterator ! = null;

bool entitysetGetNextWorkset (inout opaque workset_iterator, inout
      array<opaque> entity_handles) throws Error;
   require
      workset_iterator ! = null;
   ensure
      workset_iterator ! = null;
      result implies (entity_handles ! = null);
      (entity_handles ! = null) implies (dimen(entity_handles) == 1);

void entitysetDestroyWorksetIterator(in opaque workset_iterator)
      throws Error;

void entitysetGetEntities (in opaque entity_set_handle, in EntityType
      entity_type, in EntityTopology entity_topology, inout
      array<opaque> entity_handles) throws Error;
   require
      validTypeNTopo(entity_type, entity_topology);
   ensure
      (entity_handles ! = null) implies (dimen(entity_handles) == 1);
      (entity_handles ! = null) implies (size(entity_handles) >= 0);

void entityGetAdjacencies (in array<opaque> entity_handles, in EntityType
      entity_type_requested, inout array<opaque> adj_entity_handles,
      inout array<int> offset) throws Error;
   require
      (entity_handles ! = null) implies (dimen(entity_handles) == 1);
   ensure
      (adj_entity_handles ! = null) implies (dimen(adj_entity_handles) == 1);
      (adj_entity_handles ! = null) implies (offset ! = null);
      (offset ! = null) implies (dimen(offset) == 1);
      (offset ! = null) implies (size(offset) == (size(entity_handles) + 1));
      (offset ! = null) implies irange(offset, 0, size(adj_entity_handles));
      (offset ! = null) implies nonDecr(offset);
```

Figure 5.4: Interface specifications for methods invoked in the local study. Linear-time assertions appear in italics and violated assertions in boldface type. Work sets are simply arrays of mesh entities.

methods. In the case of program **AA**, varying input array sizes also varies the amount of time spent checking contracts with linear-time assertions.

5.4 Methodology

The basic process for conducting experiments is described in Section 4.1. The relevant points for this study are the options used by the sampling policies and the number of iterations on step 6 of Figure 4.1. The *Periodic* policy is set to a 1% sampling rate. The *Random* policy is set to a sampling rate of 2%. The 10% rule-of-thumb for scientific computing is used for *Adaptive timing*. Finally, each experiment is repeated thirty times to obtain a reasonable average of the execution times.

Results are aggregated and reported using two perspectives: input file and input array size. Aggregation of program results by input file provides a view into the data where the number of entities processed per execution is fixed. The view created by aggregating results by input array size fixes the number of entities processed per method call. Inclusion of both perspectives provides concise representations of results that facilitate identification of trials associated with patterns in the metrics, especially patterns relating to the detection of contract violations.

5.5 Results by Input File

Data for the ninety-five trials are aggregated on the basis of program and input file, forming the fifteen trial sets listed in Table 5.2. Creating trial sets in this manner provides a view into the data based on consistent numbers of entities across all trials within a given set. As described in Section 1.4.2, the three metrics of concern are: enforcement overhead, number of interface contracts checked, and number of violations detected. Since enforcement overhead is relative to the *Never* policy, only results for policies with enforcement enabled are presented. Trials appear in order of increasing mean enforcement overhead using the *Always* policy.

Figure 5.5 illustrates the three metrics for the *Always* policy. The range of mean contracts checked is 1,684 to 145,870. A total of 30 violations are detected for each trial set

Table 5.2: Trial sets by input file for the *Local* study, where *NA* indicates the input option does not apply. Input files are numbered by size from smallest (**f1**) to largest (**f5**), with the size of each file — in numbers of face entities — shown within parentheses.

Trial Set	**Program**	**Input Set(s)**	
		Input Array Sizes	**Input File (Size)**
A-f1	A	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f1 (13,464)
A-f2	A	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f2 (23,751)
A-f3	A	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f3 (52,722)
A-f4	A	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f4 (93,496)
A-f5	A	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f5 (145,870)
AA-f1	AA	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f1 (13,464)
AA-f2	AA	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f2 (23,751)
AA-f3	AA	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f3 (52,722)
AA-f4	AA	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f4 (93,496)
AA-f5	AA	1, 16, 32, 64, 128, 256, 512, 1024, & 2048	f5 (145,870)
MA-f1	MA	*NA*	f1 (13,464)
MA-f2	MA	*NA*	f2 (23,751)
MA-f3	MA	*NA*	f3 (52,722)
MA-f4	MA	*NA*	f4 (93,496)
MA-f5	MA	*NA*	f5 (145,870)

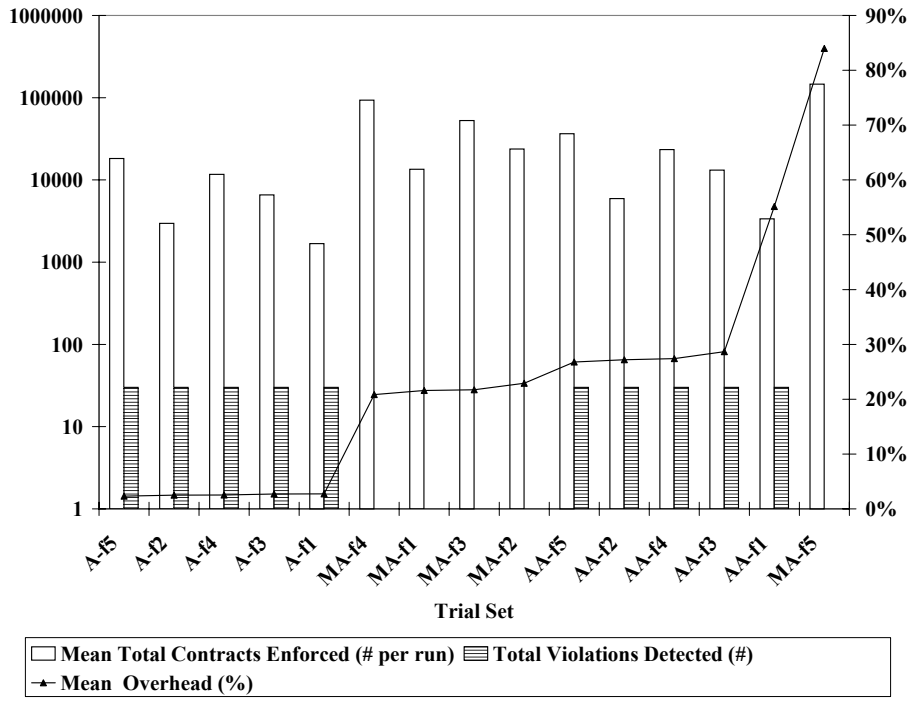

Figure 5.5: *Local* study results by input file for the *Always* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program (**A**, **MA**, and **AA**) with the input file (**f1**...**f5**). For example, **A-f5** is the trial set formed from results for program **A** and input file **f5**.

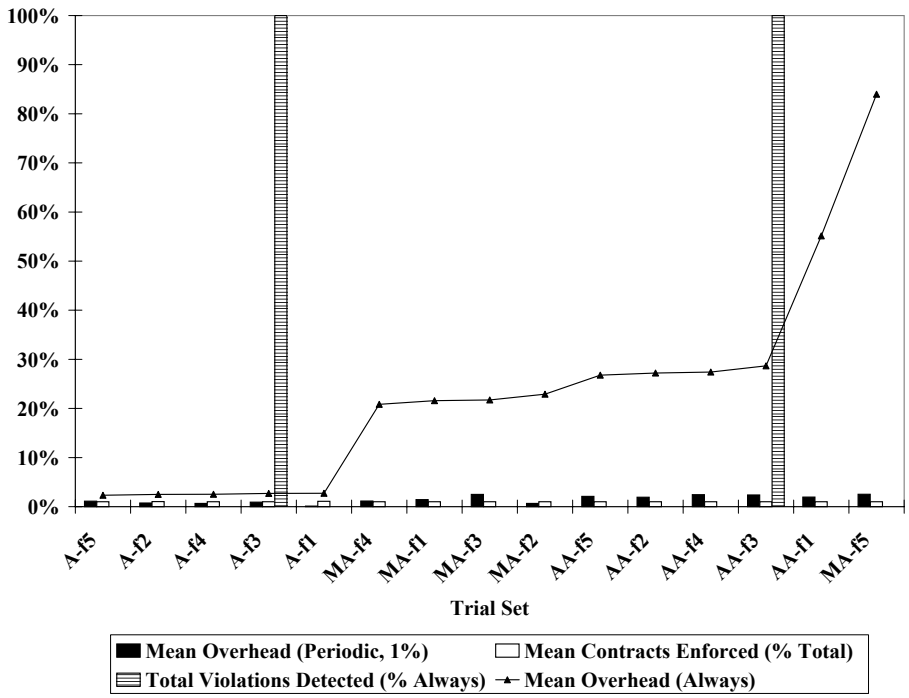

Figure 5.6: *Local* study results by input file for the *Periodic* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input file as described in the caption of Figure 5.5.

associated with programs **A** and **AA**. Program **A** incurs 3% or less enforcement overhead regardless of input file, while the overhead for trial sets involving programs **MA** and **AA** generally ranges from 21% to 29%. So it seems the overhead of checking contracts varies based on the presence or absence of linear-time assertion checks.

Results for the *Periodic* policy are presented in Figure 5.6. The enforcement overhead ranges from 0% to 3% across trial sets. The mean numbers of contracts enforced match the sampling rate across trial sets. Interestingly, all violations are detected for trial sets **A-f3** and **AA-f3**. Since a separate countdown is maintained for each method, the policy is able to check a method's contracts at the same interval regardless of the program. In these cases, the policy detects the violation in the `entitysetGetNextWorkset` method for all executions of the trial sets involving programs **A** and **AA** when run on the same input set. So the policy consistently identifies the violation as a result of sampling the contract clause under the same circumstances for every execution of the relevant trials.

Figure 5.7 illustrates the results for the *Random* policy. The mean overheads range from 1% to 4% despite the 2% sampling rate. These effects occur as a result of the

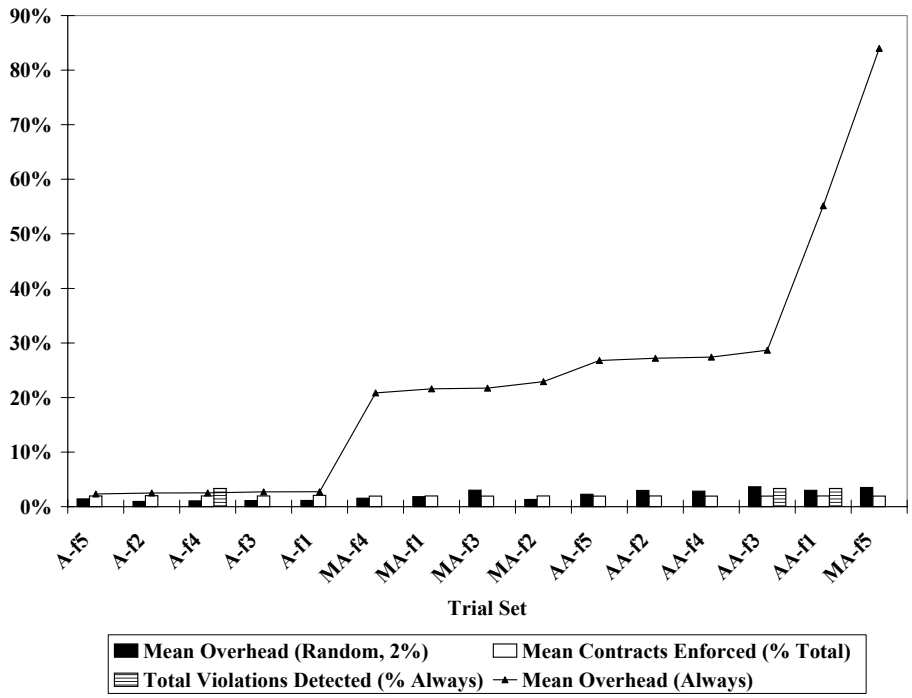

Figure 5.7: *Local* study results by input file for the *Random* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input file as described in the caption of Figure 5.5.

low numbers of sampling opportunities for some input sets. Three trial sets had 3% of their violations detected, which translates into one violation per trial set.

Finally, results for the *Adaptive timing* policy are illustrated in Figure 5.8. Trial sets involving program **A** incur between 16% and 21% overhead while checking 14% to 33% of the contracts. The increased enforcement leads to 7% to 40% of the violations being detected. At 11% to 14% mean overhead, program **AA**'s trial sets generally incur slightly more than the 10% overhead limit while detecting between 13% and 67% of the violations. With mean overheads ranging from 6% to 7%, *Adaptive timing* performs better for program **MA** trial sets. However, the lower overhead corresponds to actual sampling rates of between 2% to 3% of enforced contracts and a program with no contract violations.

The following is a summary of the above observations from Figures 5.5 through 5.8. Aggregating data for the ninety-five trials into fifteen sets based on the program and input file establishes sets with consistent numbers of entities yet different execution profiles (due to varying input array sizes). The *Always* policy incurs minor average overhead with program **A** trials, which check contracts consisting solely of constant-time assertions. Recall

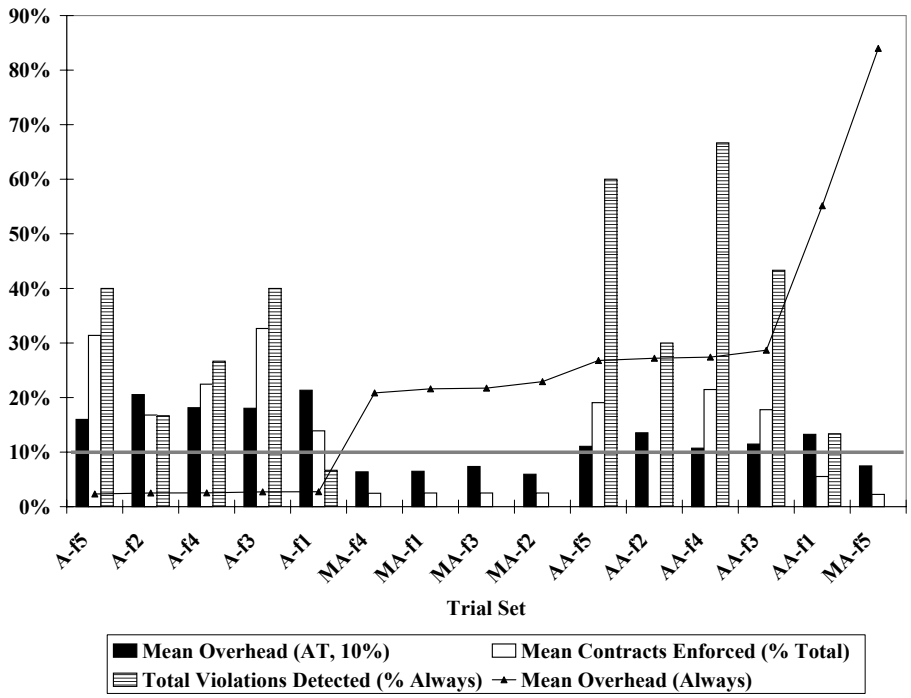

Figure 5.8: *Local* study results by input file for the *Adaptive timing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input file as described in the caption of Figure 5.5.

from Section 5.1.1, all sampling policies check contracts on the first call to every method. So the *Periodic* and *Random* policies check both constant- and linear-time contracts yet incur relatively low overhead — at low sampling rates. However, the two policies are unable to consistently detect the numbers of violations found using the *Adaptive timing* policy. While *Adaptive timing* detects violations in all ten trials where they occur, it does so by exceeding the 10% overhead limit. It appears checking methods with linear-time assertions in their contracts tends to mitigate the enforcement overhead of the *Adaptive timing* policy. So it seems the *Always* policy should be used for programs involving only constant-time assertions; while the *Adaptive timing* policy is more suited to programs that include contracts with lots of linear-time assertions.

5.6 Results by Input Array Size

Data for the ninety-five trials are aggregated on the basis of program and input array size, forming the nineteen trial sets listed in Table 5.3. Each set provides a view into

Table 5.3: Trial sets by input array size for the *Local* study, where *NA* indicates the input option does not apply. Input files are numbered in order from smallest (**f1**) to largest (**f5**).

Trial Set	**Program**	**Input Sets**	
		Input Array Size	**Input Files**
A-1	A	1	f1, f2, f3, f4, and f5
A-16	A	16	f1, f2, f3, f4, and f5
A-32	A	32	f1, f2, f3, f4, and f5
A-64	A	64	f1, f2, f3, f4, and f5
A-128	A	128	f1, f2, f3, f4, and f5
A-256	A	256	f1, f2, f3, f4, and f5
A-512	A	512	f1, f2, f3, f4, and f5
A-1024	A	1024	f1, f2, f3, f4, and f5
A-2048	A	2048	f1, f2, f3, f4, and f5
AA-1	AA	1	f1, f2, f3, f4, and f5
AA-16	AA	16	f1, f2, f3, f4, and f5
AA-32	AA	32	f1, f2, f3, f4, and f5
AA-64	AA	64	f1, f2, f3, f4, and f5
AA-128	AA	128	f1, f2, f3, f4, and f5
AA-256	AA	256	f1, f2, f3, f4, and f5
AA-512	AA	512	f1, f2, f3, f4, and f5
AA-1024	AA	1024	f1, f2, f3, f4, and f5
AA-2048	AA	2048	f1, f2, f3, f4, and f5
MA	MA	*NA*	f1, f2, f3, f4, and f5

the data based on consistent numbers of face entities in input array arguments. Enforcement overhead, number of interface contracts checked, and number of violations detected are reported for each of the nineteen trial sets on an enforcement policy basis. Results appear in order by the enforcement overhead of the *Always* policy.

Figure 5.9 illustrates the *Always* enforcement policy metrics. The mean range of contract checks is from 34 to 87,816 per trial execution. All 150 violations occur only in trial sets **A-1** and **AA-1**, which means there are 30 violations per input file for each of the two trail sets. The mean enforcement overhead is negligible to low for seven of the nine trial sets using program **A**. The introduction of a method with linear-time assertions in its postcondition clause leads to a significant jump in the mean overhead for program **AA**'s trial sets.

Results for the *Periodic* policy are shown in Figure 5.10. Mean overhead is negligible for program **A** trial sets on all but the tightest loop (at 3%). Overhead for the remaining trial sets exceeds 3% only for trial set **MA**, where it is 7%. The actual sampling

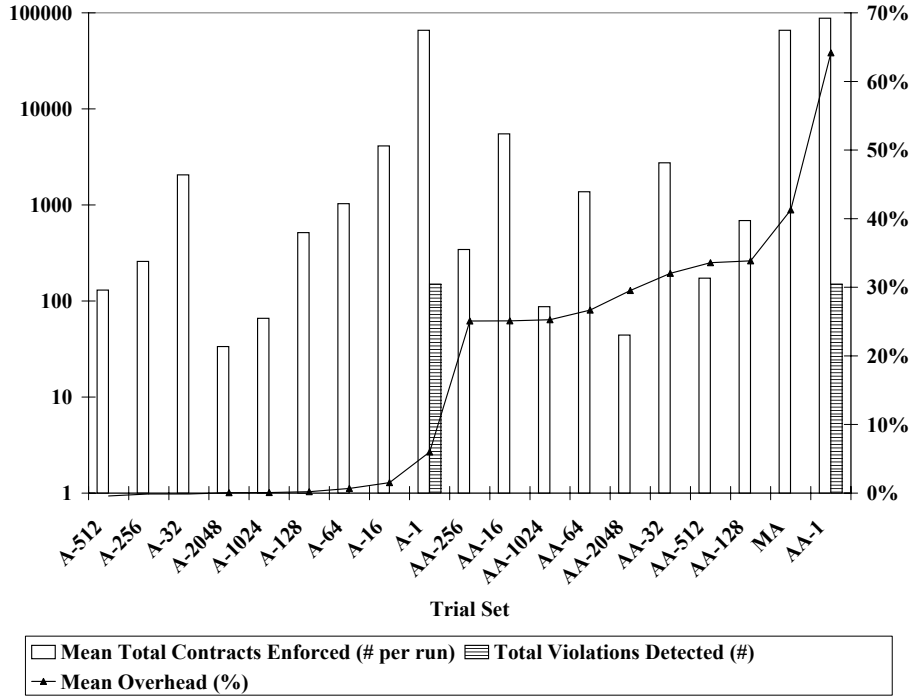

Figure 5.9: *Local* study results by input array size for the *Always* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program (**A**, **MA**, and **AA**) with the input array size (**1**...**2048**). For example, **A-1024** is the trial set formed from results for program **A** and input array size **1024**.

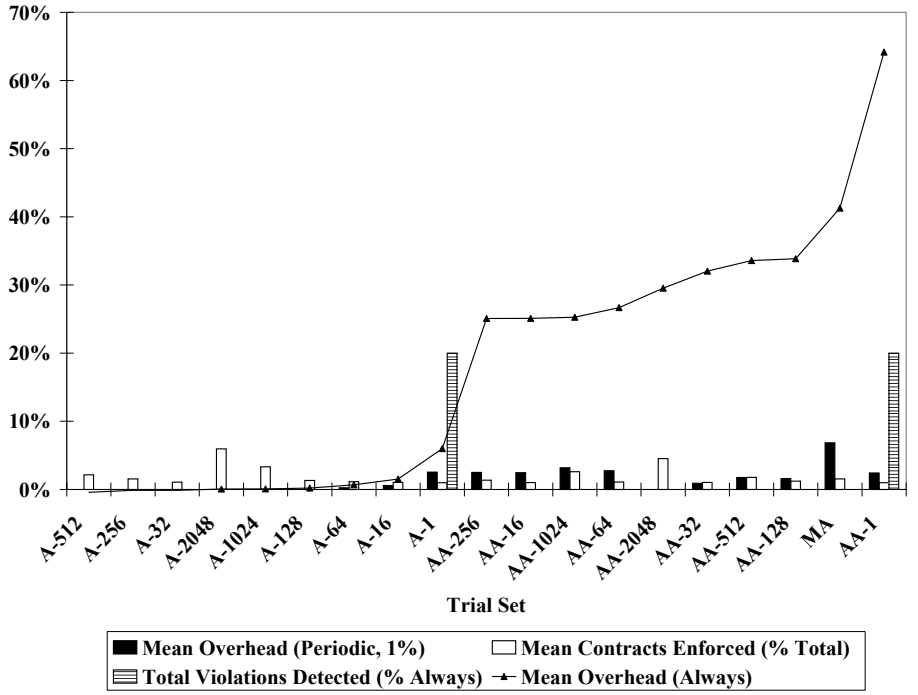

Figure 5.10: *Local* study results by input array size for the *Periodic* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input array size as described in the caption of Figure 5.9.

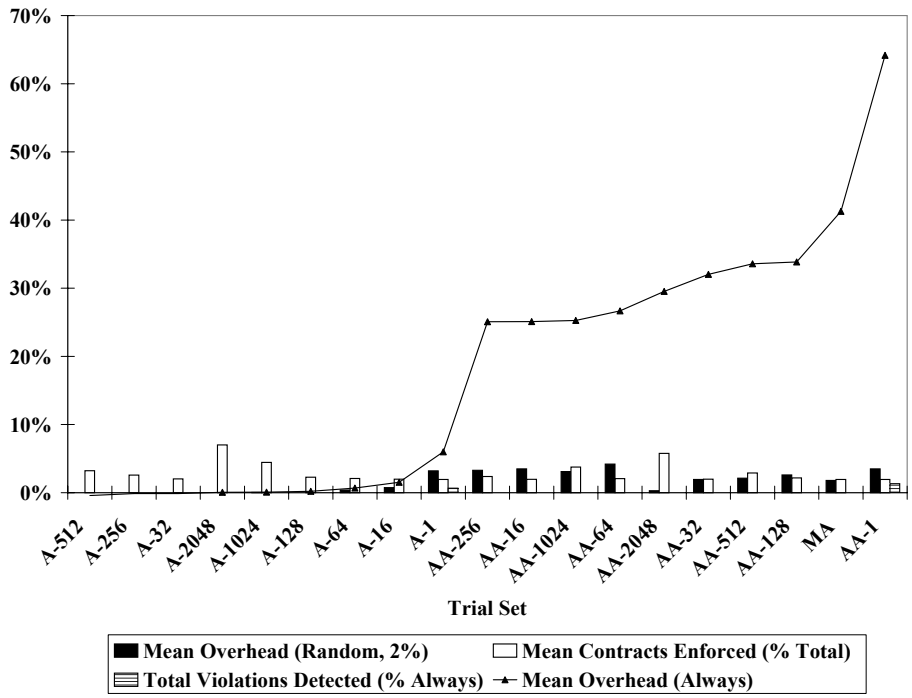

Figure 5.11: *Local* study results by input array size for the *Random* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input array size as described in the caption of Figure 5.9.

rate exceeds the specified rate as the size of the input array increases since the numbers of opportunities to check contracts decreases. For example, the 6% mean contracts enforced for trial set **A-2048** corresponds to a mean of two contracts checked per run. Finally, 20% of the violations are detected each for trial sets **A-1** and **AA-1**. This corresponds to the trial sets using input file **f3** discussed in Section 5.5.

The mean overhead and contract enforcement results are similar for the *Random* policy as illustrated in Figure 5.11. The mean overhead is relatively low across trials, though it does not exceed 4% in this case. Mean contracts enforced varied from the sampling rate of 2% to a high of 7% for trial sets with the largest input array size. However, once again, the actual rate translates into a little more than an average of two contracts checked per run. At 1% each, the number of violations detected for trials **A-1** and **AA-1** are very low though the rate translates into 1 and 2 violations, respectively.

Figure 5.12 illustrates the results for the *Adaptive timing* policy. Recall each method's countdown is initially set based on runtime timing instrumentation on its first invocation. The six trial sets with the lowest enforcement overhead rate using the *Always*

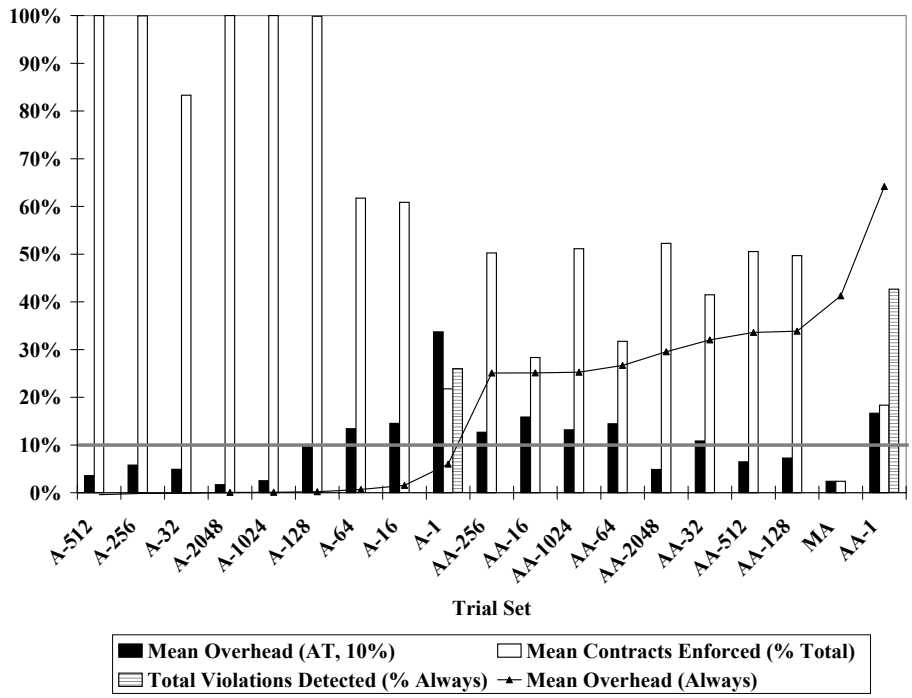

Figure 5.12: *Local* study results by input array size for the *Adaptive timing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by concatenating the program and input array size as described in the caption of Figure 5.9.

policy incur less than the 10% overhead rate with *Adaptive timing* and, at 83% to 100%, check the most contracts. With a few exceptions, trials involving contracts with linear-time assertions generally incur more than the specified 10% overhead while checking 28% to 52% of the contracts. The additional enforcement pay off in trial sets *A-1* and *AA-1* show 26% and 43% of the violations are detected, respectively. However, both trial sets experienced the worst overhead. One possible reason for this behavior may be due to violations being logged in an output file. Given the two trial sets involve tight loops, the runtime timing combined with writing to a file could negatively impact the relative overhead of enforcement.

As in Section 5.5, there is significant variation in mean overhead based on the presence or absence of linear-time assertions in contracts checked by the *Always* policy. The mean overhead is negligible in the six trial sets formed from program **A** having the largest input array sizes. On the other hand, trial sets involving the tightest loops (i.e., one entity per input array) incur the highest overhead for their programs. *Periodic* and *Random* incur relatively low overhead costs across trials but also check relatively few contracts. The

Adaptive timing policy, on the other hand, does appear to effectively adjust the number of contract checks while staying relatively close to the user-specified limit. The biggest exceptions to the overhead limit are the trial sets with a single input entity processed per iteration of the loop for programs **A** and **AA**. These trial sets also detect the most violations, which may incur extra overhead associated with logging the contract violation to a file. The presence of linear-time assertions does appear to mitigate the overhead to some extent.

5.7 Discussion

The presentation of results on aspects of the input sets provides two views into the data. Section 5.5 includes the initial observations regarding patterns in the results, while Section 5.6 helps refine those descriptions. Both views also reflect several factors influencing the metrics used to compare policies. This section summarizes the results across all enforcement experiments and identifies factors affecting metrics.

5.7.1 Overall Results

Overall results across enforcement experiments are presented in Table 5.4. The three metrics — enforcement overhead, contracts enforced, and violations detected — are supplemented by results specific to linear-time contracts. The first such column, *Checked Linear*, lists the mean percentage of checks whose contracts include linear-time assertions. Values are based on the numbers of calls made to `entityGetAdjacencies` since it is the only such method exercised in this study. The *Linear Checked* column provides the mean percentage of those contracts actually enforced during execution. The following discussion focuses primarily on baseline and performance-driven policies.

Violations are detected for only two programs: **A** and **AA**. As highlighted in Figure 5.4, the violation occurs in the postcondition clause of the last call to the `entitysetGetNextWorkset` method for trials using a single entity per input array. The violation occurs because the component returns `true` and the middleware sets the empty `entity_handles` array (pointer) to `null`. (The violation is a consequence of non-compliance resulting from the programs and component having been developed before the contracts were defined.) A

Table 5.4: Overall *Local* study results. *Checked Linear* reflects the number of contract checks including linear-time assertions. *Linear Checked* is the number of linear-time contract checks actually enforced. *AT* is the *Adaptive timing* policy. *NA* means the metric or value is not applicable. (Note: No contracts are violated during the execution of program **MA**.)

Program	Policy	Mean				
		Enf. Overhead	**Contracts Enforced**	**Vio. Detected**	**Checked Linear**	**Linear Checked**
A	*Always*	2%	100%	100%	*NA*	*NA*
	Periodic	1%	1%	20%	*NA*	*NA*
	Random	1%	2%	1%	*NA*	*NA*
	AT	17%	27%	26%	*NA*	*NA*
AA	*Always*	33%	100%	100%	50%	100%
	Periodic	2%	1%	20%	50%	2%
	Random	3%	2%	1%	50%	1%
	AT	12%	20%	43%	6%	2%
MA	*Always*	41%	100%	0%	100%	100%
	Periodic	2%	2%	0%	100%	1%
	Random	2%	2%	0%	100%	2%
	AT	7%	2%	0%	100%	2%

total of 150 violations — thirty executions per input file — occur for each program.

The *Always* policy enforces all contracts encountered and detects all violations for program **A** with a mean overall enforcement overhead of only 2%. The *Periodic* policy is able to detect 20% of the violations with half the overhead; however, from Sections 5.5 and 5.6, this impressive number is based on detecting all violations for the trial set using input file **f3** and a single entity per input array. The *Adaptive timing* policy enforces 27% of the contracts and detects 26% of the violations, but at a cost of 17% overhead. The overhead is considerably higher with *Adaptive timing* due to the execution of the four `gettimeofday` system calls surrounding the pre- and post-condition clauses of the method and the corresponding calculation of a new countdown. Program **A** does not check any contracts containing linear-time assertions.

The presence of the `entityGetAdjacencies` call within program **AA**'s loop results in an order of magnitude increase in enforcement overhead compared with program **A** — to 33% — using the *Always* policy. The method also results in the inclusion of linear-time assertions in about half the contract checks available for enforcement. The *Periodic* policy again detects the same violations found with program **A**, though at over

twice the overhead. The *Adaptive timing* policy detects an impressive 43% of the violations by checking 20% of the contracts and incurring 12% overhead. Only a mean 6% of the contract checks include linear-time assertions (compared to 50% for the other policies). A mean of 2% of the calls to the `entityGetAdjacencies` method include enforcing the contract. The basis for this reduced level of checking will be discussed shortly.

Program **MA**'s loop is similar to that of program **AA** but without the call to the `entitysetGetNextWorkset` method. That is, the loop of program **MA** is limited to calls to `entityGetAdjacencies`, which is the only method in this study containing linear-time assertions in its contract. Each invocation operates on an array containing a single entity at a time resulting in only a few entries checked in the linear assertions of the postcondition clause. The *Always* policy in this case reaches a mean 41% overhead while almost exclusively enforcing contract checks with linear-time assertions. No contracts are violated during execution of the program.

The *Adaptive timing* policy incurs about 7% overhead for program **MA** while enforcing only 2% of the contract checks. Of the mean 65,861 calls made to component methods, *Adaptive timing* checks 1,569 contracts, all but one of which is to the `entityGetAdjacencies` method. So the mean percentage of linear-time contracts enforced using the policy is about 2.4%, or one contract checked for every forty-two calls to the method.

Further analysis of program **AA** data for the *Adaptive timing* policy shows an interesting pattern in the level of contract enforcement for the `entityGetAdjacencies` method. The number of times the method's contract is checked decreases as the size of the input array increases. This number falls to one (for the first call) per execution starting at input array size **64** for input file **f1**; **128** for input files **f2** and **f3**; and **256** for input files **f4** and **f5**. Since the policy attempts to keep the execution time of the contract within the user-specified overhead limit of the time spent executing the method, in this case 10%, the two linear-time assertions (plus runtime timing instrumentation) cause the contract to be too expensive to check for these and the larger input array sizes.

The opposite relationship occurs for the constant-time contract associated with the `entitysetGetNextWorkset` method. That is, the number of times the method's contract is checked increases as the size of the input array increases. In fact, a mean of 98%

or more contract checks occur with input array sizes **128** through **2048** for both program **A** and **AA** regardless of input file. Therefore, the amount of time spent enforcing `entitysetGetNextWorkset`'s contract together with the runtime timing instrumentation appears to fall at or just under 10% of the time attributed to executing the method when there are about **128** entities per input array. So the *Adaptive timing* policy enforces all contracts consisting solely of constant-time assertions with negligible overhead if there is sufficient work performed in the corresponding method(s) to offset the timing instrumentation.

Results from this study indicate the policy choice should depend on the nature of the contracts enforced by a program. The *Always* policy appears to be most suited to programs calling methods with only constant-time contracts. An exception occurs, according to results reported in Section 5.6, for programs exhibiting execution profiles similar to the tight loop exemplified by trial set **A-1**. The *Adaptive timing* policy tends to enforce more contract checks and detect more violations than the basic sampling policies. This policy is able to maintain those levels while keeping overhead relatively low compared to the *Always* policy. However, the implementation is not able to keep the overhead below the user-specified limit in the presence of large percentages of constant-time contracts, as illustrated in Figure 5.8.

5.7.2 Influential Factors

A review of trial set results for the *Always* policy leads to observations regarding factors appearing to affect metrics. The basic factors are those used to form trial sets: programs, input files, and input array sizes. Additional factors are aspects of the contracts — number enforced, number violated, and complexity — since they are determined by the methods invoked during program execution. This section discusses the apparent influence of each of these six factors on the metrics reported in this study.

Table 5.5 lists the ratings for the affects of factors on the three metrics: enforcement overhead, enforced contracts, and violated contracts. Ratings are determined by simple relations of values, such as clusters of similar values or simple functions, across relevant trial sets. If a relation is apparent in a third or fewer trial sets, the factor is rated as

Table 5.5: Factors affecting performance overhead for the *Always* policy, where ratings are based on the numbers of simple relations between metrics across relevant trial sets. Influence ratings are: *low* for minor or no influence (as determined by effects on a third or fewer trial sets); *moderate* for apparent influence (in terms of one- to two-thirds of the trial sets); and *high* for clear influence (involving more than two-thirds the trial sets). *NA* means the factor is not applicable or insufficient data exists.

Factor	Apparent Influence on...		
	Percent Enforcement Overhead	**Number of Enforced Contracts**	**Number of Violated Contracts**
Program	High	High	Moderate
Input File	Low	High	Moderate
Input Array Size	Low	High	Low
Contracts – Number Enforced	Low	*NA*	*NA*
Contracts – Number Violated	Low	*NA*	*NA*
Contracts – Complexity	High	*NA*	*NA*

low to reflect minor or no influence. Over one-third but under two-thirds of the trial sets with related values is deemed *moderate*. A *high* rating is given to factor that have a clear influence in terms of a relationship in the values of more than two-thirds the trial sets.

Although the programs perform similar operations and have similar structures in terms of invoking component methods within loops, there are noticeable differences in enforcement overhead. An inspection of Figures 5.5 and 5.9 show clusters of overhead ranges for the majority of trial sets involving each program so the rating is *high*. All three programs iterate over simplicial meshes, thereby clearly affecting the numbers of contract checks made during execution. The rating for the impact of the program on the number of violated contracts is set to *moderate* since exactly two-thirds of the programs (and trial sets in Figure 5.5) detect interface contract violations.

Ratings for the influence of the input file on the three metrics are based on results illustrated in Figure 5.5. There is no obvious affect on overhead for trial sets using the same input file. However, the numbers of contract checks clearly depend on the size of the input file since all three programs iterate over meshes defined by the file. The influence rating for violated contracts is *moderate* due to violations being detected in exactly two-thirds of the trial sets for each file.

Input array size ratings are based on results shown in Figure 5.9. In this case, the only clear influence is on the numbers of contract checks. This effect results from the

fact the numbers of iterations over the meshes are determined in part by the size of the input array.

Of the three contract-specific factors — number enforced, number violated, and complexity — only the complexity of contract checks appears to influence enforcement overhead. For example, program **A**, with only constant-time assertions in its contracts, incurs the least overhead regardless of the input set. The remaining trial sets enforce contracts containing linear-time assertions within their loops. As a result, they generally incur significantly more overhead ranging from 21% to 29% (from Figure 5.9).

A review and discussion of the data provides insights into factors affecting enforcement metrics. While the number of contracts enforced and violations detected appear to have little impact on overhead, the same cannot be said for the program or the complexity of the assertions contained within the contracts. Complexity impacts enforcement overhead in terms of the relative amount of time required to check contracts. The more time it takes to check a contract, the less likely it is to be enforced by the performance-driven enforcement policy — unless a sufficient amount of work is done in the method. Since overhead is the criteria driving performance-driven enforcement decisions, the nature of the program and complexity of contracts checked must be given due consideration when trying to determine the most appropriate enforcement policy.

5.8 Summary

Results from a study investigating the impacts of interface contract enforcement using three sampling strategies — *Periodic*, *Random*, and *Adaptive timing* — are presented in this chapter. All three policies check contracts the first time a method is called. However, only the *Adaptive timing* policy factors in the enforcement execution cost by basing decisions on execution times obtained at runtime for methods and their contracts.

The study involves experiments with three programs using different combinations of methods to retrieve sets of entities from a single implementation of a mesh component. Five input files and nine input array sizes are combined with the programs to form a total of ninety-five trials. Results are presented first in sets of trials based on input files then again aggregated in sets based on input array sizes.

The vision for this research, as described in Section 1.3, is to provide an interface contract enforcement alternative to completely disabling enforcement during deployment that takes execution costs into consideration. It is believed the strategy would be most useful for a specific class of programs. As discussed in Section 1.3, this research targets programs making sufficient calls to methods with contracts (to make enforcement sampling worthwhile) and incurring unacceptably high execution time overhead with full contract enforcement.

Findings indicate the *Adaptive timing* policy shows promise for programs with the target attributes. The policy adjusts the level of contract checking to the program based on the user-specified overhead limit. It also consistently detects violations in all trial sets when violations occur in constant-time contracts. However, the *Adaptive timing* policy incurs excessive overhead when numerous calls are made to methods whose contracts consist solely of constant-time assertions and when there is insufficient work performed in the methods to offset the runtime timing instrumentation. The performance effects seem to be mitigated somewhat when an equal number of methods with contracts including linear-time assertions are called.

Chapter 6

Global Enforcement

This chapter covers two studies of global, performance-driven interface contract clause enforcement based on *a priori* execution time estimates. Unlike per-method enforcement decisions in the *Local* study described in Chapter 5, the centralized approach taken here is expected to provide better control over enforcement overhead across methods and components. Sampling decisions are made on a contract clause basis to include support for the traditional interface contract enforcement strategies akin to those described in Section 2.4. Two additional performance-driven strategies are introduced as well. Both studies extend the work flow described in Section 4.1 to include a preliminary pass over phase two for the purposes of obtaining the execution time estimates.

While the two studies share the same set of enforcement policies and the same approach, there are differences. The first study, referred to as *Global simple*, relies on estimates from basic timing experiments. It also re-uses the full set of trials from the *Local* study reported in Chapter 5. The second, or *Global trace*, study obtains its execution time estimates from enforcement tracing capabilities added to the toolkit. *Global trace* also re-uses the programs from the earlier studies — with different input sets — and adds two new programs.

Breakdowns on the characteristics of enforced contract clauses and detected violations are given for each study. Enforcement results are summarized for the baseline and sampling policies in the same manner used in the *Local* study. A comparison of the results across policies provides additional insights into their effects.

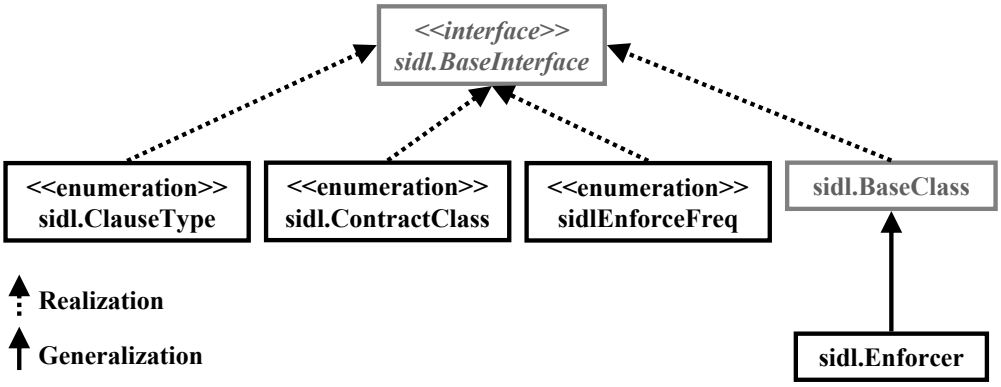

Figure 6.1: SIDL runtime library extensions for global enforcement.

Table 6.1: SIDL enforcement enumerations.

sidl.ClauseType	sidl.ContractClass	sidl.EnforceFreq
Invariant	AllClasses	AdaptiveFit
Postcondition	Constant	AdaptiveTiming
Precondition	Invariants	Always
	Linear	Never
	MethodCalls	Periodic
	Postconditions	Random
	Preconditions	SimulatedAnnealing
	SimpleExpressions	

6.1 Babel Extensions for Global Enforcement

Enforcement infrastructure changes for global enforcement required modifications to the Babel compiler and SIDL runtime library described in Chapter 3. Changes specifically relate to enforcement policies and the enforcement decision process. The management and tracking of enforcement decisions shifts from the generated routines to a new class in the runtime library.

A wider range of interface contract enforcement policies and a centralized enforcement manager are supported through extensions to the specification of the SIDL runtime library. Those extensions, shown in Figure 6.1, consist of three new enumerations and one class. The enumerations identify policy options and support enforcement decisions made by an instance of the new enforcement manager class. Table 6.1 lists the relevant values, in alphabetical order, for each enumeration. The `sidl.ContractClass` and `sidl.EnforceFreq` enumerations define the enforcement policy. The `sidl.ClauseType` enumeration is used

by the generated enforcement routines to identify the type of clause whose assertions are being considered for enforcement. The `sidl.Enforcer` class is responsible for making and tracking contract clause enforcement decisions.

6.1.1 Enforcement Policies

The range of policies is extended to support both traditional and experimental interface contract enforcement strategies. Traditional interface contract enforcement strategies are described in Section 2.4, while experimental strategies consist of the basic sampling techniques together with the new, performance-driven approaches introduced in this research. To provide the most flexibility, the toolkit is modified to form enforcement policies from a combination of options indicating the nature of the clauses to be checked and the frequency at which they should be checked. The options are defined as the following new enumerations in the SIDL specification: `sidl.ContractClass` and `sidl.EnforceFreq`. Of the forty-nine meaningful combinations of values, results for only thirteen policies are reported in the studies.

The `sidl.ContractClass` enumeration supports eight values representing different classifications of contract clauses. Two classifications are supported for the complexity of the assertions within a clause: `Constant` and `Linear`. The presence and absence of method calls in assertions are reflected in the `MethodCalls` and `SimpleExpressions` options, respectively. Clauses can also be distinguished by type through: `Invariants`, `Postconditions`, and `Preconditions`. Finally, all contract clauses are considered for enforcement with the `AllClasses` option.

The desired enforcement frequency is specified through `sidl.EnforceFreq`. Baseline options are `Always` (for checked contract clauses and detected violations) and `Never` (for execution times). Traditional sampling strategies are retained through the `Periodic` and `Random` frequencies. Three performance-driven enforcement options are now supported: `AdaptiveTiming`, `AdaptiveFit`, and `SimulatedAnnealing`. The `AdaptiveTiming` option checks whether the execution time estimate is within the user-specified overhead limit applied to the method time estimate. `AdaptiveFit` checks whether the execution time estimate of a clause, added to the accumulated time of all previously checked clauses,

Table 6.2: Global enforcement policies, where the basis for decisions can be performance constraints, baseline metrics collection, basic sampling, or characteristics of the assertions within the clause.

Enforcement Policy	**Decision Basis**	**Enforcement Enumeration Options**	
		sidl.ContractClass	**sidl.EnforceFreq**
Always	Baseline	`AllClasses`	`Always`
Never	Baseline	`AllClasses`	`Never`
Constant	Characteristics	`Constant`	`Always`
Linear	Characteristics	`Linear`	`Always`
Method calls	Characteristics	`MethodCalls`	`Always`
Postconditions	Characteristics	`Postconditions`	`Always`
Preconditions	Characteristics	`Preconditions`	`Always`
Simple expressions	Characteristics	`SimpleExpressions`	`Always`
Periodic	Sampling	`AllClasses`	`Periodic`
Random	Sampling	`AllClasses`	`Random`
Adaptive fit	Performance	`AllClasses`	`AdaptiveFit`
Adaptive timing	Performance	`AllClasses`	`AdaptiveTiming`
Simulated annealing	Performance	`AllClasses`	`SimulatedAnnealing`

remains within the overhead limit of the accumulated execution time of invoked methods. `SimulatedAnnealing` is essentially `AdaptiveFit` with an allowance for exceeding the overhead limit, but with decreasing probability over time.

The combinations of the `sidl.ContractClass` and `sidl.EnforceFreq` options are used to establish enforcement policies. There are fifty-six possible enforcement option combinations. However, for the purposes of this research, experiments were performed and results analyzed using only the thirteen combinations listed in Table 6.2. `AllClasses` is combined with each enforcement frequency to provide metrics for baseline and basic sampling strategies. Six of the contract clause classification options are combined with the `Always` enforcement frequency to allow characterization of contract clause checks with respect to the trial's execution profile. The `Invariants` classification is not included since none of the specifications contain invariant clauses.

6.1.2 Enforcement Decisions

Global enforcement decisions are made by the `sidl.Enforcer` class based on information about the clause under consideration and the enforcement policy options in

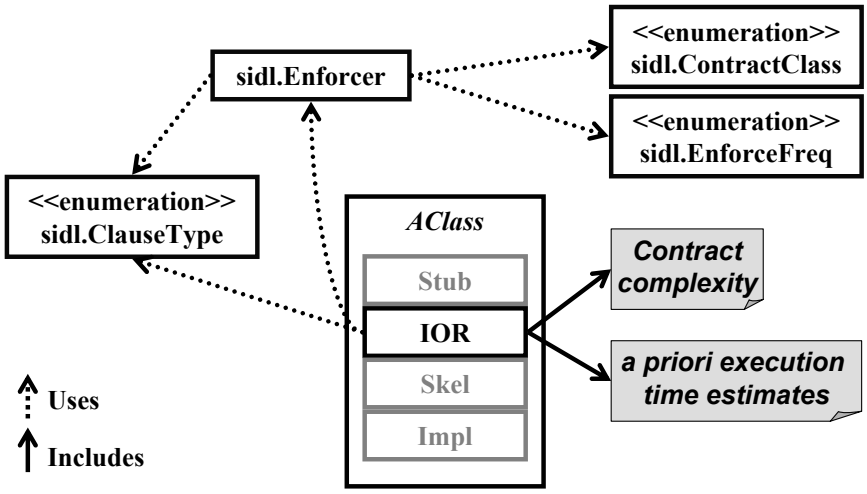

Figure 6.2: Global enforcement decision dependencies, where dotted lines indicate the IOR uses or calls features of the entity and solid lines indicate input values required for enforcement decisions.

affect. Figure 6.2 shows the connections between the generated middleware, represented by *AClass*, and the new extensions to the SIDL runtime library. The Babel compiler infers the default complexity of contract clauses from any built-in function calls contained in the assertions. The `check_method` routine within the IOR — shown in Figure 3.3 — passes data about the contract clause, such as its type, complexity, and execution time estimates, to the `sidl.Enforcer`. The `sidl.Enforcer` then determines whether the clause should be checked based on the enforcement policy in affect. The middleware also maintains statistics on enforcement decisions and violations.

6.1.3 Review

Global interface contract clause enforcement expands on the Babel toolkit extensions described in Chapter 3 by supporting a wide range of enforcement policies and a centralized enforcement manager. Enforcement policies are formed from a combination of contract clause classification and enforcement frequency options. Generated enforcement routines pass data about the contract clause to the enforcement manager for approval and tracking.

Table 6.3: Trials for the *Global simple* study are formed from programs with different combinations of input files and, when appropriate, array sizes.

Component	Program Trials	
	Program	**Description**
Simplicial Mesh	**A**	Retrieve all face entities from the mesh in sets based on the size of the input array. Input array sizes 1, 16, 32, 64, 128, 256, 512, 1024, and 2048 are combined with the five input files to form forty-five trials.
	AA	Retrieve face entities in the same manner as program **A** plus, for each set of faces, retrieve their corresponding adjacent vertexes. The same input set combinations are repeated to form another forty-five trials.
	MA	Retrieve all face entities from the mesh, then for each face, retrieve the adjacent vertexes. Five input files each form a separate trial.

6.2 Simple Execution Time Estimates Study

This study, referred to as *Global simple*, investigates the effects of performance-driven interface contract clause sampling using global enforcement decisions based on execution time estimates from simple timing experiments. Ninety-five trials are formed from the same programs and input sets used for the *Local* study described in Chapter 5. Experiments conducted using the thirteen enforcement policies listed in Table 6.2. Characteristics of the contract clauses and violations are described in this section. Results summarized by input file are then presented and compared for the baseline and sampling policies.

6.2.1 Trials

Once again the simplicial mesh component is employed within the context of three programs illustrating multiple interface usage patterns and reflect different execution profiles. The same combinations of input sets are also employed. Hence, this study re-uses the ninety-five trials defined in the *Local* study. Table 6.3 summarizes the programs and corresponding input sets forming the trials. More information about the programs and trials can be found in Section 5.3.

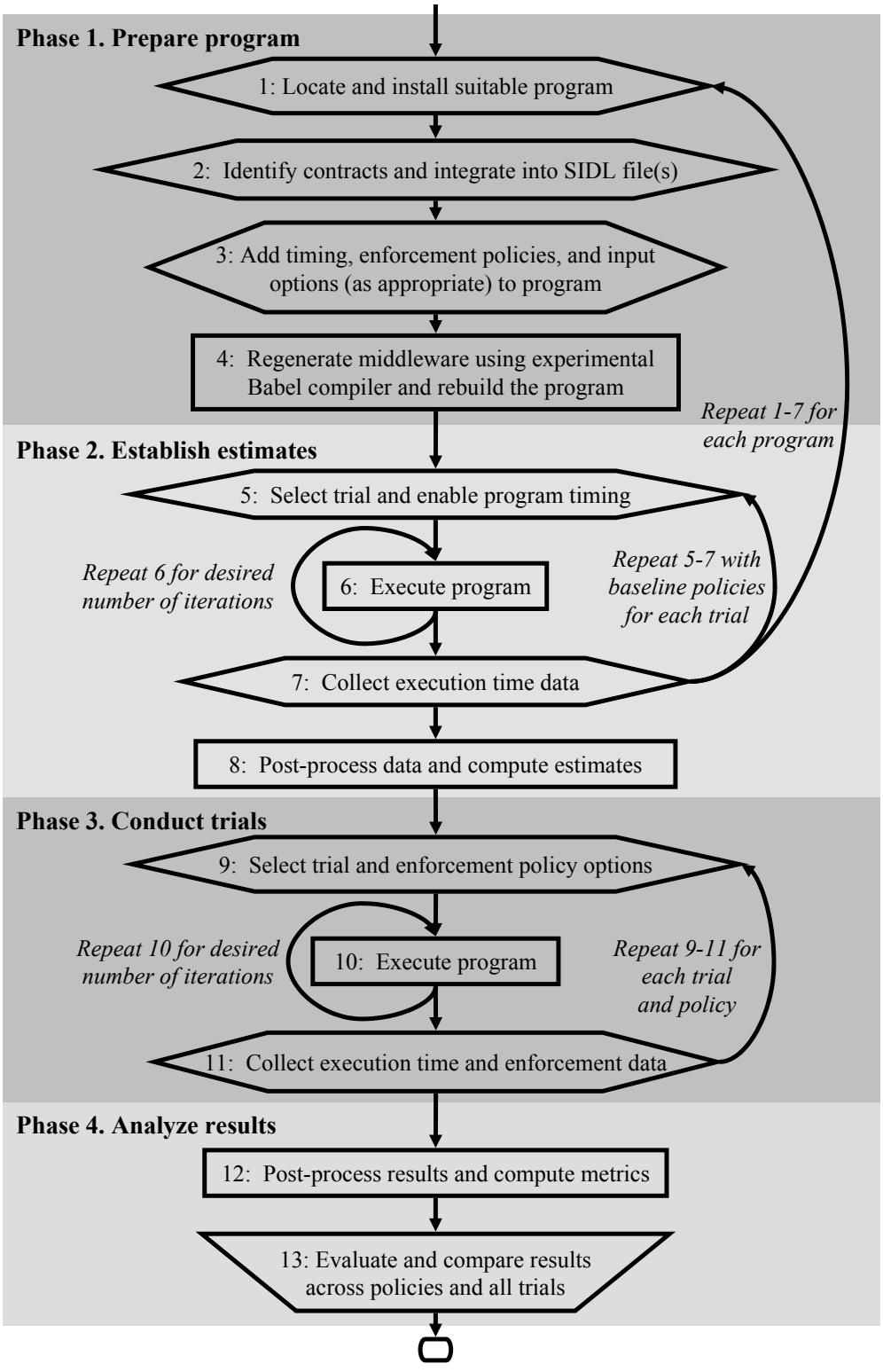

Figure 6.3: *Global simple* work flow for all programs.

6.2.2 Methodology

The process for conducting experiments in this study involves a preliminary set of program executions used to obtain execution time estimates for the performance-driven policies. Section 4.1 describes the common work flow for the trials associated with a single program. The execution time estimates are obtained during a new phase between program preparation and enforcement experiments as illustrated in Figure 6.3. After execution time estimates are calculated from the data, the results are manually integrated into the generated middleware along with the contract clause complexity values associated with the user-defined methods. Phase three in the revised work flow is then repeated for each of the thirteen policies. *Periodic* and *Random* use a 5% sampling rate. Since the number of enforcement opportunities per method potentially doubles (over the *Local* study) with checking on a clause basis, all three performance-driven policies use a 5% enforcement overhead limit. Although the ninety-five trials used in the *Local* study are repeated here, performance data are aggregated in and reported by the scripts on a program and input file basis, yielding fifteen trial sets. Ten iterations of each trial are performed. Finally, the results are compared and analyzed in phase four.

6.2.3 Execution Time Estimates

Estimates are obtained from timing experiments with the *Never* and *Always* enforcement policies using a subset of the trials described in Section 6.2.1. Since the idea was to emulate estimates from component testing, the trials used to obtain estimates are those formed from the three programs, nine input array sizes, and smallest input file. Additional instrumentation added to each program takes time stamps before and after each call to a component method. Execution time estimates are calculated for each method using differences in measurements for the two policies across programs. The precondition and postcondition clauses are each assigned a fraction of the difference based on their proportions of the contract's method calls. For example, as shown in Figure 5.4, `entitysetGetEntities` has one method call (i.e., `validTypeNTopo()`) in its precondition clause and two (i.e., `dimen()` and `size()`) in its postcondition clause for a total of three method calls. Since all three methods are simple, constant-time functions, one third of

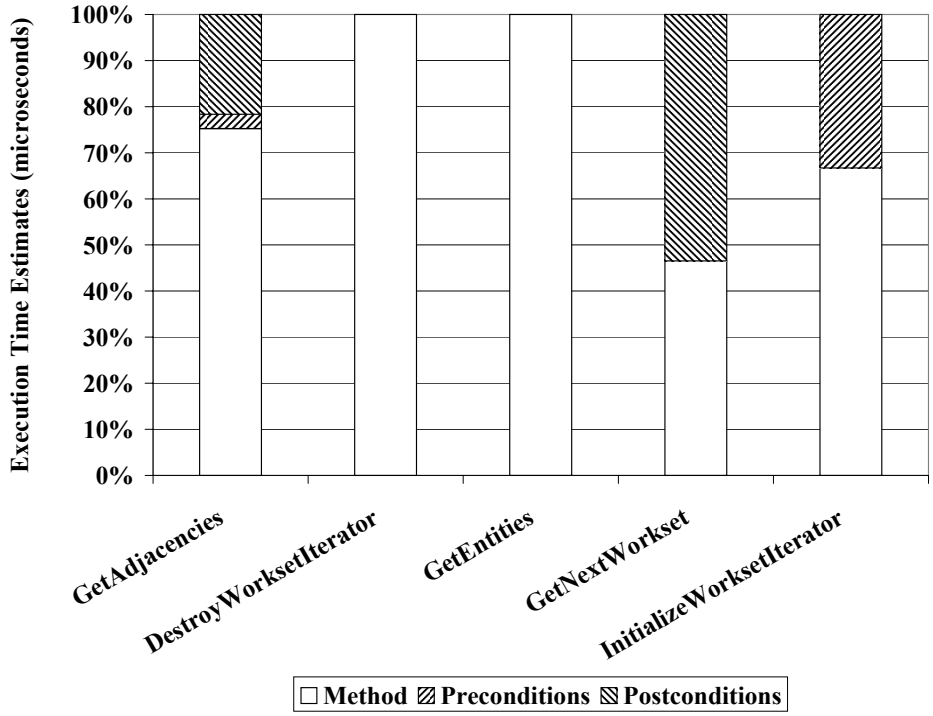

Figure 6.4: *Global simple* execution time estimates. Relative execution costs of methods and contract clauses are shown for those interfaces used by the programs.

the difference becomes the execution time estimate for the method's precondition clause and two-thirds become the estimate for the postcondition clause. (Since input array sizes could consist of a single element, methods of linear-time complexity are weighted the same as those of constant-time complexity.) Figure 6.4 shows the relative contribution of the resulting execution time estimates for the methods invoked in the trials.

6.2.4 Contract Clause Characteristics

Results from the experiments involving contract clause characteristics-based enforcement policies — described in Section 6.1.1 — reveal the contract clause enforcement and violation characteristics for each trial. Since little variation exists in trial set results, Table 6.4 summarizes the mean contract clause checks on a program basis. Results for the *Constant* enforcement policy on program **A** correspond to the expectation formed by an inspection of the specification; namely, all contract clause checks involve only constant-time assertions. The table also indicates half of the clauses checked in program **A** include at least one method call. Although the mean contract clause checks for program **AA** vary for

Table 6.4: Characteristics of mean contract clause checks for the *Global simple* study. Results obtained from experiments using the corresponding enforcement policy. A single asterisk (*) is used to indicate the values apply to such trials regardless of input file, while two asterisks (**) represent all such trials regardless of input array size.

	Mean Contract Clause Checks by Policy (% *Always*)					
Trial(s)	***Constant***	***Linear***	***Method Calls***	***Simple Expr.***	***Precond.***	***Postcond.***
A-*-**	100%	0%	50%	50%	50%	50%
AA-*-1	75%	25%	25%	75%	50%	50%
AA-*-16	75%	25%	25%	75%	50%	50%
AA-*-32	75%	25%	25%	75%	50%	50%
AA-*-64	75%	25%	25%	75%	50%	50%
AA-*-128	75%	25%	25%	75%	50%	50%
AA-*-256	75%	25%	25%	75%	50%	50%
AA-*-512	75%	25%	25%	75%	50%	50%
AA-f1-1024	76%	24%	26%	74%	50%	50%
AA-f2-1024	76%	24%	26%	74%	50%	50%
AA-f3-1024	75%	25%	25%	75%	50%	50%
AA-f4-1024	75%	25%	25%	75%	50%	50%
AA-f5-1024	75%	25%	25%	75%	50%	50%
AA-f1-2048	77%	23%	27%	73%	50%	50%
AA-f2-2048	76%	24%	26%	74%	50%	50%
AA-f3-2048	75%	25%	25%	75%	50%	50%
AA-f4-2048	75%	25%	25%	75%	50%	50%
AA-f5-2048	75%	25%	25%	75%	50%	50%
MA-*	50%	50%	100%	0%	50%	50%

Table 6.5: Classification of mean detected violations for the *Global simple* study, where the same number of violations are detected for each program regardless of input file. Results obtained from experiments using the corresponding enforcement policy.

	Mean Detected Violations by Policy (% *Always*)					
Program	***Constant***	***Linear***	***Method Calls***	***Simple Expr.***	***Precond.***	***Postcond.***
A	100%	0%	100%	0%	0%	100%
AA	100%	0%	100%	0%	0%	100%
MA	0%	0%	0%	0%	0%	0%

four of the six classification-based enforcement policies, the differences are relatively slight, with roughly a quarter of the checked clauses including linear-time assertions and slightly more consisting solely of simple expressions. Half of the clauses enforced in program **MA** contain linear-time assertions but all checks have at least one method call.

Enforcing contract clauses on a classification basis also facilitates describing characteristics of the violated clauses, as shown in Table 6.5. Program **MA** has no detected violations regardless of enforcement policy. Since programs **A** and **AA** retrieve the mesh using the same method their contract clause violation characteristics are identical.

So the trial sets for the three programs are dominated by checks of constant-time clauses and clauses with at least one method call. All violations occur in constant-time postcondition clauses containing at least one method call. While the characteristics of checked contract clauses and detected violations are obvious from experience with the *Local* study and inspection of the specification, the results provide some confidence in the middleware's ability to track enforcement and violation data.

6.2.5 Results

Data for the ninety-five trials are automatically aggregated by the scripts on the basis of program and input file, forming the fifteen trial sets listed in Table 6.6. Consequently, as in the *Local* study, the view into the data is based on consistent numbers of entities across all trials within a set. Baseline data and overhead metrics are provided for the *Always* policy. Results for basic sampling policies are followed by those from the performance-driven enforcement policies. Figures present trial set metrics, relative to baseline policies, in order of mean enforcement overhead using the *Always* policy.

Table 6.6: Trial sets by input file for the *Global simple* study, where *NA* indicates the input option is not applicable. Input files are numbered by size from smallest (**f1**) to largest (**f5**).

Trial Set	**Program**	**Input Set**	
		Input Array Sizes	**Input File**
A-f1	**A**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f1**
A-f2	**A**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f2**
A-f3	**A**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f3**
A-f4	**A**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f4**
A-f5	**A**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f5**
AA-f1	**AA**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f1**
AA-f2	**AA**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f2**
AA-f3	**AA**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f3**
AA-f4	**AA**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f4**
AA-f5	**AA**	1, 16, 32, 64, 128, 256, 512, 1024, and 2048	**f5**
MA-f1	**MA**	*NA*	**f1**
MA-f2	**MA**	*NA*	**f2**
MA-f3	**MA**	*NA*	**f3**
MA-f4	**MA**	*NA*	**f4**
MA-f5	**MA**	*NA*	**f5**

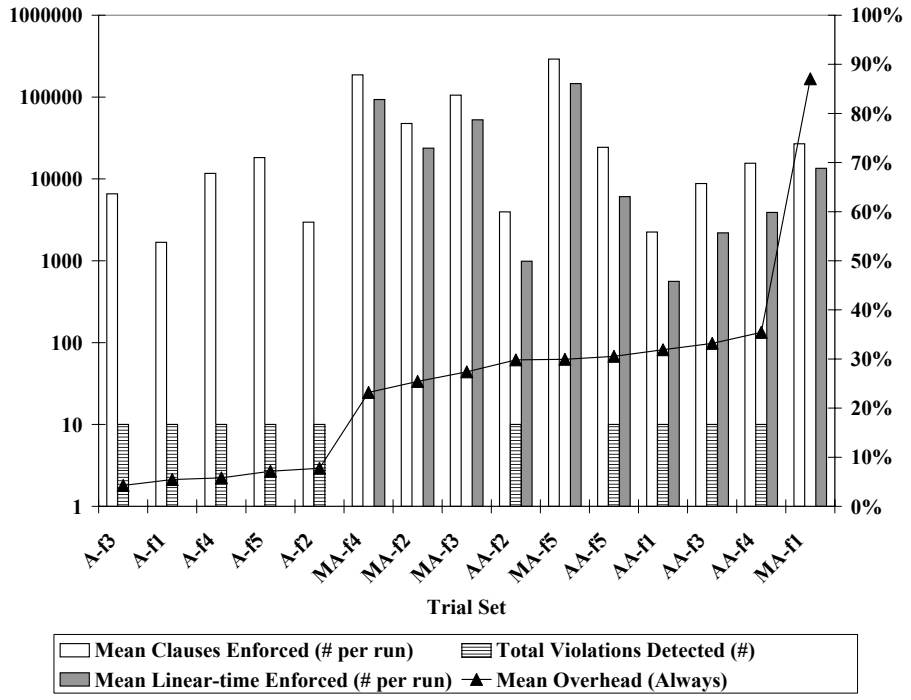

Figure 6.5: *Global simple* study results for the *Always* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program (**A**, **MA**, and **AA**) with the input file (**f1**...**f5**). For example, **A-f5** is the trial set formed from results for program **A** and input file **f5**.

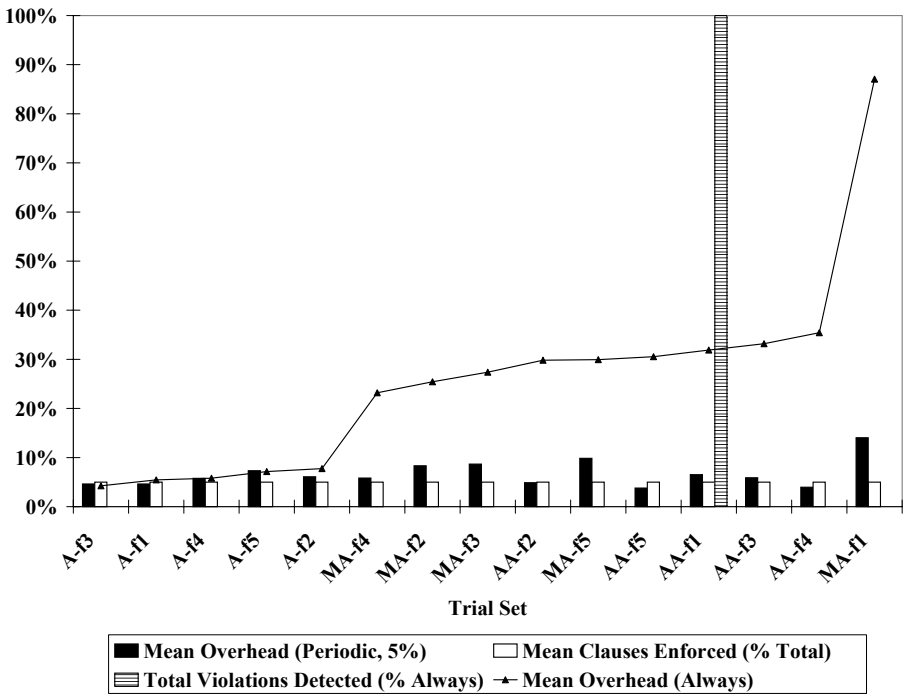

Figure 6.6: *Global simple* study results for the *Periodic* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and input file as described in the caption of Figure 6.5.

Figure 6.5 illustrates metrics data and mean enforcement overhead for the *Always* policy. The range of mean contract clauses checked across trial sets is 1,684 to 291,740. Mean checks of contract clauses including at least one linear-time assertion ranges from 561 to 145,869 for the ten trial sets with such clauses. A total of ten violations are detected for each trial set associated with programs **A** and **AA** as a result of the ten iterations. Trial sets for program **A**, whose methods have contract clauses consisting solely of constant-time assertions, incurs between 4% and 8% mean overhead. Program **MA**'s trial sets generally incur between 23% and 30% mean overhead, while program **AA**'s overhead ranges from 30% to 35%. So, as happened in the *Local* study, the overhead of checking contract clauses seems to be tied to the absence or presence of linear-time assertion checks.

Results for the *Periodic* policy are presented in Figure 6.6. The enforcement overhead ranges from 4% to 7% across trial sets for programs **A** and **AA** but between 6% and 14% for program **MA**. The mean number of contract clauses enforced matches the 5% sampling rate across trial sets. Violations are only detected for trial set **AA-f1**, however, which reflects the shift to global enforcement decisions and the violations occurring in the

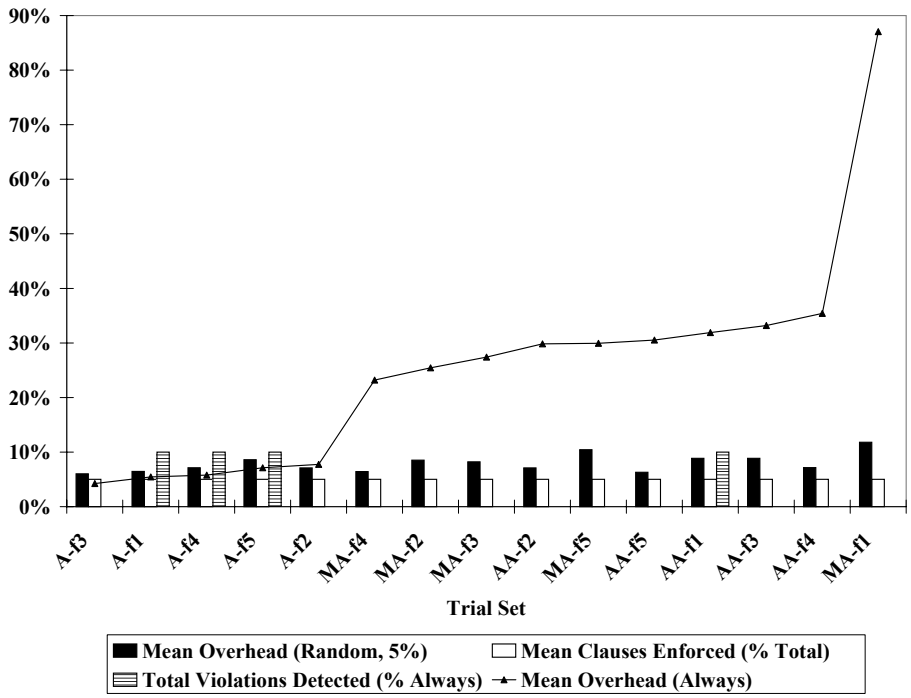

Figure 6.7: *Global simple* study results for the *Random* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and input file as described in the caption of Figure 6.5.

contracts associated with the calls made on the sampling interval for only one of the trial sets.

Figure 6.7 shows results for the *Random* policy relative to baselines. The mean enforcement overhead generally ranges from 6% to 10% across trial sets, but reaches a high of 12% for trial set **MA-f1**. The mean number of contract clauses enforced matches the 5% sampling rate across trial sets. Finally, four trial sets detect 10% of their violations, which translates into one violation per trial set. So the *Random* policy is able to detect violations across more trial sets than the *Periodic* policy.

Results for the *Adaptive timing* policy are presented in Figure 6.8. Mean overhead ranges from 6% to 9% for programs **A** and **AA**, but 6% to 11% for program **MA**. Mean contract clauses enforced is 50% across all trial sets. Unfortunately, the increased checking does not translate into the detection of contract clause violations. The reason is clear when the execution time estimate of the offending clause is considered. That is, the postconditions clause of the method with the violation is never enforced with the *Adaptive timing* policy since it exceeds 5% of the method's estimate.

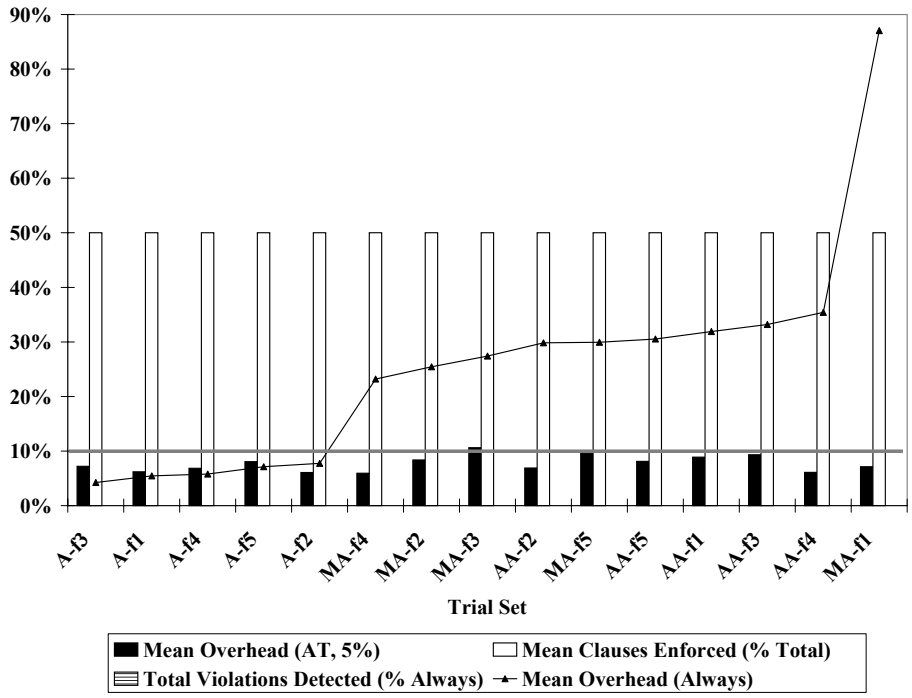

Figure 6.8: *Global simple* study results for the *Adaptive timing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and input file as described in the caption of Figure 6.5.

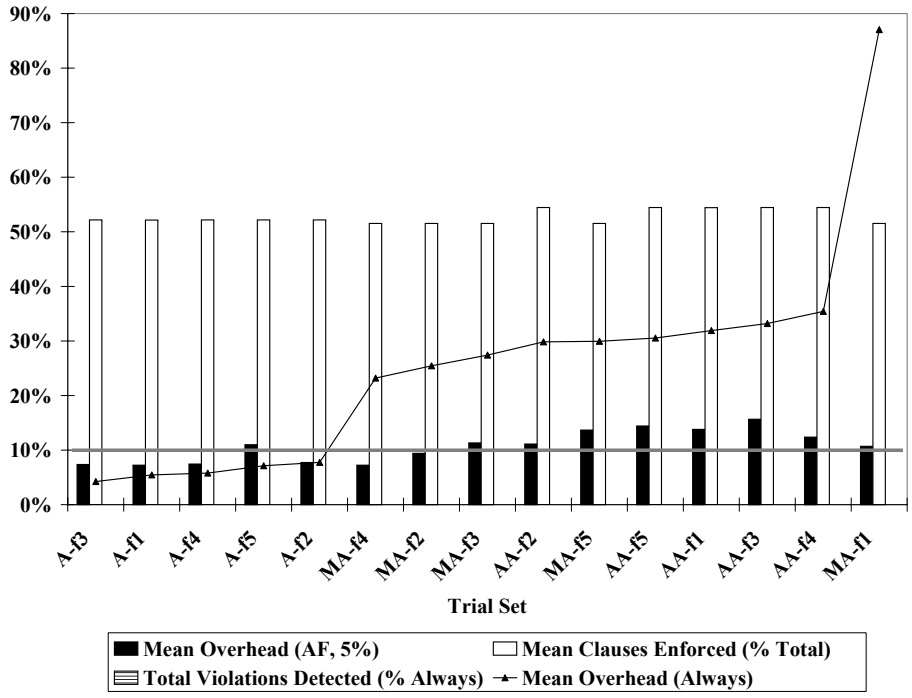

Figure 6.9: *Global simple* study results for the *Adaptive fit* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and input file as described in the caption of Figure 6.5.

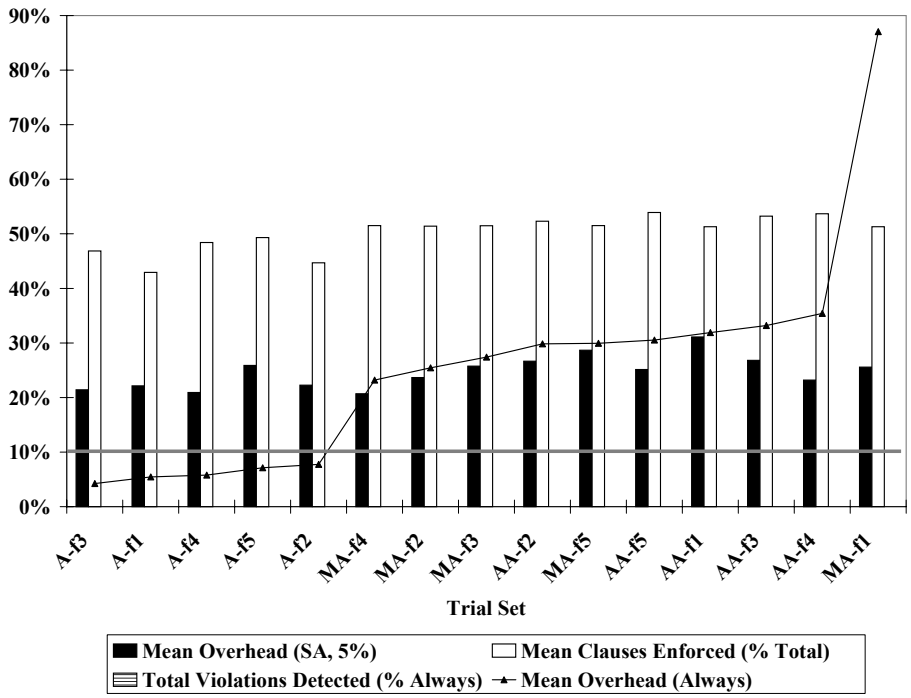

Figure 6.10: *Global simple* study results for the *Simulated annealing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and input file as described in the caption of Figure 6.5.

Figure 6.9 shows *Adaptive fit* policy results. The mean enforcement overhead ranges from 7% to 11% for programs **A** and **MA** — except trial set **MA-f5** — while checking 52% of the contract clauses. Trial sets for program **AA** check 54% of the contract clauses, while incurring between 11% and 16% mean overhead. Once again, however, no violations are detected as a result of the *a priori* execution time estimates precluding enforcement of the offending clause.

Results for the *Simulated annealing* policy are presented in Figure 6.10. Median overhead ranges from 21% to 26% while checking 43% to 49% of the contract clauses for program **A**. The addition of the method with linear-time assertions increases the overhead and enforcement ranges for the other two programs. In particular, program **MA** incurs 21% to 29% overhead while checking 51% to 52% of its clauses. Program **AA** checks 51% to 54% of its clauses while incurring between 23% and 31% mean overhead. Again, no contract clause violations are detected for any trial sets. In this case, the excess overhead across trial sets can be attributed to an error in `sidl.Enforcer`. Specifically, the allowance for exceeding the overhead limit decreases over time but is not applied on a random basis.

Aggregating data for the ninety-five trials into fifteen, input file-based sets establishes trial sets with consistent numbers of entities yet different execution profiles resulting from varying the input array sizes. Execution time estimates used by the three performance-driven enforcement policies are calculated based on simple timing experiments and shared by all three programs. The *Always* policy incurs overhead within the 10% rule of thumb for program **A** but 2.3 to 8.7 times the amount for the two programs with linear-time assertion checks. *Periodic* and *Random* sampling generally incur less than 10% overhead across trial sets while checking only 5% of the clauses; however, they are able to detect violations in one to four of the trial sets, respectively. The *Adaptive timing* policy generally stays within 10% overhead across trial sets while checking 50% of the clauses. Similarly, the *Adaptive fit* policy generally incurs no more than a few percent over the 10% overhead level, while enforcing between 52% and 54% of the clauses. As a result of a bug in `sidl.Enforcer`, the *Simulated annealing* policy incurs between 21% and 31% overhead across trial sets, though less than the *Always* policy for programs involving linear-time assertion checks. Unfortunately, the *Adaptive timing*, *Adaptive fit*, and *Simulated annealing* policies are not able to detect any violations. So, while two of the three performance-driven enforcement policies generally incur at or near the 10% overhead level while enforcing at least half of the contract clauses, none are able to detect violations as a result of the *a priori* execution time estimates.

6.2.6 Discussion

An analysis of the results for the *Global simple* study reveals a couple of patterns relating primarily to the levels of enforcement for the various sampling strategies. As in the *Local* study, the mean overhead again appears to be tied to the presence of contract clauses with linear-time assertions. The use of method-specific execution time estimates across all programs also has an obvious impact on the three performance-driven enforcement policies.

Once again, the absence and presence of linear-time assertions in enforced contract clauses affects the mean enforcement overhead. Results for the *Always* policy reveal a minimum increase of 15% in mean overhead between trial sets for program **A**, which calls methods whose contract clauses consist solely of constant-time assertions, and the overhead

of the remaining trial sets. Trial sets for program **AA** tends to incur more enforcement overhead than those of program **MA**, but the difference is less obvious in this study than it was in the *Local* study.

Between 43% and 54% of contract clauses are checked across trial sets with the three performance-driven enforcement policies. The *Adaptive timing* policy is able to keep the mean enforcement overhead under 10% for all but one of the trial sets. The policy's enforcement level (of 50%) is accomplished within the 10% rule of thumb by enforcing only those clauses whose execution time estimates are within 5% of their corresponding methods' execution time estimates. The *Adaptive fit* policy increases its minimum enforcement level to 52% but tends to exceed the 10% overhead level in most trial sets involving linear-time assertions. By allowing the 5% limit to be exceeded early on in the execution of trials, the *Simulated annealing* policy incurs two to three times the 10% overhead level while checking between 43% and 54% of the clauses.

By emulating the use of simple, *a priori* execution cost estimates, this study provides insights into potential interface contract clause enforcement issues arising with general metadata extensions to component repositories. That is, maintaining general execution time estimates for contract clauses and methods on a component basis is likely to be too coarse for performance-driven policies to adjust their level of enforcement to individual combinations of programs and input sets.

While performance-driven enforcement policies do not detect interface contract clause violations in this study, they do tend to check roughly half of the contract clauses with savings on the mean overhead on all trial sets involving linear-time assertions. Unfortunately, the technique used to obtain the *a priori* execution time estimates — where a single set of estimates is applied to all programs — appears to be too coarse to allow more variation in the enforcement level between trial sets.

6.2.7 Review

Results from a study of global interface contract clause enforcement with three performance-driven enforcement policies relying on simple, *a priori* execution time estimates are described in this section. The estimates are obtained from basic timing ex-

periments conducted on a subset of the trials investigated in the study. The goal of the study is to determine the nature of the checked contract clauses and the effects of the new enforcement approach. Experiments are performed with three programs using different combinations of methods to retrieve sets of entities from a single implementation of a mesh component. Five input files and nine input array sizes are combined with the programs to form a total of ninety-five trials. Results are presented as an aggregation of the data on a program and input file basis.

Findings indicate the performance-driven enforcement policies — especially *Adaptive timing* and *Adaptive fit* — do automatically adjust the level of contract clause enforcement to the programs. They also have better overall control over the mean enforcement overhead. However, the approach to obtaining execution time estimates precludes detection of violations in the trials. This result most likely stems from the fact that the estimate for the violated postconditions clause exceeds the estimate of the associated method by 15%, as shown in Figure 6.4.

6.3 Trace-based Execution Time Estimates Study

This study, referred to simply as *global trace*, investigates the effects of performance-driven interface contract clause sampling using global enforcement decisions based on execution time estimates from enforcement traces. Thirteen trials are formed from five, single-component programs. Enforcement experiments conducted using the thirteen policies listed in Table 6.2 are used to determine the nature of checked contract clauses and the impacts of several sampling strategies. Characteristics of the contract clauses and violations are described in this section. Results are presented and compared for the baseline and sampling policies.

6.3.1 Babel Extensions for Enforcement Tracing

The Babel toolkit supports interface contract clause enforcement tracing through new capabilities added to the runtime library and generated middleware. The SIDL specification provides methods for starting, logging, stopping, and disabling enforcement traces through the `sidl.Enforcer` class. When tracing is enabled, time stamps are taken at

Table 6.7: Enumeration `sidl.EnfTraceLevel`.

Value	Description
`None`	Disabled or no tracing.
`Core`	Generate start and end (trace) timing only.
`Basic`	`Core` plus clause and method execution timing.

Table 6.8: Subjects for the *Global trace* study consist of five programs.

Component	Program	Description
Simplicial Mesh	**A**	Retrieve all face entities from the mesh in work sets based on the size of the input array.
	AA	Retrieve face entities in the same manner as program **A** plus, for each work set of faces, retrieve their corresponding adjacent vertexes.
	MA	Retrieve all face entities from the mesh, then retrieve the adjacent vertexes for each face.
Volume Mesh	**MT**	Exercise and check the consistency of five mesh interface sets: 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 standard array operations in one of three modes: successful execution; one or more precondition violations; and one or more postcondition violations.

key points during processing based on the desired tracing level, which is set through the addition of the `sidl.EnfTraceLevel` enumeration. The available levels are described in Table 6.7. `Core` is useful for timing the execution of a program. *Basic* logs the amount of time spent in the program (once tracing is enabled), the methods, and each contract clause. These levels take affect when the generated enforcement routines call `sidl.Enforcer`. The logging method, which is passed method-specific execution times obtained through runtime system calls, is invoked right before control is returned to the caller.

6.3.2 Subjects

Table 6.8 lists the five programs leveraged as subjects in this study. The first three programs are re-used from the *Local* and *Global simple* studies, so are described further in Section 5.2. The two new programs — **MT** and **VT** — are test suites available with their components. The volume mesh component used by program **MT** is a componentized version of the Generation and Refinement of Unstructured, Mixed-Element Meshes

Table 6.9: Trials for *Global trace* study are formed from combinations of programs, input files, and, when appropriate, input array sizes. Array sizes for programs **A** and **AA** are chosen to induce the violation discovered in the first study. *NA* indicates the option is not applicable or not varied.

Trial	Program	Input Set	
		Input File	Input Array Size
A-f5-1	**A**	f5	1
A-f5-14587	**A**	f5	14587
A-f5-145870	**A**	f5	145870
AA-f5-1	**AA**	f5	1
AA-f5-14587	**AA**	f5	14587
AA-f5-145870	**AA**	f5	145870
MA-f5	**MA**	f5	*NA*
MT	**MT**	*NA*	*NA*
VT-6	**VT**	*NA*	6
VT-10	**VT**	*NA*	10
VT-100	**VT**	*NA*	100
VT-1000	**VT**	*NA*	1000
VT-10000	**VT**	*NA*	10000

in Parallel (GRUMMP) [76] (version 0.2.2b) volume mesh. Program **MT** is a test suite developed by the TSTT community for checking compliance with the mesh interface specification [140, 173]. The program exercises five interface sets, described in Table 6.8, on volume mesh data loaded from an input file. Program **VT** is a regression test suite distributed with the Babel language interoperability source code [111]. The program exercises standard vector operations available in a simple implementation of a utilities component supporting vectors of `double` values. Program **VT** initializes several vectors with fixed values at startup. Many of the operations perform simple computations involving elements of one or two vectors. Hence, of the five subjects in this study, four utilize mesh data management components and one exercises all features of a utilities component supporting standard vector operations.

6.3.3 Trials

Experiments in this study are conducted using the thirteen trials listed in Table 6.9. The program, input file (if any), and input array size are shown in the table. Section 6.3.2 describes each program. Unlike in the previous studies, only the largest input file is used for the first three programs. The input array size options were chosen to induce

the violation discovered in the *Local* study. Consequently, three sizes are used: 1, 14587, and 145870. The first size, 1, processes individual faces in a tight loop. Size 14587 operates with 10% of the input file at a time. Size 145870 is process all entities from the entire mesh — described by the input file — in the first iteration of each loop. Only one trial is formed from program **MT** due to the volume of calls made using a small input array readily available with the software. Finally, five trials are formed with program **VT**, starting with the original six elements and generally increasing the size by an order of magnitude. Hence, the thirteen trials are formed from five programs, one input file (when appropriate), and either three or five input array sizes (when appropriate).

6.3.4 Methodology

The basic process for conducting experiments is described in Section 4.1 with the addition of preliminary experiments to obtain execution time estimates from enforcement traces. The revised work flow is illustrated in Figure 6.11. Section 6.3.5 elaborates on the process used to obtain timing data. Estimates of contract clause and method execution times along with the actual complexity of contract clauses are stored in a file automatically read by the middleware to establish trial-specific enforcement inputs. Phase three is repeated for each policy under study using either ten or thirty iterations, depending on the program. The *Periodic* and *Random* policies used a 5% sampling rate. The overhead limit for all three performance-driven policies was set at 5% to account for checking precondition and postcondition clauses separately, as described in Section 6.2.2. Finally, the results are analyzed and compared in phase four.

6.3.5 Execution Time Estimates

Enforcement traces are produced to obtain program, method, and contract clause execution times for use in the experiments. The enforcement tracing feature, described in Section 6.3.1, is used to establish those costs. Due to the sizes of generated trace files, each trial is executed five times with tracing using the *Always* policy. Mean execution times are then computed to obtain trial-specific estimates. Figure 6.12 illustrates the resulting enforcement trace results for each trial. Results are presented in increasing order

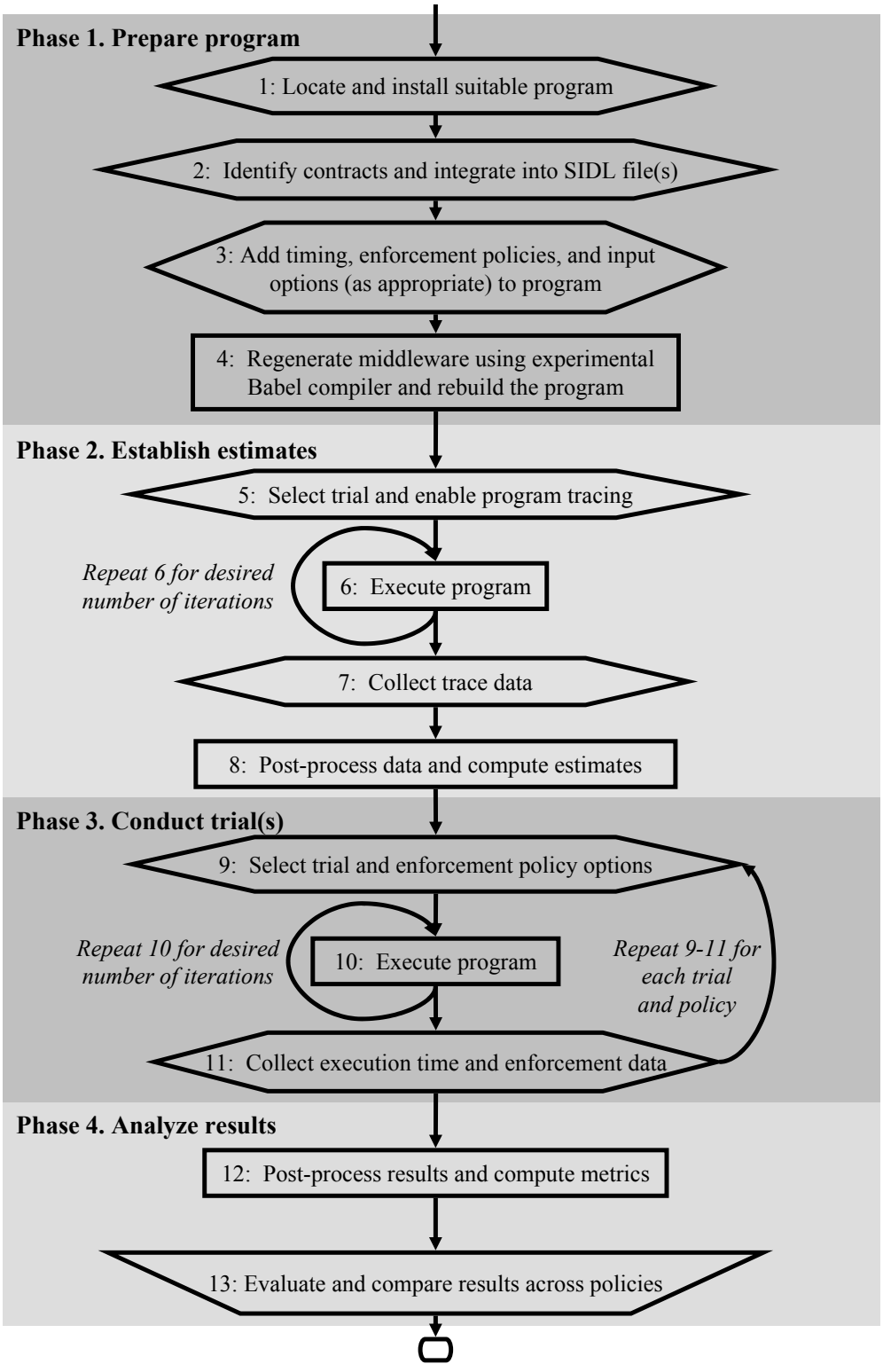

Figure 6.11: *Global trace* work flow for a single program.

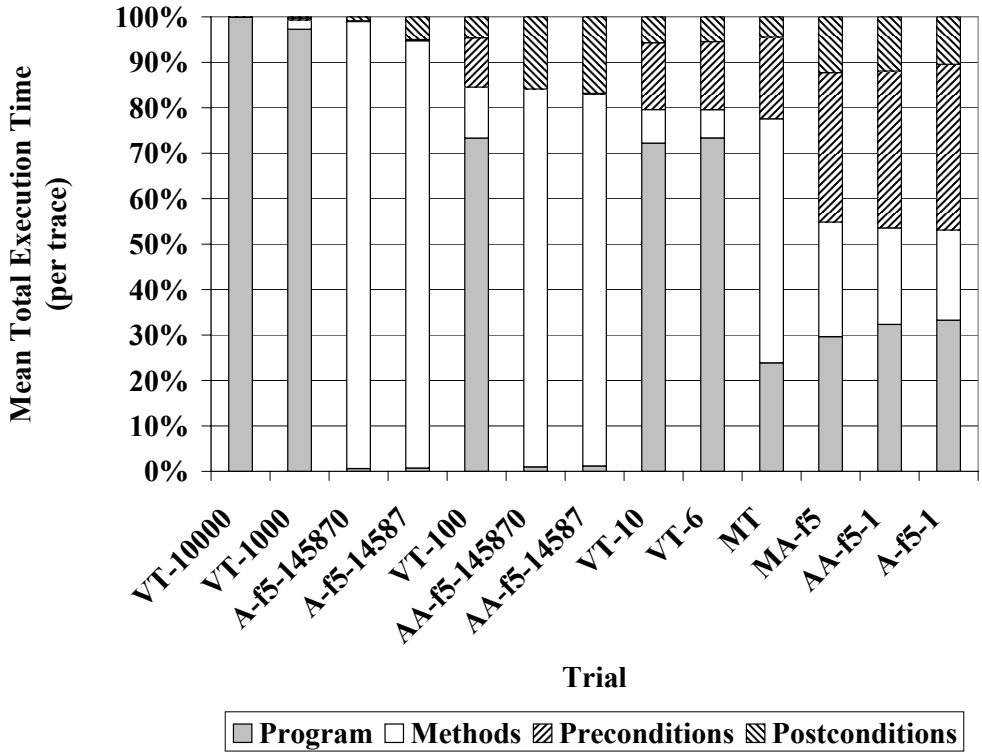

Figure 6.12: Trace execution profiles for *Global trace* study, where execution times are measures of time attributed to: program statements, component methods, precondition enforcement, and postcondition enforcement. Trial set names are formed by the concatenation of the program (**A**, **MA**, **AA**, **MT**, and **VT**) with, as appropriate, the input file (**f5**) and/or input array size. For example, **A-f5-145870** is the trial set formed from results for program **A**, input file **f5**, and input array size **145870**.

Table 6.10: Characteristics of mean clause checks for the *Global trace* study. Results obtained from experiments using the corresponding enforcement policy. A single asterisk (*) is used to indicate the values apply to such trials regardless of input file.

Trial(s)	Mean Contract Clause Checks by Policy (% *Always*)					
	Constant	***Linear***	***Method Calls***	***Simple Expr.***	***Precond.***	***Postcond.***
A-f5-*	100%	0%	50%	50%	50%	50%
AA-f5-1	75%	25%	75%	25%	50%	50%
AA-f5-14587	77%	24%	73%	26%	50%	50%
AA-f5-145870	88%	22%	70%	30%	50%	50%
MA-f5	50%	50%	100%	0%	50%	50%
MT	99.995%	0.005%	27%	73%	58%	42%
VT-*	95%	8%	100%	0%	80%	33%

of estimated enforcement overhead. Preconditions clauses dominate contract costs for trials **MA-f5**, **A-f5-1**, and **AA-f5-1**. The total costs of methods far exceed the times attributed to the other categories for trials **A-f5-14587**, **A-f5-145870**, **AA-f5-14587**, and **AA-f5-145870**. However, contract execution times are dominated by postconditions clauses for the latter two as a result of their linear-time contract clauses. Only 15-20% of the execution times of trials **VT-6**, **VT-10**, and **VT-100** are attributed to contract clauses, where even less time is spent in the associated methods. Finally, execution times for trials **VT-1000** and **VT-10000** are almost exclusively spent in the programs. Nearly every trial illustrates a different combination of execution time percentages.

6.3.6 Contract Clause Characteristics

Results from experiments involving classification-based enforcement policies reveal characteristics of enforced and violated clauses for each trial. The data indicate the nature of the exercised and violated contract clauses. This data provides clues regarding the type of work performed during enforcement.

Clause characteristics for the thirteen trials in this study are summarized in Table 6.10. As in the previous study, the number of method calls affects the contract clause characteristics of program **AA**. There is as much as a 13% difference in mean clause checks between the smallest and largest input array sizes in this case. While contract clause checks

Table 6.11: Classification of mean detected violations for the *global trace* study. Results obtained from experiments using the corresponding enforcement policy. The same violations occur for each program regardless of input array size.

	Mean Detected Violations by Policy (% *Always*)					
Program	***Constant***	***Linear***	***Method Calls***	***Simple Expr.***	***Precond.***	***Postcond.***
A	100%	0%	100%	0%	0%	100%
AA	100%	0%	100%	0%	0%	100%
MA	0%	0%	0%	0%	0%	0%
MT	9%	91%	100%	0%	9%	91%
VT	94%	8%	100%	0%	94%	40%

for program MT are nearly 100% constant-time, a mere 0.005% are classified as linear-time. Program **MT** checks constant-time contract clauses whose assertions include at least one method call. Therefore, the majority of clause checks involve constant-time clauses and those containing method calls.

Since only contract clauses meeting the enforcement policy criteria are checked, the programs may continue to make calls to the component. The corresponding additional contract clause checks are not necessarily enforced with the other policies. This can lead to additional violations not otherwise detected. Consequently, the metrics for complementary enforcement policies — such as those obtained using the *Method calls* and *Simple expressions* policies — may not add up to 100%.

Results from the experiments involving contract clause classification-based enforcement policies reveal the contract clause enforcement and violation characteristics of each trial's execution profile. Since only contract clauses meeting the enforcement policy criteria are checked, the programs may continue to make calls to the component, thereby checking additional contract clauses not necessarily enforced with other policies — including *Always*. Consequently, the numbers for complementary enforcement policies may not add up to 100%.

Violation characteristics for this study are presented in Table 6.11. A single contract clause violation occurs per execution for all trials using programs **A** and **AA**. Since both programs retrieve the mesh using the same method their violated clause characteristics are identical. By employing modified enforcement wrappers to continue processing after violations are detected, a total of forty-seven contract clause violations are found

Table 6.12: Description of clause violations for the *Global trace* study, where the same violations occur regardless of input array size.

Program	Violation Descriptions
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 occurs as in program **A** trials.
MA	No contract clause violations.
MT	Four precondition violations occur in constant-time contract clauses 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 the *Always* policy, where postcondition failures are emulated. In all, 94% of the violations are triggered in constant-time preconditions.

per execution of the trial *MT*. Interestingly, while 99.995% of *MT*'s contract clauses are constant-time, 91% of the violations are detected in clauses containing at least one linear-time assertion. Overall, trials in this study are dominated by violations in postconditions clauses and constant-time contract clauses with method calls.

Table 6.12 describes the interface contract clause violations detected in this study. The three mesh programs violating the community-developed interface specification do so as a result of both the programs and components being developed prior to the definition of the interface contract clauses. Recall the violated assertion for programs **A** and **AA** is indicated in boldface type in Figure 5.4. Similarly, Figures 6.13 and 6.14 highlight the interface specifications for the four methods incurring contract clause violations during execution of program **MT**'s trial. Program **VT**, on the other hand, exhibits characteristics of non-compliant programs and implementations as a result of deliberately triggering violations of assertions within precondition and postcondition clauses. Figures 6.15 and 6.16 show the interface contract clauses associated with all of the methods in the specification used by program **VT**. Since the goal of the program is to violate every assertion at least once, every executable assertion appears in boldface type.

Hence, the thirteen trials are dominated by checks of constant-time clauses and clauses with at least one method call. So it is not surprising to find the majority of violations being detected in constant-time clauses containing at least one method call for three of the four programs. The **MT** trial, however, is an exception. Even though nearly all

```
void getAllVtxCoords (in opaque entity_set, inout array<double> coords, out
      int coords_size, inout array<int> in_entity_set, out int in_entity_set_size,
      inout StorageOrder storage_order) throws TSTTB.Error;
   require
      passed_set_handle: entity_set != null;
      passed_allocd_coords: coords != null;
      coords_is_1d: dimen(coords) == 1;
      passed_allocd_in_set: in_entity_set != null;
      in_set_is_1d: dimen(in_entity_set) == 1;
   ensure
      returned_coords_array: coords != null;
      coords_still_1d: dimen(coords) == 1;
      coords_size_okay: irange(coords_size, getGeometricDim(), size(coords));
      coords_size == (in_entity_set_size * getGeometricDim());
      returned_in_set_array: in_entity_set != null;
      in_set_still_1d: dimen(in_entity_set) == 1;
      in_set_size_okay: irange(in_entity_set_size, 0, size(in_entity_set));
      claim_no_side_effects: is pure;

void getVtxCoordIndex (in opaque entity_set, in EntityType
      requested_entity_type, in EntityTopology requested_entity_topology,
      in EntityType entity_adjacency_type, inout array<int>
      offset, out int offset_size, inout array<int> index, out int
      index_size, inout array<EntityTopology> entity_topologies,
      out int entity_topologies_size) throws TSTTB.Error;
   require
      passed_set_handle: entity_set != null;
      allocd_offset_array: offset != null;
      offset_array_is_1d: dimen(offset) == 1;
      allocd_index_array: index != null;
      index_array_is_1d: dimen(index) == 1;
      allocd_topo_array: entity_topologies != null;
      topo_array_is_1d: dimen(entity_topologies) == 1;
   ensure
      offset_still_valid: offset != null;
      offset_still_1d: dimen(offset) == 1;
      offset_size_in_range: irange(offset_size, 0, size(offset));
      offset_size_okay: offset_size == entity_topologies_size + 1;
      offset_non_decreasing: nonDecr(offset);
      index_still_allocd: index != null;
      index_still_1d: dimen(index) == 1;
      index_size_in_range: irange(index_size, 0, size(index));
      topo_still_alloc: entity_topologies != null;
      topo_still_1d: dimen(entity_topologies) == 1;
      topo_size_okay: irange(entity_topologies_size, 0, size(entity_topologies));
      claim_no_side_effects: is pure;
```

Figure 6.13: Interface contracts for trial **MT** methods with preconditions violations appearing in boldface type and linear-time assertions in italics.

```
void getAdjEntities (in opaque entity_set, in EntityType entity_type_requestor,
      in EntityTopology entity_topology_requestor, in EntityType
      entity_type_requested, inout array<opaque> adj_entity_handles,
      out int adj_entity_handles_size, inout array<int> offset, out int
      offset_size, inout array<int> in_entity_set, out int
      out int in_entity_set_size) throws TSTTB.Error;
   require
      passed_set_handle: entity_set ! = null;
   ensure
      adj_is_returned: adj_entity_handles ! = null;
      adj_is_1d: dimen(adj_entity_handles) == 1;
      adj_size_okay: irange(adj_entity_handles_size, 0, size(adj_entity_handles));
      offset_is_returned: offset ! = null;
      offset_is_1d: dimen(offset) == 1;
      offset_size_okay: irange(offset_size, 0, size(offset));
      offset_values_okay: irange(offset, 0, offset_size);
      offsets_non_decreasing: nonDecr(offset);
      in_set_is_returned: in_entity_set ! = null;
      in_set_is_1d: dimen(in_entity_set) == 1;
      in_set_size_okay: irange(in_entity_set_size, 0, size(in_entity_set));
      claim_no_side_effects: is pure;

void getEntArrAdj (in array<opaque> entity_handles, in int
      entity_handles_size, in EntityType entity_type_requested,
      inout array<opaque> adj_entity_handles, out int
      adj_entity_handles_size, inout array<int> offset, out int
      offset_size) throws TSTTB.Error;
   require
      passed_entity_handles: entity_handles ! = null;
      handles_is_1d: dimen(entity_handles) == 1;
      handles_size_okay: irange(entity_handles_size, 0,
         size(entity_handles));
   ensure
      adj_handles_returned: adj_entity_handles ! = null;
      adj_handles_is_1d: dimen(adj_entity_handles) == 1;
      adj_handles_size_okay: irange(adj_entity_handles_size, 0,
         size(adj_entity_handles));
      offset_returned: offset ! = null;
      offset_is_1d: dimen(offset) == 1;
      offset_size_okay: offset_size == entity_handles_size + 1;
      offset_values_okay: irange(offset, 0, offset_size);
      offset_non_decreasing: nonDecr(offset);
      claim_no_side_effects: is pure;
```

Figure 6.14: Interface contracts for trial **MT** methods with postcondition violations appearing in boldface type and linear-time assertions in italics.

```
static bool isZero (in array<double> u, in double tol)
    throws      sidl.PreViolation;
    require     u != null; dimen(u) == 1; tol >= 0.0;
    ensure      no_side_effects : is pure;

static bool isUnit (in array<double> u, in double tol)
    throws      sidl.PreViolation, NegativeValueException, sidl.PostViolation;
    require     u != null; dimen(u) == 1; tol >= 0.0;
    ensure      no_side_effects : is pure;
                dimen(u) == 1; tol >= 0.0;

static bool areEqual (in array<double> u, in array<double> v, in double tol)
    throws      sidl.PreViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v); tol >= 0.0;
    ensure      no_side_effects : is pure;

static bool areOrthogonal (in array<double> u, in array<double> v,
        in double tol)
    throws      sidl.PreViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v); tol >= 0.0;
    ensure      no_side_effects : is pure;

static bool schwarzHolds (in array<double> u, in array<double> v,
        in double tol)
    throws      sidl.PreViolation, NegativeValueException, sidl.PostViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v); tol >= 0.0;
    ensure      no_side_effects : is pure;

static bool triangleInequalityHolds (in array<double> u, in array<double> v,
        in double tol)
    throws      sidl.PreViolation, NegativeValueException, sidl.PostViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v); tol >= 0.0;
    ensure      no_side_effects : is pure;

static double norm (in array<double> u, in double tol, in int badLevel)
    throws      sidl.PreViolation, NegativeValueException, sidl.PostViolation;
    require     u != null; dimen(u) == 1; tol >= 0.0;
    ensure      no_side_effects : is pure;
            result >= 0.0; nearEqual(result, 0.0, tol) iff isZero(u, tol);
```

Figure 6.15: Interface contracts for trial **VT** methods (Part 1), where linear-time assertions appear in italics and violation assertions in boldface type.

```
static double dot (in array<double> u, in array<double> v,
        in double tol, in int badLevel)
    throws      sidl.PreViolation, sidl.PostViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v); tol >= 0.0;
    ensure      no_side_effects : is pure;
                areEqual(u, v, tol) implies (result >= 0.0);
                (isZero(u, tol) and isZero(v, tol))
                        implies nearEqual(result, 0.0, tol);

static array<double> product (in double a, in array<double> u,
        in int badLevel)
    throws      sidl.PreViolation, sidl.PostViolation;
    require     u != null; dimen(u) == 1;
    ensure      no_side_effects : is pure;
          result != null; dimen(result) == 1; size(result) == size(u);

static array<double> negate (in array<double> u, in int badLevel)
    throws      sidl.PreViolation, sidl.PostViolation;
    require     u != null; dimen(u) == 1;
    ensure      no_side_effects : is pure;
          result != null; dimen(result) == 1; size(result) == size(u);

static array<double> normalize (in array<double> u, in double tol,
        in int badLevel)
    throws      sidl.PreViolation, DivideByZeroException, sidl.PostViolation;
    require     u != null; dimen(u) == 1; tol >= 0.0;
    ensure      no_side_effects : is pure;
          result != null; dimen(result) == 1; size(result) == size(u);

static array<double> sum (in array<double> u, in array<double> v,
        in int badLevel)
    throws      sidl.PreViolation, sidl.PostViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
                size(u) == size(v);
    ensure      no_side_effects : is pure;
          result != null; dimen(result) == 1; size(result) == size(u);

static array<double> diff (in array<double> u, in array<double> v,
        in int badLevel)
    throws      sidl.PreViolation, sidl.PostViolation;
    require     u != null; dimen(u) == 1; v != null; dimen(v) == 1;
    ensure      no_side_effects : is pure;
          result != null; dimen(result) == 1; size(result) == size(u);
```

Figure 6.16: Interface contracts for trial **VT** methods (Part 2), where linear-time assertions appear in italics and violation assertions in boldface type.

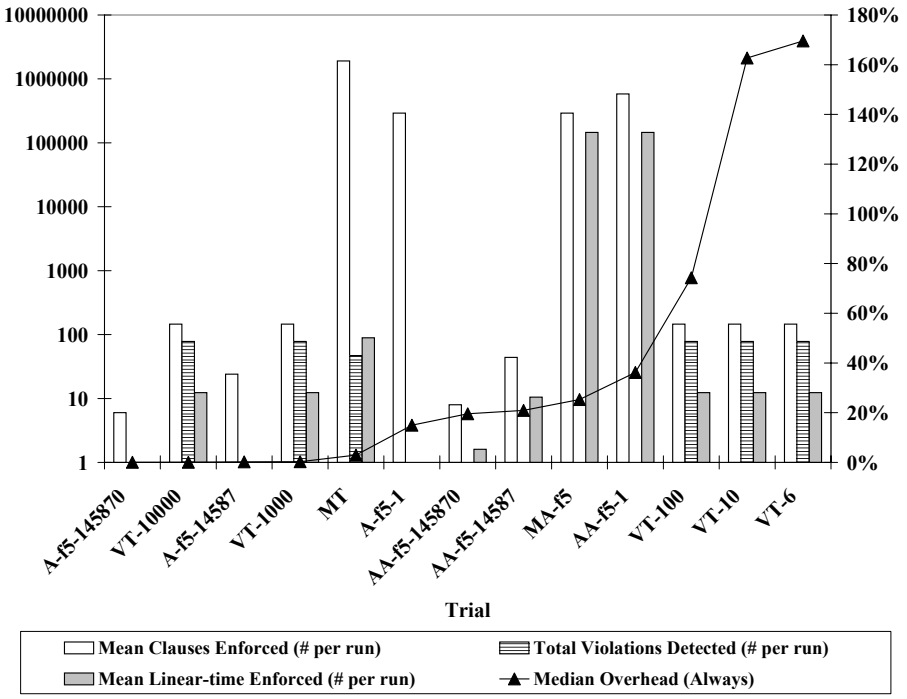

Figure 6.17: *Global trace* study results for the *Always* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program (**A**, **MA**, **AA**, **MT**, and **VT**) with, as appropriate, the input file (**f5**) and/or input array size.

of its clauses contain solely constant-time assertions, 91% of its violations are in linear-time assertions within postcondition clauses.

6.3.7 Results

Results from and a summary of the analysis of experiments based on global enforcement using trace-based execution time estimates are presented in this section for thirteen trials. Baseline data and overhead metrics are shown for the *Always* policy. Results for traditional sampling policies are followed by those of the performance-driven enforcement policies. Figures present trial metrics, relative to baseline policies, in order by the enforcement overhead of the *Always* policy.

Figure 6.17 illustrates the three metrics for the *Always* policy. The range of mean contract clauses checked across trials is 6 to 1,909,217 per run. Of those, between 0 and 145,871 involve linear-time assertions. Five of the thirteen trials incur 3% or less enforcement overhead. The median overheads for another five trials range from 15% to

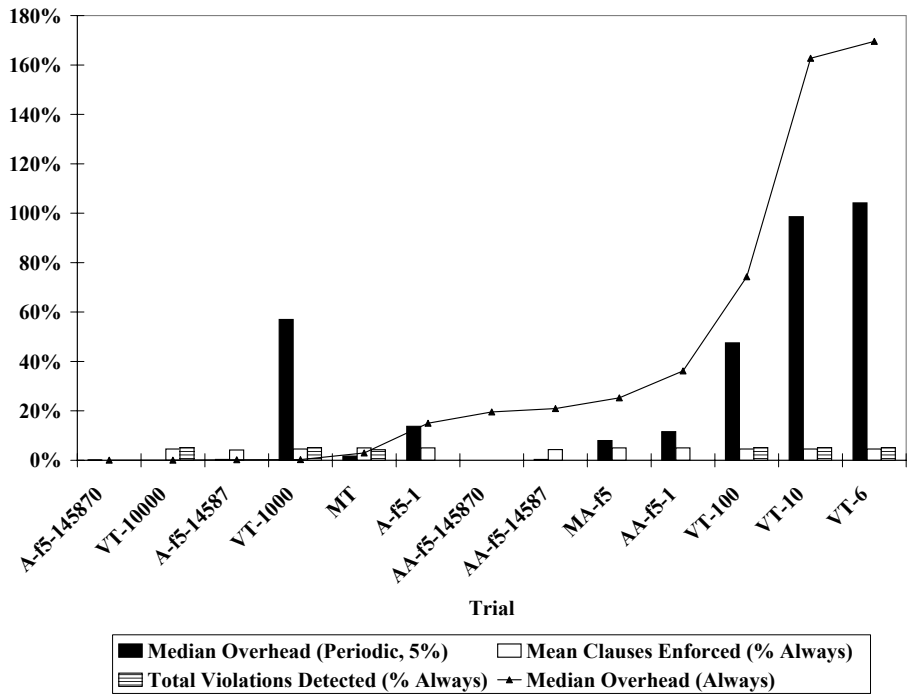

Figure 6.18: *Global trace* study results for the *Periodic* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and relevant input file and/or input array size as described in the caption of Figure 6.17.

36%. The last three trials incur between 74% and 170% median enforcement overhead. A range of between one and seventy-eight contract clauses are violated per run, with the exception of no violations detected in trial **MA-f5**. As discussed in Section 6.3.6, the numbers of violations are tied to the programs not the trials.

Results for the *Periodic* policy are presented in Figure 6.18. Enforcement overheads are 8% or less for seven of the trials. Four trials, however, incur a hefty 48% to 104% overhead. This effect is thought to result from insufficient work being performed within the methods to mitigate the overhead of the enforcement decision process and/or checking the (possibly linear-time) contracts enforced by the policy. The mean number of contract clauses enforced is between 4% and 5% on all but the two trials totaling less than ten contract clauses per run. Finally, the policy detects 4% to 5% of the violations in six trials, which translates into 2 to 4 violations per run.

Figure 6.19 illustrates enforcement metrics for the *Random* policy. The median enforcement overhead is 8% or less in seven of the thirteen trials. However, the overhead

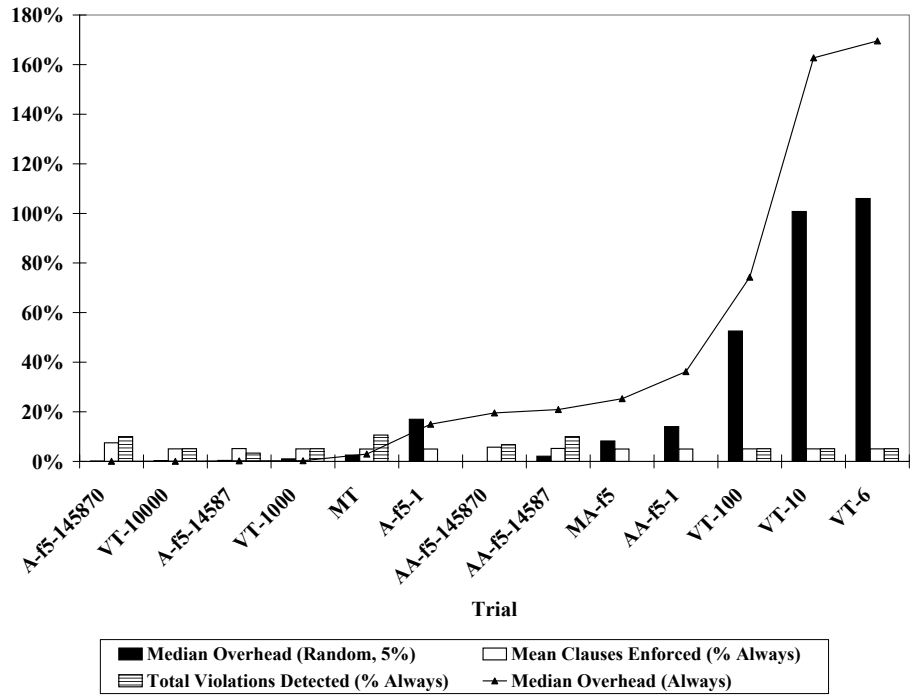

Figure 6.19: *Global trace* study results for the *Random* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and relevant input file and/or input array size as described in the caption of Figure 6.17.

goes as high as 53% to 106% for three trials. Once again, the excess overhead is attributable to a combination of the costs of going through the enforcement routines and the execution time required to check (possibly linear-time) contract clauses sampled by the policy. Contract clause enforcement ranges from 5% to 8% across trials, where it exceeds 5% only in the two trials involving less than ten clause checks. Although it detects no violations in the two program **A** and **AA** trials with the smallest input size, the *Random* policy does detect between 3% and 11% of the violations in the remaining ten trials having faults leading to interface contract clause violations. So the *Random* policy generally performs slightly more checking and detects the same or more violations in these trials than the *Periodic* policy. However, the improvements generally result in either no increase or up to 5% increase in overhead except. An exception exists in the case of trial *VT-1000*, which incurs 56% less overhead.

Results for the *Adaptive timing* policy are presented in Figure 6.20. Median enforcement overheads are 8% or less in eight of the thirteen trials. However, the overheads

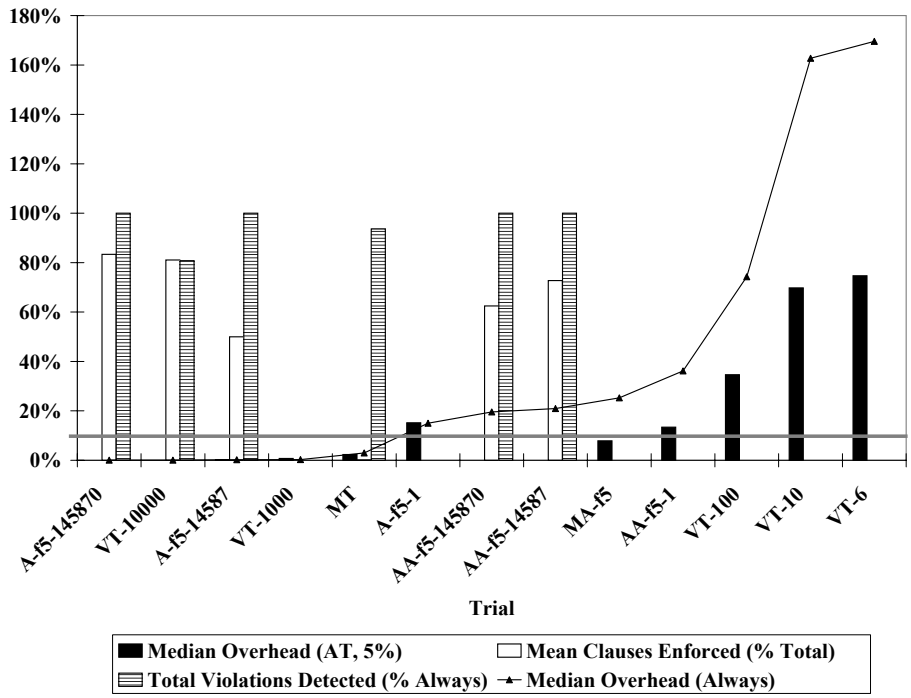

Figure 6.20: *Global trace* study results for the *Adaptive timing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and relevant input file and/or input array size as described in the caption of Figure 6.17.

go as high as 35% to 75% in three trials. Contract clause checks range from 50% to 83% in five of the trials. There are few, if any, clause checks in the remaining eight trials. Detected violations in six of the trials range from 81% to 100%, where it is 100% in four of the six trials. Interestingly, 94% of the violations are detected in trial **MT**, despite a negligible level of contract checking and 91% of the violations being of linear-time assertions. The *Adaptive timing* policy also incurs negligible overhead while checking significant numbers of contract clauses and detecting significant numbers of violations for five of the trial sets — or one more than the *Always* policy.

Figure 6.21 illustrates enforcement metrics for the *Adaptive fit* policy. Median enforcement overhead is 11% or less in eight of the thirteen trials. Three of the trials, however, incur 66% to 137% median overhead. The mean number of contract clauses enforced ranges from 27% to 100% in nine of the thirteen trials, but negligible to no checking in the remaining trials. Violations are detected in ten trials at a rate of 6% for **MT**, 33% to 50% for the three program **VT** trials with the smallest input array sizes, and

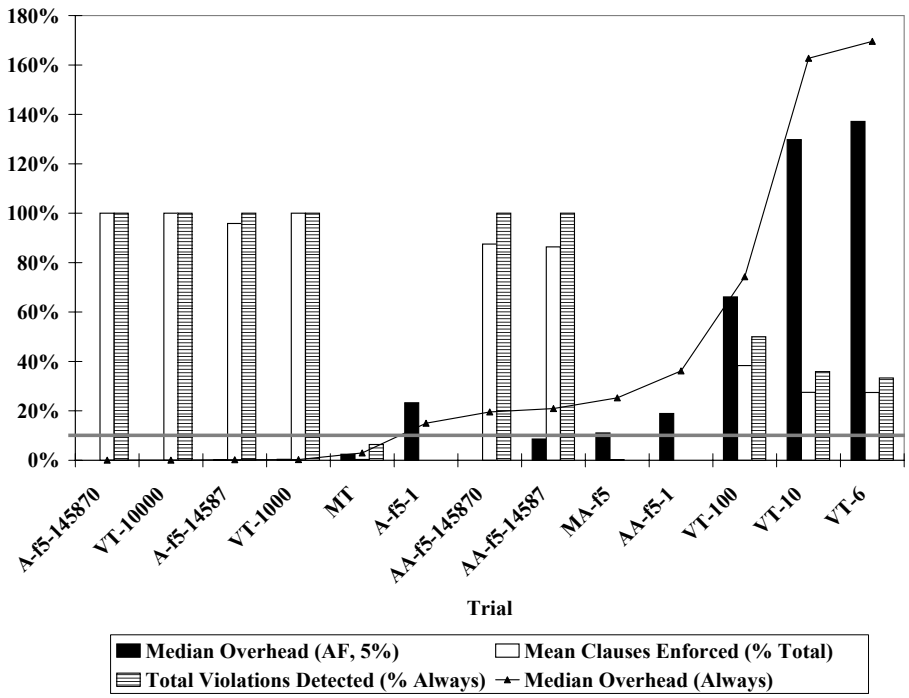

Figure 6.21: *Global trace* study results for the *Adaptive fit* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and relevant input file and/or input array size as described in the caption of Figure 6.17.

100% for the remaining six trials. So the *Adaptive fit* policy also incurs negligible overhead while checking significant numbers of contract clauses and detecting significant numbers of violations for five of the trial sets — including the same four trials as the *Always* policy.

Finally, results for the *Simulated annealing* policy are shown in Figure 6.22. Seven of the thirteen trials incur 9% or less median overhead. Three other trials incur between 38% and 66% overhead, while the remaining three trials incur 81% to 133% median overhead. The range of mean contract clause checks is from 86% to 100% for six of the thirteen trials, though 20% to 35% are checked in three other trials. All violations are detected in the six trials with the most clauses checked. Three more trials have 29% to 36% of their violations detected with the *Simulated annealing* policy. Only 6% of trial **MT**'s violations are detected. So the *Simulated annealing* policy is able to detect violations in ten of the thirteen trials while incurring negligible overhead in five of them. Unfortunately, the overhead in six of the trials is still excessive. This may be due in part to the relative cost of the calculations required to make the enforcement decision.

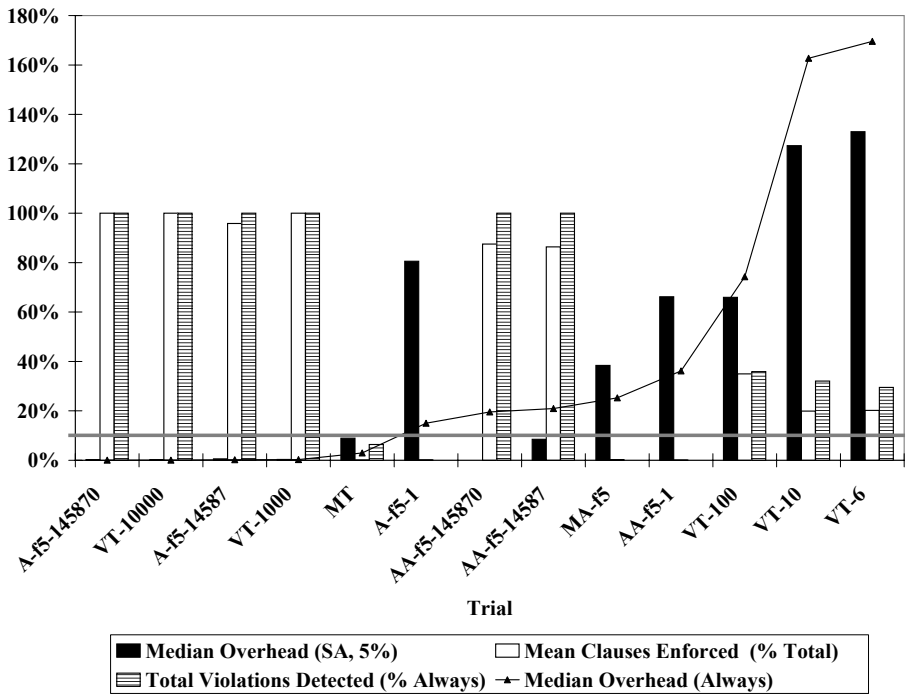

Figure 6.22: *Global trace* study results for the *Simulated annealing* policy. Overhead is relative to results using the *Never* policy. Trial set names are formed by the concatenation of the program and relevant input file and/or input array size as described in the caption of Figure 6.17.

Results for the *Always* policy indicate only four of the thirteen trials incur negligible median enforcement overhead and a fifth incurs only 3% overhead. Two of the trials incur over 160% median enforcement overhead. The *Periodic* policy is able to detect 4% to 5% of the violations in each of six trials but incurs 48% or more overhead in four of them. Incurring 53% or more enforcement overhead in only three of the ten trials in which it detects between 3% and 11% of the violations, the **Random** policy generally performs better in these trials. However, all three of the performance-driven enforcement policies do very well in at least five of the thirteen trials. The *Adaptive timing* policy detects 81% or more of the violations while incurring no more than 2% enforcement overhead for six of the trials. Similarly, the *Adaptive fit* policy detects 100% of the violations while incurring no more than 9% enforcement overhead for six trials. *Adaptive fit* detects violations in four additional trials as well. The *Simulated annealing* policy also detects violations in ten trials, with 100% of the violations being detected in the same trials as *Adaptive fit*. The *Simulated annealing* policy also incurs no more than 9% enforcement overhead in

those trials. So while interface contract enforcement sampling policies tend perform well on about half the trials, the performance-driven policies does very well in terms of detecting violations in six of the trials.

6.3.8 Discussion

An analysis of the results for the *Global trace* study reveals a couple of patterns and surprises. Key patterns relate to the ties between enforcement overhead reported in the traces versus sampling results. A major surprise is the enforcement effects for the **MT** trial.

The *Always* policy and two of the three performance-driven policies — *Adaptive fit* and *Adaptive timing* — incur negligible overhead while detecting 100% of the violations in the same four trials. According to the enforcement traces, those trials are to incur 6% or less overhead. So the performance-driven policies performed as well as full enforcement in those cases. Two of those trials — **VT-1000** and **VT-10000** — have 5% to 8% of their contract clauses containing linear-time assertions.

Enforcement traces for trials **AA-f5-145870** and **AA-f5-14587** indicate over 80% of the execution time is spent in the methods and less than 20% enforcing contracts. The *Always* policy incurs 20% to 21% overhead in both trials while the *Adaptive timing* policy is able to detect 100% of the violations while incurring negligible overhead. The policy is able to detect the violations because they occur only in the class of contracts checked by the policy. That is, all of the violations occur in contracts whose execution time estimates are within 5% of the estimates for their methods. The *Adaptive fit* and *Simulated annealing* policies also detect 100% of the violations in those trials; however, they incur 9% enforcement overhead. The extra overhead is attributable to the ability of both policies to check contracts with execution time estimates in excess of the overhead limit applied to method execution time estimates. Therefore, all three performance-driven policies perform better than the *Always* policy in the two trials since the violations occur in contracts whose execution time estimates are low relative to the estimates of the methods.

According to the enforcement trace for trial **MT**, about 50% of its execution time is spent in the method and the remaining time split between contracts and the program.

However, the *Always* policy actually incurs a median overhead of only 3% while checking 1,909,217 contract clauses per run. Since estimates for most of the offending clauses are within 5% of the estimates for their methods, the *Adaptive timing* policy is able to detect 94% of the violations even though it checks only 688 contract clauses per run. This result actually differs from the findings in the previous two studies. That is, the previous studies seem to indicate the presence of linear-time assertions in contract clauses leads to a large increase in enforcement overhead. However, inspection of the source code reveals the program includes a subset of tests operating on a very small, hard-coded data structure not tied to the size of the input file.

The trials representing tight program loops — **A-f5-1**, **MA-f5**, and **AA-f5-1** — are to incur about 20% to 25% enforcement overhead according to the traces. The *Always* policy actually incurs 15% to 36% enforcement overhead. The *Adaptive fit* and *Adaptive timing* policies perform slightly better but check few if any contract clauses.

The final three trials — all from program **VT** with the smallest input array sizes — incur considerable overhead with all enforcement policies. Enforcement trace executions, shown in Figure 6.12, provide some clues. Between 80% and 84% of each trial's execution times are attributed to the program and component methods, leaving the remaining 16% to 20% of the time on contract enforcement. In all three trials, times spent enforcing preconditions alone meet or exceed those spent on component methods. While postcondition trace execution times are lower than the times attributed to methods, they incur roughly half or more of the execution time attributed to methods. For example, the breakdown for trial **VT-10** is: 72% of the trace execution time on program statements; 7% of the time in component methods; 15% of the time enforcing preconditions; and 6% of the time enforcing postconditions. So, even though enforcement tracing attributes 72% to 74% of the trial execution times to program statements, none of the sampling policies are able to adapt well to the overhead.

Using *a priori* execution cost estimates obtained from trial-specific enforcement traces enables the performance-driven enforcement policies to better adjust their enforcement levels and detect violations for roughly half of the trials in this study. The *Adaptive timing* and, especially, the *Adaptive fit* enforcement policies appear to perform very well on trials whose contracts are relatively fast to enforce or whose contracts may be moderately

fast to enforce but the trial spends the majority of its time within the component methods.

6.3.9 Review

A total of thirteen trials are formed from five programs and several input sets. Enforcement tracing experiments are used to obtain the *a priori* execution time estimates guiding the three performance-driven enforcement policies. In addition to determining the nature of checked contract clauses and the effects of the new enforcement approach, a key goal of this study is to determine if refined execution time estimates restore the performance-driven enforcement policies' ability to detect interface contract violations.

Findings indicate the performance-driven enforcement policies — especially *Adaptive timing* and *Adaptive fit* — are able to better adjust their enforcement levels to the trial with refined execution time estimates. The policies are also better able to detect significant numbers of violations in trials with certain characteristics. In particular, the policies tended to favor trials involving contracts whose execution costs are relatively inexpensive. One or more of the policies also perform well with moderately expensive contracts — according to enforcement trace results — relative to the enforcement overhead limit. Interestingly, all sampling policies encounter difficulties containing enforcement overhead with the three trials spending about 10% or less of their time in the component methods.

6.4 Summary

This chapter describes results from two studies of global, performance-driven interface contract enforcement using *a priori* execution cost estimates. The centralized approach taken in these studies is intended to provide better control over enforcement overhead across methods and components than the decentralized technique used in the *Local* study. While much of the enforcement infrastructure and work flow are common between the two global enforcement studies, the second study relies on refined execution time estimates obtained from enforcement tracing features added to the middleware. The two studies also use different trials to investigate the effects of the same sampling strategies on enforcement overhead, contract clause checking, and violation detection.

Findings from both studies indicate the performance-driven enforcement policies

— especially *Adaptive timing* and *Adaptive fit* — automatically adjust the level of contract clause enforcement to the programs. They also tend to have better overall control over the enforcement overhead though the quality of the execution time estimates does make a difference. While the coarse estimates in the *Global simple* study allowed the performance-driven policies to keep the overhead closer to the 10% level, they precluded the detection of violations. The refined estimates in the *Global trace* study enabled the performance-driven policies to detect significant numbers of violations in roughly half of the trials. However, the benefits seem to be limited to trials enforcing contract clauses whose checking is fast to moderately expensive relative to the amount of time spent in component methods.

Chapter 7

Summary

The goal of this research is to help scientists gain confidence in software built from emerging, plug-and-play component technologies through specialized interface contract enforcement. The vision of scientists developing insights into and predictions about physical phenomena through these technologies is based on the idea of a repository of compatible components. Components can be used in a plug-and-play manner only to the extent they conform 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, the performance overhead concerns permeating the scientific computing community are a roadblock to the adoption of technologies providing these capabilities.

So this research proposes the use of and investigates the effects of performance-driven sampling as a means of controlling the impact of interface contract enforcement on program execution time. The new policies are intended for use in applications making numerous calls to methods with contracts that are expected to incur unacceptable overhead from enforcing all contracts. The guiding principle is to adjust the level of enforcement to the program based on performance overhead constraints. This essentially means interface contract checking is automatically reduced when the costs are considered too high and increased (when feasible) when the costs are below a given tolerance. While such an approach cannot guarantee detection of all contract violations, the traditional alternative is running deployed applications without enforcement.

Three studies are conducted to empirically evaluate performance-driven interface

Table 7.1: Comparison of interface contract enforcement study approaches.

Study	Synopsis	
	Pros	**Cons**
Local	Enforcement decisions made on a per-method basis; performance-driven enforcement policies rely on runtime timing instrumentation.	
	No *a priori* execution time estimates required.	Approach does not control overhead across methods. Timing instrumentation is excessive when all contracts are constant-time.
Global simple	Global enforcement decisions made using *a priori* execution time estimates obtained from simple timing experiments.	
	Global enforcement approach better mitigates execution time costs across methods.	Approach requires execution time estimates for methods and contract clauses. Execution time estimates based on simple timing experiments are not sufficiently accurate to tailor enforcement decisions to trials.
Global trace	Global enforcement decisions are made using *a priori* execution time estimates obtained from enforcement traces.	
	Refined execution time estimates better tailor enforcement decisions to the program and input set.	Approach requires preliminary enforcement tracing experiments to obtain accurate execution time estimates.

contract enforcement: *Local*, *Global simple*, and *Global trace*. Measures are taken to determine the overhead of enforcement, contracts checked, and violations detected using baseline and sampling policies. Results are then compared to gain insights into the value of the different enforcement strategies under study. Table 7.1 summarizes the approaches used in the studies and highlights pros and cons of each.

The investigation begins with the introduction of a single policy relying on runtime instrumentation for establishing execution times of contracts and methods. Contract enforcement decisions are made on a per-method basis to establish local countdowns used to guide enforcement decisions. Findings indicate performance-driven enforcement can automatically adjust the level of contract checking to the program while detecting significant numbers of violations across trials. However, there appears to be a tendency to incur excessive overhead with programs making numerous calls to methods whose contracts consist solely of constant-time assertions. The performance effects seem to be mitigated somewhat when an equal number of methods with contracts including linear-time assertions are invoked. The excessive overhead is not present when all contracts are enforced, so the

runtime timing instrumentation appears to be a factor.

So the idea of using *a priori* execution cost estimates arises out of the first study. The effects associated with assertion complexity bring out another idea; namely, the desire to better understand the characteristics of contract checks performed for a program. Finally, conceptually, the ability to manage performance-driven enforcement with the local decisions is in question. Consequently, the remaining studies pursue global enforcement decisions using *a priori* execution time estimates.

The technique employed to establish estimates for the second, or *Global simple* study, is based on the idea of using test cases as sources of execution time data. Differences between timing runs with all contracts enforced and runs with contract enforcement bypassed establish the performance overhead costs. An algorithm assigns times to each method's contract clauses based on characteristics of the contained assertions. While the strategy increases the number of contract checks across the board and generally keeps the mean performance overhead below the target level, the execution time estimates preclude detecting violations in the trials.

The final study (*Global trace*) pursues trial-specific execution time estimates to tailor performance-driven, global enforcement decisions. The necessary data is collected through interface contract enforcement tracing features added to the experimental Babel toolkit. Findings indicate the more accurate estimates available from tracing provide better control over enforcement overhead and improve the ability to detect violations.

Hence, performance-driven interface contract enforcement appears to be a viable alternative to the traditional strategy of disabling enforcement for applications exhibiting target characteristics. Accurate execution time estimates facilitate adjusting the level of contract checking to the program. Programs executing methods whose contracts contain assertions which are moderate to fast to check relative to the time spent executing component methods appear to benefit most from performance-driven strategies.

Appendix A

Glossary

A — Array. A program that retrieves an array of face entities at a time from the mesh.

AA — Array Adjacency. A program that retrieves an array of face entities at a time, retrieving their adjacent vertexes before getting the next array of faces from the mesh.

Adaptive fit — AF. The performance-driven enforcement policy that allows contract checks when the accumulated amount of time spent so far in executing contracts is within the overhead tolerance applied to the total amount of time spent on the program and within methods.

Adaptive timing — AT. The first performance-driven enforcement policy, *AT* allows contract checks when the amount of time spent executing the contract (or contract clause) is within the overhead tolerance applied to the amount of time spent on the method.

ADL — Assertion Definition Language [160]. An extension of CORBA IDL that supports postconditions.

AF — See *Adaptive Fit*.

`all()` — A built-in assertion function that evaluates an expression required to hold for all elements of a SIDL array. Refer to Table 3.1 for more information.

Always — The interface contract enforcement policy or frequency option used to check all contracts encountered during program execution.

ANSI — American National Standards Institute.

`any()` — A built-in assertion function used to determine if any of the elements of a SIDL array satisfy the expression. Refer to Table 3.1 for more information.

APPC — Annotation Preprocessor for C [157].

ASL — Architecture Specification Language [27, 106]. A family of design languages for component-based software engineering.

AT — See *Adaptive timing*.

Babel — The language interoperability toolkit that is leveraged for this research. Babel [111] actually consists of SIDL, a compiler, and a runtime library. Babel is often used to refer to the compiler.

CBSE — Component-based Software Engineering.

CCA — Common Component Architecture [31].

CCTTSS — Center for Component Technology for Terascale Simulation Software [176], which was replaced by the *TASCS*.

Class invariants — See *Invariants*.

Contract — See *Interface contract*, *Preconditions*, *Postconditions* and *Invariants*.

Contract checks — Software statements automatically added to the language interoperability middleware — in `check_method` routines illustrated in Figure 3.3 — that check, or enforce, the assertions specified in the associated interface contract clause.

Contract clause — A set of assertions that must evaluate to true at a predetermined location within the execution of a class or method. See also *Invariants*, *Preconditions*, and *Postconditions*.

Contracts enforced — One of three metrics calculated from raw experiment data. The number is given as a total for all experiments for the given trial (or trial set) when reported for the *Always* policy but as a percentage of the total for interface contract enforcement sampling policies.

CORBA — Common Object Request Broker Architecture.

`count()` — A built-in assertion function used to count the number of elements in a SIDL array whose values satisfy the expression. Refer to Table 3.1 for more information.

`dimen()` — Dimension. A built-in assertion function that returns the dimension of a SIDL array. Refer to Table 3.1 for more information.

DOE Department of Energy (United States).

Enforcement overhead See *Overhead*.

Enforcement policy See *Interface contract enforcement policy*.

Entry point vector EPV. A vector, or array, that contains pointers to methods (or routines). It is a C programming language data structure used by the Babel-generated middleware to direct a method call made by a callee to the corresponding implementation of the method provided by the caller. An illustration of the relevant EPVs is provided in Figure 3.3.

EPV See *Entry point vector*.

Execution profile The term is used to refer to a sequence of time values that result from the execution of a program. Refer to Section 1.2 for more information.

Function Another term for a method that returns a value.

GRUMMP Generation and Refinement of Unstructured, Mixed-Element Meshes in Parallel [76].

IDL Interface Definition Language. Commonly used to refer to CORBA IDL [138].

Interface contract Obligations on the caller and callee of a method. In the former case, the obligations are specified in a precondition clause while the latter are specified in a postcondition clause. Additionally, a component (or class) may have invariants that must hold both before and after the method call. Also see *Preconditions*, *Postconditions*, and *Invariants*.

Interface contract enforcement policy The set of options used to make interface contract enforcement decisions, including the option to by-pass enforcement altogether.

Intermediate object representation IOR. A common representation of a component (or class) generated in ANSI C by the Babel compiler that leverages object-oriented inheritance of basic SIDL interface and class features.

Invariants The contract clause that contains assertions specifying properties that must be true before and after its methods are called from the time a class instance is initialized through its termination.

IOR See *Intermediate Object Representation*.

`irange()` Integer range. A built-in assertion function for checking an argument against a range of values. Refer to Table 3.1 for more information.

ISL Interface Specification Language [27, 106]. The language extends CORBA IDL with preconditions, postconditions, invariants, and protocol (or states).

ITAPS The Interoperable Technologies for Advanced Petascale Simulations Center [96, 97], which replaced the *TSTT*.

Jass Java with Assertions [99]. A preprocessor that supports Design-by-Contract for the Java language.

JML Java Modeling Language [115]. A modeling language for Java that supports preconditions and postconditions.

LLNL Lawrence Livermore National Laboratory [113].

`lower()` A built-in assertion function used to obtain the lower index of a given dimension of a SIDL array. Refer to Table 3.1 for more information.

MA Mesh Adjacency. A program that retrieves all face entities from a mesh before retrieving, on an individual face basis, the corresponding vertex entities.

`max()` Maximum. A built-in assertion function used to obtain the maximum value in a SIDL array. Refer to Table 3.1 for more information.

Method A method is the object-oriented equivalent of a (sub)routine. It may or may not return a value (like a function).

Method calls The interface contract enforcement classification option that corresponds to contract clauses that include at least one method, or function, call.

Metrics See *Overhead*, *Contracts enforced*, and *Violations detected*.

`min()` Minimum. A built-in assertion function used to obtain the minimum value in a SIDL array. Refer to Table 3.1 for more information.

MT Mesh Test. A program that exercises most of the capabilities defined in the TSTT/ITAPS mesh specification.

NA Not Applicable.

`nearEqual()` A built-in assertion function used to ensure two arguments are equal within a tolerance value. Refer to Table 3.1 for more information.

Never The interface contract enforcement policy (and enforcement frequency option) used to completely by-pass the middleware check routines illustrated in Figure 3.3.

`nonDecr()` Non-decreasing. A built-in assertion function used to ensure the values of elements in an array are in non-decreasing order. Refer to Table 3.1 for more information.

`none()` A built-in assertion function used to ensure none of the elements of a SIDL array satisfy the expression. Refer to Table 3.1 for more information.

`nonIncr()` Non-increasing. A built-in assertion function used to ensure the values of elements in an array are in non-increasing order. Refer to Table 3.1 for more information.

OCL Object Constraint Language [139]. A textual language for expressing modeling constraints.

Overhead The execution (or performance) overhead attributable to interface contract enforcement. One of three metrics calculated from raw experiment data, *overhead* represents the average amount of execution time with interface contracts being enforced for a given policy above the amount of time when interface contract enforcement is being by-passed.

Performance overhead See *Overhead*.

Periodic The interface contract enforcement frequency option used to check contracts at a user-specified interval.

Policy See *Interface contract enforcement policy*.

Postconditions Assertions within the postcondition clause of a method's contract. Also serves as the contract classification (`sidl.ContractClass`) option for enforcing only postcondition clauses.

Preconditions Assertions within the precondition clause of a method's contract. Also serves as the contract classification (`sidl.ContractClass`) option for enforcing only precondition clauses.

Random The interface contract enforcement frequency option for checking contracts on a random basis within a user-specified range of random numbers.

`range()` A build-in assertion function that checks that an argument is in a specified range. Refer to Table 3.1 for more information.

RISC Run-time Interface Specification Checker [89].

SA See *Simulated Annealing*.

SciDAC Scientific Discovery through Advanced Computing [178].

SIDL Scientific Interface Definition Language. SIDL is the programming language-neutral interface specification language used by Babel. As such, it defines the object-oriented hierarchy implemented through the SIDL runtime library and the Babel compiler-generated language interoperability wrappers. More information can be found in [44].

`sidl.-ClauseType` An enumeration used to identify valid interface contract clause types. It was introduced for use in the global enforcement studies. Table 6.1 lists the valid values.

`sidl.-ContractClass` An enumeration used to identify interface contract classifications for enforcement purposes. Introduced for use in the global enforcement studies, the classifications identify a characteristic of the assertions contained within a contract clause. Table 6.1 lists the valid values. See also *Constant*, *Invariants*, *Linear*, *Method calls*, *Postconditions*, *Preconditions*, *Results*, and *Simple expressions*.

`sidl.-EnforceFreq` An enumeration for valid enforcement frequency options added to the experimental Babel toolkit's SIDL runtime library for the global enforcement studies. Refer to Table 6.1 for the list of values. See also *Never*, *Always*, *Adaptive fit*, *Adaptive timing*, *Periodic*, *Random*, and *Simulated annealing*.

Simple expression The interface contract enforcement classification option that corresponds to contract clauses consisting solely of simple expressions (i.e., no method calls or built-in function calls).

Simulated annealing The interface contract enforcement frequency option that allows some checks to exceed the user-specified overhead limit but with decreasing frequency over time.

`size()` A built-in assertion function used to obtain the allocated size of a SIDL array. Refer to Table 3.1 for more information.

SLOC Source Lines of Code. Version 2.26 of SLOCCount [188] was used to calculate language-specific source statements.

`stride()` A built-in assertion function used to obtain the stride of the specified dimension of a SIDL array. Refer to Table 3.1 for more information.

`sum()` A built-in assertion function used to obtain the total of the values of all elements in the SIDL array. Refer to Table 3.1 for more information.

TASCS Center for Technology for Advanced Scientific Component Software [177], which replaces *CCTTSS*.

Trial A combination of program, component, and input set used in experiments. Multiple runs of the trial are performed for each of the interface contract enforcement policies under study to collect data for metrics computations.

TSTT Terascale Simulation Tools and Technologies Center [173], which was replaced by *ITAPS*.

`upper()` A built-in assertion function used to obtain the upper index for the given dimension of a SIDL array. Refer to Table 3.1 for more information.

VT Vector Test. A program that exercises all of the vector methods and deliberately violates each precondition and postcondition clause.

Bibliography

[1] IEEE Std 610.12-1990. *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.

[2] J. Mack Adams, James Armstrong, and Melissa Smartt. Assertional checking and symbolic execution: An effective combination for debugging. In *Proceedings of the 1979 annual conference*, pages 152–156, 1979.

[3] Assertion definition language. `http://adl.opengroup.org/index.html`. Visited 2004.

[4] Yuri Alexeev, Benjamin A. Allen, Robert C. Armstrong, David E. Bernholdt, Tamara L. Dahlgren, Dennis Gannon, Curtis L. Janssen, Joseph P. Kenny, Manohkumar Krishnan, James A. Kohl, Gary Kumfer, Lois Curfman McInnes, Jarek Nieplocha, Steven G. Parker, Craig Rasmussen, and Theresa L. Windus. 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.

[5] Paul D. Amer and Lillian N. Cassel. Management of sampled real-time network measurements. In *Proceedings of the 14th IEEE Conference on Local Computer Networks*, pages 62–68, October 1989.

[6] Dorothy M. Andrews and Jeoffrey P. Benson. An automated program testing methodology and its implementation. In *Proceedings of the 5th International Conference on Software Engineering*, pages 254–260, March 1981.

[7] Robert Armstrong, David E. Bernholdt, Tammy Dahlgren, Wael R. Elswasif, Gary Kumfert, Lois Curfman McInnes, Jarek Nieplocha, and Boyana Norris. High end computing component technology (white paper). In *Workshop on the Road Map for the Revitalization of High End Computing*, Washington, DC, USA, 2003.

[8] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 168–179, May 2001.

[9] Thomas Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, July 2000.

[10] Mike Barnett and Wolfram Schulte. The ABCs of specification: ASML, behavior, and components. *Informatica*, 17, 2002. To appear.

[11] Victor R. Basili and Barry Boehm. COTS-based systems top 10 list. *IEEE Computer*, 34(5):91–95, May 2001.

[12] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[13] Benoit Baudry, Yves Le Traon, and Jean-Marc Jézéquel. Robustness and diagnosability of OO systems designed by contracts. In *Proceedings of the 7th International Software Metrics Symposium*, pages 272–284, 2001.

[14] Friedrich W. Beichter, Otthein Herzog, and Heiko Petzsch. SLAN-4 — a software specification and design language. *IEEE Transactions on Software Engineering*, SE-10(2):155–162, March 1984.

[15] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, January 1995.

[16] David E. Bernholdt, Benjamin A. Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. McInnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. A component architecture for high-performance scientific computing. *International Journal of High-Performance Computing Applications, ACTS Collection special issue*, 20(2):163–202, 2006.

[17] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[18] T. J. Biggerstaff. Reuse technologies and their niches. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 613–614, 1999.

[19] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.

[20] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, January 2001.

[21] Cristina Boeres, Alexandre Lima, and Vinod E. F. Rebello. Hybrid task scheduling: Integrating static and dynamic heuristics. In *Proceedings of the* 15^{th} *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, pages 199–206, November 2003.

[22] Jihad Boulos and Kinji Ono. Cost estimation of user-defined methods in object-relational database systems. *SIGMOD Record*, 28(3):22–28, September 1999.

[23] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Library functions timing characterization for source-level analysis. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE '03)*, pages 1132–1133, 2003.

[24] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 70–80, July 2002.

[25] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering*, pages 86–95, May 2004.

[26] Monica Brockmeyer, Franam Jahanian, Constance Heitmeyer, and Bruce Labaw. An approach to monitoring and assertion-checking of real-time specifications. In *Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems*, pages 236–243, April 1996.

[27] Francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Ásgeir Ólafsson, and John W. Wetterstrand. Toward software plug-and-play. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 19–29, Boston, MA, May 17–20, 1997.

[28] David Brown, Lori Freitag, and Jim Glimm. 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*, pages 57–61, Honolulu, HI, June 3–6, 2002. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-PRES-151494, Livermore, CA, 2002.

[29] J. C. Carver, L. M. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post. Observations about software development for high end computing. *CTWatch Quarterly*, 2(4), November 2006.

[30] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[31] Common Component Architecture (CCA) Forum. `http://www.cca-forum.org/`.

[32] T. Y. Chen, Jianqiang Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, pages 327–333, August 2002.

[33] Wen-Tsuen Chen, Jone-Ping Ho, and Chia-Hsien Wen. Dynamic validation of programs using assertion checking facilities. In *The IEEE Computer Society's 2nd International Computer Software and Applications Conference*, pages 533–538, November 13–16, 1978.

[34] Trishul M. Chilimbi and Matthias Hauswirth. 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*, pages 156–164, Boston, MA, October 9–13, 2004.

[35] P. L. Chiu, Y. T. Chen, and K. H. Lee. A request scheduling algorithm to support flexible resource reservations in advance. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, volume 4, pages 1971–1974, May 2004.

[36] Cynthia Della Torre Cicalese and Shmuel Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, July 1999.

[37] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. In *Conference Proceedings on Communications, Architectures, Protocols and Applications*, pages 194–203, September 13–17, 1993.

[38] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, May 2005.

[39] P. Collet, A. Ozanne, and N. Rivierre. Enforcing different contracts in hierarchical component-based systems. In *Proceedings of the 5th International Symposium on Software Composition (SC '06)*, pages 50–65, Vienna, Austria, March 25–26 2006.

[40] P. Collet and R. Rousseau. Towards efficient support for executing the object contraint language. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pages 399–408, 1999.

[41] Dennis W. Cooper. Adaptive testing. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 102–105, 1976.

[42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001. Second Edition.

[43] Bill Curtis. Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68(9):1144–1157, September 1980.

[44] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, California, version 0.10.0 edition, March 2005.

[45] Tamara L. Dahlgren. Adaptive enforcement of interface assertions. Technical Report UCRL-PRES-211741, Lawrence Livermore National Laboratory, Livermore, California, April 2005. Presentation at the Common Component Architecture Forum's Spring 2005 meeting in Lincoln City, Oregon, USA.

[46] Tamara L. Dahlgren. Babel assertion and method hook basics. Technical Report UCRL-PRES-211708, Lawrence Livermore National Laboratory, Livermore, California, April 2005. Presentation at the Common Component Architecture Forum's Spring 2005 meeting in Lincoln City, Oregon, USA.

[47] Tamara L. Dahlgren. Performance-driven interface contract enforcement for scientific components. In *Proceedings of the Tenth International Symposium on Component-Based Software Engineering (CBSE '07)*, Medford, MA USA, July 2007. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-228332, Livermore, CA, 2007.

[48] Tamara L. Dahlgren and Premkumar T. Devanbu. Adaptable assertion checking for scientific software components. In *Proceedings of the Workshop on Software Engineering for High Performance Computing System Applications*, pages 64–69, Edinburgh, Scotland, May 24, 2004. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-202898, Livermore, CA, 2004.

[49] Tamara L. Dahlgren and Premkumar T. Devanbu. Improving scientific software component quality through assertions. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 73–77, St. Louis, Missouri, May 2005. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-211000, Livermore, CA, 2005.

[50] Tamara L. Dahlgren and Premkumar T. Devanbu. Improving scientific software component quality through assertions. Technical Report UCRL-PRES-212172, Lawrence Livermore National Laboratory, Livermore, California, November 2005. Presentation at LLNL's Inaugural CAR Research and Technology Showcase.

[51] Tammy Dahlgren, Tom Epperly, Scott Kohn, and Gary Kumfert. Introducing design-by-contract to SIDL/Babel. Technical Report UCRL-PRES-150101, Lawrence Livermore National Laboratory, Livermore, California, October 2002. Presentation at the Common Component Architecture Forum's Fall 2002 meeting in Half Moon Bay, CA, USA.

[52] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. Babel/SIDL design by contract: Status. Technical Report UCRL-PRES-152674, Lawrence Livermore National Laboratory, Livermore, California, April 2003. Presentation at the Common Component Architecture Forum's Spring 2003 meeting in Salt Lake City, Utah, USA.

[53] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of the 21st Annual International Computer Software and Applications Conference*, pages 515–521, August 1997.

[54] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, volume 26, pages 246–255, September 2001.

[55] Paolo Donzelli, Marvin Zelkowitz, Victor Basili, Dan Allard, and Kenneth N. Meyer. Evaluating COTS component dependability in context. *IEEE Software*, 22(4):46–53, July 2005.

[56] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40:1–8, 1988.

[57] Paul F. Dubois. Scientific components are coming. *IEEE Computer*, 32(3):115–117, March 1999.

[58] Paul F. Dubois. Maintaining correctness in scientific programs. *IEEE Computing in Science and Engineering*, 7(3):80–85, May/June 2005.

[59] Stephen H. Edwards. 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.

[60] Stephen H. Edwards, Gulam Shakir, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings of the 5th International Conference on Software Reuse*, pages 46–55, Takamatsu, Japan, June 2–5, 1998.

[61] Sebastian Elbaum, Satya Kanduri, and Anneliese Amschler Andrews. Anomalies as precursors of field failures. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 108–118, November 2003.

[62] Sebastian Elbaum and John C. Munson. Investigating software failures with a software black box. In *Proceedings of the 2000 IEEE Aerospace Conference*, pages 547–566, March 2000.

[63] Albert Endres. An analysis of errors and their causes in system programs. In *Proceedings of the 1975 International Conference on Reliable Software*, pages 327–336, April 21–23, 1975.

[64] Albert Endres. Lessons learned in an industrial software lab. *IEEE Software*, 10(5):58–61, September 1993.

[65] Norman Fenton and Shari Lawrence Pfleeger. Can formal methods always deliver? *IEEE Computer*, 30(2):34, February 1997.

[66] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[67] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 23–32, Sydney, Australia, February 2002.

[68] S. Flake. Real-time constraints with the OCL. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, 2002.

[69] Mark Fleischer. Simulated annealing: Past, present, and future. In *Proceedings of the 1995 Winter Simulation Conference*, pages 155–166, Arlington, VA USA, December 1995.

[70] Franck Fleurey, Yves Le Traon, and Benoit Baudry. From testing to diagnosis: An automated approach. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE '04)*, pages 306–309, 2004.

[71] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposia in Applied Mathematics, Mathematical aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967.

[72] John Franco. The brick wall: NP-completeness. *IEEE Potentials*, 16(4):37–40, October/November 1997.

[73] Robert L. Glass. A sad SAC story about the state of the practice. *IEEE Software*, 22(4):120–119, July 2005.

[74] Robert L. Glass and Iris Vessey. Focusing on the application domain: Everyone agrees it's vital, but who's doing anything about it? In *Proceedings of the 31st Hawaii International Conference on System Sciences*, volume 3, pages 187–196, 1998.

[75] Carlos Gonzalez and Kian Tavakoli. A model for an adaptive scheduler. In *Proceedings of the 1988 ACM 16th Annual Conference on Computer Science*, pages 424–426, February 1988.

[76] GRUMMP — Generation and Refinement of Unstructured, Mixed-Element Meshes in Parallel. `http://tetra.mech.ubc.ca/GRUMMP/`. Visited 2005.

[77] Pedro Guerreiro. Another mediocre assertion mechanism for C++. In *Proceedings of the* 33^{rd} *International Conference on Technologies of Object-Oriented Languages (TOOLS 33)*, pages 226–237, June 2000.

[78] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[79] Babak Hamidzadeh, Lau Ying Kit, and David J. Lilja. Dynamic task scheduling using online optimization. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1151–1163, November 2000.

[80] Ali Hamie. Enhancing the object constraint language for more expressive specifications. In *Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC '99)*, pages 376–383, December 7–10, 1999.

[81] Ali Hamie, John Howse, and Stuart Kent. Interpreting the object constraint language. In *Proceedings of the 5th Asia-Pacific Software Engineering Conference (APSEC'98)*, pages 288–295, 1998.

[82] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Center for Advanced Studies on Collaborative Research*, pages 42–55, October 2004.

[83] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge, and Lianjiang Fu. SEAT: A usable trace analysis tool. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pages 157–160, May 2005.

[84] Warren Harrison. Skinner wasn't a software engineer. *IEEE Software*, 22(3):5–7, May 2005.

[85] Les Hatton. Reexamining the fault density—component size connection. *IEEE Software*, pages 89–97, March/April 1997.

[86] Les Hatton. Software failures: Follies and fallacies. *IEEE Review*, 43(2):49–52, March 1997.

[87] Les Hatton. The T experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, 4(2):27–38, April/June 1997.

[88] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797, October 1994.

[89] George 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.

[90] Raymond R. Hill. An analytical comparison of optimization problem generation methodologies. In *Proceedings of the 30th Conference on Winter Simulation*, pages 609–616, Washington, DC USA, December 1998.

[91] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[92] C. A. R. Hoare. Assertions: a personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, April–June 2003.

[93] Charles Antony Richard Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981.

[94] T. E. Hull, M. S. Cohen, J. T. M. Sawchuk, and D. B. Wortman. Exception handling in scientific computing. *ACM Transactions on Mathematical Software*, 14(3):201–217, September 1988.

[95] J. W. Hutchinson and P. G. Hindley. A preliminary study of large-scale software re-use. *Software Engineering Journal*, 3(5):208–212, September 1988.

[96] Interoperable technologies for advanced petascale simulations (ITAPS) center. `http://www.scidac.gov/math/ITAPS.html/`.

[97] Interoperable technologies for advanced petascale simulations (ITAPS) center. `http://www.tstt-scidac.org/`.

[98] Java^{TM}software — assertion facility. `http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html`, 2001.

[99] The Jass page. `http://csd.informatik.uni-oldenburg.de/~jass`, January 2003.

[100] jContractor. `http://jcontractor.sourceforge.net/`. Visited 2005.

[101] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[102] J. P. Kenny, C. L. Janssen, E. F. Valeev, and T. L. Windus. Components for integral evaluation in quantum chemistry. *Journal of Computational Chemistry*, July 2007.

[103] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.

[104] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, January 1996.

[105] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, January 1996.

[106] W. (Voytek) Kozaczynski and J. D. Ning. Concern-driven design for a specification language supporting component-based software engineering. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 150–154, 1996.

[107] Reto Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages (TOOLS 26)*, pages 295–307, August 3–7, 1998.

[108] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, May 1997.

[109] J. Larson, B. Norris, F. Bertrand, R. Bramley, D. Gannon, C. Rasmussen, T. Dahlgren, T. Epperly, G. Kumfert, D. Bernholdt, W. Elwasif, J. Kohl, R. Armstrong, B. Allan, S. Parker, K. Damevski, K. Chiu, and M. Govindaraju. CCA infrastructure and enabling technologies. Technical report, U. S. Department of Energy Office of Science, June 2005. Advanced Scientific Computing Research; Computer Science; FY 2005 Accomplishments flyer.

[110] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[111] Lawrence Livermore National Laboratory. Babel. `http://www.llnl.gov/CASC/components/babel.html`.

[112] Lawrence Livermore National Laboratory. Components technologies home page. `http://www.llnl.gov/CASC/components/`.

[113] Lawrence Livermore National Laboratory. Lawrence livermore national laboratory (LLNL) home page. `http://www.llnl.gov/`.

[114] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA 1998*, October 1998.

[115] Gary T. Leavens, K. Rustan, M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. Technical Report TR 00-15, Iowa State University, Ames, Iowa, August 2000. To appear in OOPSLA 2000.

[116] Paul Luo Li, Mary Shaw, Jim Herbsleb, Bonnie Ray, and P. Santhanam. Empirical evaluation of defect projection models for widely-deployed production software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations for Software Engineering*, pages 263–272, October 2004.

[117] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 141–154, San Diego, CA, June 9–11, 2003.

[118] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Sampling user executions for bug isolation. In *Proceedings of the 1st Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '03)*, pages 3–6, Portland, OR, May 2003.

[119] Ralph L. London. A view of program verification. In *Proceedings of the International Conference on Reliable Software*, pages 534–545, Los Angeles, CA, November 7–9, 1975.

[120] Kenneth C. Louden. *Programming Languages: Principles and Practice*. Brooks/Cole, Pacific Grove, CA 93950, 2003. Second Edition.

[121] David C. Luckham and Friedrich W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[122] Robyn R. Lutz and Ines Carmen Mikulski. Empirical analysis of safety-critical anomolies during operations. *IEEE Transactions on Software Engineering*, 30(3):172–180, March 2004.

[123] Wenhong Ma, James Yan, and Changcheng Huang. Adaptive sampling methods for network performance metrics measurement and evaluation in MPLS-based IP networks. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, volume 2, pages 1005–1008, May 4–7, 2003.

[124] Mike A. Marin. Effective use of assertions in C++. *ACM SIGPLAN Notices*, 31(11):28–32, November 1996.

[125] M. D. McIlroy. Mass produced software components. In *Proceedings of the NATO Software Engineering Conference*, pages 138–155, October 1968. Also available at `http://cm.bell-labs.com/cm/who/doug/components.txt`.

[126] Lois Curfman McInnes, Benjamin A. Allan, Robert Armstrong, Steven J. Benson, David E. Bernholdt, Tamara L. Dahlgren, Lori Freitag Diachin, Manojkumar Krishnan, James A. Kohl, J. Walter Larson, Sophia Lefantzi, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, and Shujia Zhou. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Parallel PDE-Based Simulations Using the Common Component Architecture, pages 327–381. Springer, 2006. Invited chapter; also available as Argone National Laboratory technical report ANL/MCS-P1179-0704 via `http://www.cms.anl.gov/cca/publications/p1179.pdf`.

[127] Lois Curfman McInnes, Jaideep Ray, Rob Armstrong, Tamara L. Dahlgren, Allen Malony, Boyana Norris, Sameer Shende, Joseph P. Kenny, and Johan Steensland. Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report preprint ANL/MCS-P1326-0206, Argonne National Laboratory, February 2006.

[128] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, 1997. Second Edition.

[129] Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 660–667, Portland, OR, May 3–10, 2003.

[130] Richard Mitchell, John Howse, and Ali Hamie. Contract-oriented specifications. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 24)*, pages 131–140, 1997.

[131] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 282–292, May 2004.

[132] Naoki Mori and Hajime Kita. Genetic algorithms for adaptation to dynamic environments — a survey. In *Proceedings of the 26th Annual Conference off the IEEE*

Industrial Electronics Society (IECON2000), volume 4, pages 2947–2952, October 22–28, 2000.

[133] John C. Munson and Sebastian Elbaum. Software reliability as a function of user execution patterns. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, page 12, January 1999.

[134] John C. Munson and Allen P. Nikora. Toward a quantifiable definition of software faults. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 388–395. IEEE Computer Society, November 2002.

[135] Nachiappan Nagappan. Toward a software testing and reliability early warning metric suite. In *Proceedings of the 26th International Conference on Software Engineering*, pages 60–62, May 2004.

[136] Object Management Group. CORBA basics. `http://www.omg.org/gettingstarted/corbafaq.htm`. Visited 2004.

[137] Object Management Group. Object management group (OMG) home page. `http://www.omg.org/`.

[138] Object Management Group. OMG IDL: Details. `http://www.omg.org/gettingstarted/omg_idl.htm`. Visited 2004.

[139] Object constraint language (OCL). `http://www./`.

[140] Carl Ollivier-Gooch, Kyle Chand, Tamara Dahlgren, Lori Freitag Diachin, Brian Fix, Jason Kraftcheck, Xiaolin Li, Eunyoung Seol, Mark Shephard, Timothy Tautges, and Harold Trease. The TSTT mesh interface. In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, January 2006.

[141] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–64, Rome, Italy, 2002.

[142] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96, Boston, Massachusetts, USA, 2004.

[143] Behrooz Parhami. From defects to failures: A view of dependable computing. *ACM SIGARCH Computer Architecture News*, 16(4):157–168, September 1988.

[144] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[145] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[146] Edgar M. Pass and John Gwynn. An adaptive microscheduler for a multiprogrammed computer system. In *Proceedings of the Annual Conference*, pages 327–331, August 1973.

[147] D. E. Perry and W. M. Evangelist. An empirical study of software interface faults. In *Proceedings of the International Symposium on New Directions in Computing*, pages 32–38, Trondheim, Norway, August 1985. IEEE Computer Society. Also available at `http://citeseer.ist.psu.edu/perry85empirical.html`.

[148] Dewayne E. Perry and Carol S. Stieg. Software faults in evolving a large, real-time system: A case study. In *Proceedings of the Fourth European Software Engineering Conference*, pages 48–67. Springer-Verlag, 1993. Also available at `http://citeseer.ist.psu.edu/perry93software.html`.

[149] Shari Lawrence Pfleeger. Soup or art? the role of evidential force in empirical software engineering. *IEEE Software*, 22(1):66–73, January/February 2005.

[150] Shari Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *IEEE Computer*, 30(2):33–43, February 1997.

[151] Amit A. Phadke and Edward B. Allen. Predicting risky modules in open-source software for high-performance computing. In *Proceedings of the* 2_{nd} *International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS '05)*, pages 60–64, May 2005.

[152] Reinhold Plösch. Design by contract for Python. In *Proceedings of the Asia Pacific Software Engineering Conference and International Computer Science Conference 1997 (APSEC '97 and ICSC '97)*, pages 213–219, December 1997.

[153] Reinhold Plösch. Evaluation of assertion support for the java programming language. *Journal of Object Technology*, 1(3):5–17, August 2002. Special issue: TOOLS USA 2002.

[154] Paula Prata and Joao Gabriel Silva. Algorithm based fault tolerance versus result-checking for matrix computations. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 4–11, June 15–18, 1999.

[155] David Reiner and Tad Pinkerton. A method for adaptive performance improvement of operating systems. In *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Methodology of Computer Systems*, pages 2–10, September 1981.

[156] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, volume 7, pages 65–78, July 1994.

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

[158] Johann Rost. Software engineering theory in practice. *IEEE Software*, 22(2):96–95, March 2005.

[159] S. H. Saib. Executable assertions — an aid to reliable software. In *Proceedings of the 11th Asilomar Conference on Circuits, Systems and Computers*, pages 277–281, November 7–9, 1977.

[160] Sriram Sankar and Roger Hayes. ADL — an interface definition language for specifying and testing software. *ACM SIGPLAN Notices, IDL Workshop*, 29(8):13–21, August 1994.

[161] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson Education, Inc., Boston, MA 02116, 2004. Sixth Edition.

[162] Mary Shaw. Truth vs. knowledge: The difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD-8)*, pages 181–185, March 1996.

[163] Forrest Shull, Vic Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Makael Lindvall, Ioana Rus Dan Port, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS '02)*, pages 249–258, June 2002.

[164] Mark E. M. Stewart and Scott Townsend. An experiment in automated, scientific-code semantic analysis. Technical Report AIAA-99-3276, American Institute of Aeronautics and Astronautics, Brook Park, Ohio, June 1999.

[165] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Upper Saddle River, NJ, special edition, 2000.

[166] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability — a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, June 25–27, 1991.

[167] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1999.

[168] Richard N. Taylor. Assertions in programming languages. *ACM SIGPLAN Notices*, 15(1):105–114, January 1980.

[169] Dave Thomas. Agile programming: Design to accomodate change. *IEEE Software*, 22(3):14–16, May 2005.

[170] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 73–81, March 1998.

[171] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 285–288, October 13–16 1998.

[172] N. Tran, C. Mingins, and D. Abramson. Design and implementation of assertions for the common language infrastructure. In *Proceedings of the IEE Software Engineering*, pages 329–336, October 27 2003.

[173] Terascale simulation tools and technologies (TSTT) center. `http://www.tstt-scidac.org/`. Visited 2005.

[174] Terascale simulation tools and technologies specification (version 0.5.1). `http://www.tstt-scidac.org/software/TSTT.sidl`. Visited 2004.

[175] Michael Turmon, Robert Granat, Daniel S. Katz, and John Z. Lou. Tests and tolerances for high-performance software-implemented fault detection. *IEEE Transactions on Computers*, 52(5):579–591, May 2003.

[176] United States Department of Energy. Center for Component Technology for Terascale Simulation Software (CCTTSS) Initiative. `http://www.scidac.gov/compsci/CCTTSS.html`.

[177] United States Department of Energy. Center for Technology for Advanced Scientific Component Software (TASCS) Initiative. `http://www.scidac.gov/compsci/TASCS.html`.

[178] United States Department of Energy. Scientific Discovery through Advanced Computering (SciDAC) Initiative homepage. `http://www.osti.gov/scidac/`.

[179] Michel Vasquez and Jin-Kao Hao. A hybrid approach for the 0-1 multidimensional knapsack problem. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, pages 328–333, 2001.

[180] Bart Verheecke and Ragnhild 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)*, pages 23–32, Sydney, Australia, February 2002.

[181] J. Voas. How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–119, 122, March–April 1997.

[182] Jeffrey Voas. Software quality's eight greatest myths. *IEEE Software*, 16(5):118–120, September/October 1999.

[183] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 73–84, Orlando, Florida, June 1994.

[184] Damien Watkins. Using interface definition languages to support *Path Expressions* and *Programming by Contract*. In *Proceedings of the Technology of Object-Oriented Languages (TOOLS-26)*, pages 308–317, August 1998.

[185] Damien Watkins and Dean Thompson. Adding semantics to interface definition languages. In *Proceedings of the 1998 Australian Software Engineering Conference*, pages 66–78, November 1998.

[186] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[187] Elaine J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

[188] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount`. Visited 2005.

[189] Wikipedia — Heuristic. `http://en.wikipedia.org/wiki/Heuristic`. Visited 2005.

[190] Peter Winkler. Optimality and greed in dynamic allocation. *Journal of Algorithms*, 41(2):244–261, November 2001.

[191] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 2000.

[192] Michal Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the 11th International Conference on Software Engineering*, pages 53–62, Pittsburgh, Pennsylvania, May 1989.