

Injecting Software Architectural Constraints into Legacy Scientific Applications

David Woollard^{1,2}

Chris Mattmann^{1,2}

Nenad Medvidovic²

¹Jet Propulsion Laboratory
4800 Oak Grove Drive, MS 171-264
Pasadena, California 91109, USA

Email: {woollard, mattmann}@jpl.nasa.gov

²Computer Science Department
University of Southern California
Los Angeles, California 90089, USA

Email: neno@usc.edu

Abstract

While software architectures have been shown to aid developers in maintenance, reuse, and evolution as well as many other software engineering tasks, there is little language-level support for these architectural concepts in legacy programming languages such as Fortran and C. Because many existing scientific codes are written in legacy programming languages, it is difficult to integrate them into architected software systems. By wrapping these scientific codes in architecturally-aware Java interfaces, we are able to componentize legacy programs, integrating them into systems built with first-class architectural elements while meeting the performance and throughput requirements of scientific codes.

1 Introduction

Scientific applications are complex, numerically intense codes which require significant new algorithm development and are often extremely long-lived due to this development difficulty. As a result, much of the scientific software in use today is written in Fortran and C, and these have remained the languages of choice for computational sciences.

One growing area of research emerging from the field of software engineering that has shown considerable worth in not only the construction, but also the maintenance and evolution of complex software is *software architectures* [8]. Software architecture functions as the blueprint of a software system, abstracting the system into constituent elements such as components (the computational units), connectors (the communications facilities between these components), and configurations (the organization of the components and connectors including their mapping to physical hosts).

Though design-level architectural constraints are greatly beneficial to software developers, the gap between these

concepts and implementation-level code requires the developer to maintain a complex mapping between architectural elements on one hand and language elements on the other. Reifying these architectural constructs as first-class elements of the implemented software system would directly aid the developer in a critical way: not only are design choices fundamentally codified, but there is a maintainable traceability between design and implementation which would eliminate the possibility of architectural drift and erosion [8].

It is exceedingly difficult to implement first-class architectural constructs in legacy languages such as Fortran and C due to lack of support for high-level concepts such as objects with internal state and separation of concerns. While these languages do not natively support such concepts, we propose that wrappers which provide architecturally-defined component interfaces can be used to componentize scientific codes. In turn, this will allow us to integrate them with architecturally-aware components to form complex software systems while still meeting the performance requirements of the scientific application.

In the rest of this paper, we will motivate the need for support of first-class architectural elements in legacy scientific code with an example of ongoing work at NASA's Jet Propulsion Laboratory (JPL). We will then discuss related work, introduce our wrapper technology and explore its computational impact on high performance scientific codes in both computation time and spacetime. Finally, we will conclude with a discussion of planned research in the area of software architectures for legacy scientific codes.

2 Motivation

At JPL, scientists process data from space-based instruments including Earth-monitoring satellites, exploratory probes, planetary (i.e., Mars) rovers, and so on as part of complex systems known collectively as ground data systems. These systems incorporate not only multiple legacy

scientific codes but also a broad set of functionality including data cataloging and archiving as well as workflow management.

In developing our next generation ground data system, JPL has initiated a project called The Division Architecture, or TDA, with the goals of achieving use of standardized interfaces, code reuse and extensibility, flexible deployment (i.e., fully distributed and running on multiple platforms), and rapid re-configurability via separable components.

While component-based software development might aid in re-configurability and reuse, it alone could not meet our goals of deployment flexibility or extensibility, nor provide a straightforward process for integrating legacy software. Likewise, rigorous development practices might aid in standardized interfaces, but it could not meet the needs of our ground data systems. Rather, we have developed a unified software architecture for ground data systems and adapted existing ground data system services to be used in compliance with this architecture.

While we have met with success in applying software architectures to other software built at JPL, such as the Planetary Data System [5], the project described in this paper is the first that incorporates JPL's more recently developed software services built in a componentized fashion with well-defined interfaces, as well as multiple scientific processing codes known as PGEs. These PGEs include a number of legacy radar packages (imaging and interferometry) and trajectory packages.

By fully integrating these scientific codes into our system architecture and enforcing TDA constraints, not only can we meet our goals for next generation ground data systems, but we can also provide system developers with guarantees of architectural properties and scientists guarantees about the performance of their code.

3 Related Work

Within the high performance realm, there have been a number of initiatives to integrate software engineering principles with high performance scientific development. One such initiative of note is the Common Component Architecture (CCA) [2]. Developed by a consortium of universities and national laboratories, CCA seeks to define a common component interface and framework that will allow scientific developers to reuse code that has been written to the standard. While component-based software development has been helpful for system developers, it suffers from certain pitfalls including architectural mismatch [3]. Additionally, performance problems have limited CCA's adoption among scientific programmers.

Software architects have developed a handful of techniques for integrating the high-level concepts of software architectures with underlying implementations. Architec-

tural description languages (ADLs) [6], for example, have been used to define, describe, and validate software architectures. These languages have focused primarily on model checking in terms of interface correctness and other system-level properties such as deadlock freedom. While ADLs occasionally provide a mechanism for code generation, they do not support integration with legacy code or traceability to implementation artifacts.

Two systems have tried to remedy this gap between architectural elements and implementation artifacts. ArchJava [1] is a programming language that provides the features of an ADL as a superset of Java. ArchJava thus requires a custom compiler, and limits the engineer to the natively supported set of architectural primitives.

Prism-MW[4] takes a different approach: it is an extensible, architecturally-aware middleware platform implemented in standard Java. Prism-MW uses a set of core classes to represent the basic architectural elements: *Architecture*, *Component*, *Connector*, and *Port*. It also provides an explicit extension mechanism (via Java's abstract classes and interfaces) to implement specific variants of these elements for use in different application scenarios and architectural styles [8, 7]. We use Prism-MW, and rely particularly on its extensibility mechanisms and native support for architectural styles, in our approach as discussed below.

4 Java Component Interface Wrappers

In this section, we will discuss the injection of architectural constraints in the form of Prism-MW wrappers applied to legacy Fortran scientific codes. We have accomplished this injection by instantiating legacy Fortran and C code through the Java Native Interface (JNI). Minor alterations of the legacy code are required in order to make it callable via JNI methods, though our experience has shown these modifications to be unobtrusive to the scientific algorithms, only requiring changes to the initialization and completion of the code.

One of JPL's current missions is the Orbiting Carbon Observatory (OCO), a earth-orbiting satellite launching in 2008 which will monitor CO₂ levels around the world and correlate onboard spectrometers with ground-based instruments before delivering data to scientists. As part of the ground data system for OCO, the Pre-flight Instrument Testing Process Pipeline (OpsPipeline) is a system consisting of a number of science codes processing data from instruments in a pipe-and-filter fashion, while also communicating with a Catalog and Archive Service (CAS). This system is illustrated in Figure 1. One goal of TDA has been to enforce architectural constraints on the components and connectors of the OpsPipeline.

In order to enforce these constraints, Prism-MW uses an *Architecture* class to record the configuration of its con-

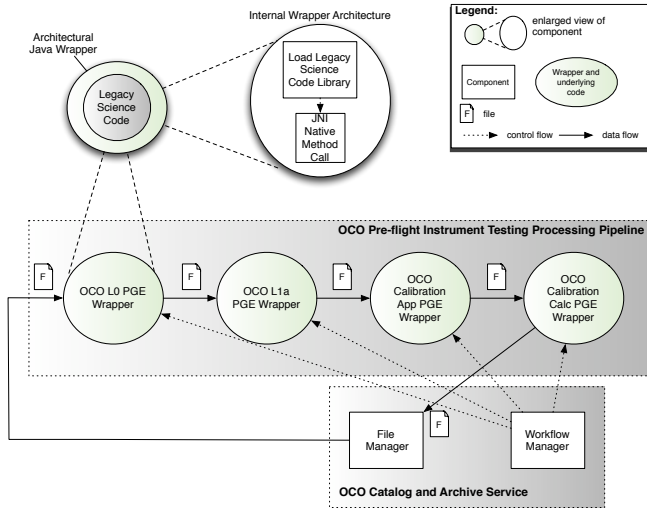


Figure 1. Instrument testing pipeline (OpsPipeline) for the Orbiting Carbon Observatory's ground data system.

stituent *Components*, *Connectors*, and *Ports*, and facilitate their addition, removal and reconfiguration. Distributed systems are implemented as sets of interactive architecture objects. Interaction between components in each architecture is facilitated by passing *Events* between *Components'* *Ports*. Each of these base classes has an *Extensible* counterpart to allow customization of Prism-MW to different application scenarios and architectural styles.

Prism-MW supports multiple styles by providing extensibility along five dimensions: structure, topology, behavior, interaction, and data. **Topological** and **structural** constraints are satisfied via implementations of the *AbstractTopology* class which is used by the *ExtensibleArchitecture* class. Topological constraints for TDA, such as the pipeline architecture of OpsPipeline in Figure 1, can be enforced via a child class of *AbstractTopology*.

To specify style **interactions**, *ExtensibleConnectors* use *AbstractHandlers* to implement event routing policies. For example, the unidirectional data forwarding used by connectors in the OpsPipeline (see Figure 1) can be developed as implementations of the *AbstractHandler* class. Style-specific **data** constraints are implemented via *ExtensibleEvents*.

The final dimension of architectural variance, **behavior**, is implemented via *ExtensibleComponent*. In order to componentize legacy scientific code, each Java wrapper is an extension of this *ExtensibleComponent* class, containing *Ports* in order to facilitate communications and registering style information with the system's *ExtensibleArchitecture*.

The resulting system enforces the pipe-and-filter archi-

tectural style, including the interactions between PGEs and the structure of the OpsPipeline through implementations of Prism-MW's abstract classes. The science algorithms implemented in legacy code remain untouched while the wrapper handles integration into the ground data system, including registration with the *ExtensibleArchitecture*, event handling and buffering, and other architectural constraints needed for the OpsPipeline to conform to TDA.

5 Performance and Wrapper Overhead

Scientific application developers are concerned not only with the accuracy of their code, but also with the code's performance. Large data sets and high precision require intense computation. In injecting architectural constraints into these legacy scientific codes, we have incurred a performance overhead which we must consider. The conventional wisdom in the scientific computation community suggests that the costs of abstraction and other "good" software engineering practices result in inefficient runtime performance and thus cannot be adopted for their applications. In this study of wrapper performance, we have disproved this mythos.

In order to measure the performance overhead of own Prism-MW wrappers on Fortran and C code, we used a canonical scientific software metric—the multiplication of two matrices. Starting with legacy base applications in both C and Fortran 77, we developed two types of Java wrappers: a "Thin Wrapper" which simply executes the legacy code without passing it any parameters or getting any in return, and a "Full Wrapper" which passes the legacy code matrices to be multiplied and gets the resulting matrix of the calculation when the legacy code returns. These two modes of interaction with the legacy code reflect our use of various PGEs on ground data systems.

Using Prism-MW, we implemented a filter interface from a pipe-and-filter architecture in each of the wrappers around legacy codes. The results obtained from our execution of the Fortran as well as our wrappers in both execution time and spacetime are given in Figure 2. These results closely match our performance results for C wrappers.

While the execution time overhead incurred by both the thin and the full wrapper is significant for shorter executions (approximately 40% execution time overhead for multiplying two matrices of 200×200 doubles), this overhead is insignificant for larger computations (the overhead for the multiplication of 1000×1000 doubles is under 2%). This decrease in overhead is a result of the amortization of the relatively constant cost of the Java Virtual Machine (JVM). In addition, we found that execution time is unaffected by the passing of data from the java wrapper to legacy code (i.e., the execution time difference between our thin wrapper and full wrapper is negligible).

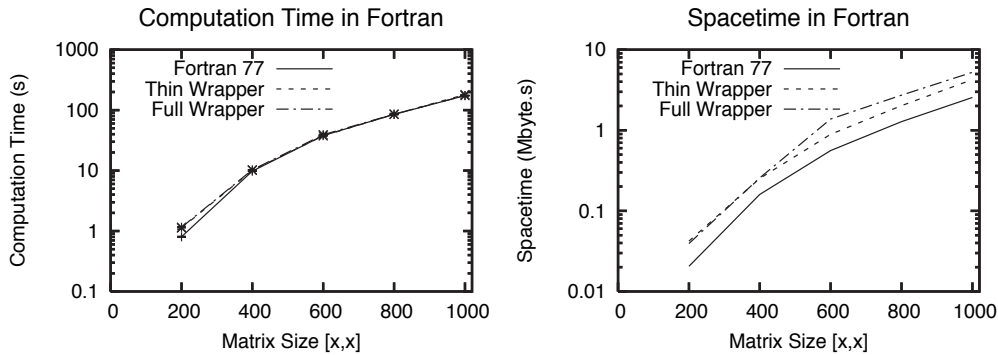


Figure 2. The performance impact of architectural constraint wrappers on Fortran codes.

In terms of memory performance, both java wrappers incur overhead due not only to the instantiation of the JVM, but also due to memory allocations in the Java portion of the wrapper. Since data must be passed through native methods by value, additional overhead is incurred by the full wrapper: the full wrapper occupies approximately double the memory of the original code due to its need to allocate matching space for parameters.

Existing literature suggests that a performance penalty of more than 10% execution time overhead in the high-performance scientific community is too great a price to pay for increased functionality [2]. We have shown that our architectural wrappers are well within this upper bound. Though our spacetime performance is more obtrusive, we feel that for many applications, including JPL's ground data systems, this penalty is acceptable.

6 Conclusion

Ground data systems are large, complex software used to preprocess data from space-based instruments and distribute it to scientists in multiple fields. These systems incorporate both data management functionality and high performance scientific codes often written in legacy programming languages. Injecting software architectural constraints into these legacy codes allows software developers at JPL to fully incorporate these legacy applications into architected ground data systems, allowing them to reap the benefits of software architecture-based development, including structured maintenance, reuse, and evolution.

While these interfaces meet the needs of software engineers, reifying software architectural elements as artifacts in code, they also meet the strict performance requirements of scientific code developers, incurring only minimal computational overhead.

In our future work, we propose to continue to improve the spacetime performance exhibited by our Java wrappers by exploring the use of per-compiled Java (via GCJ) and more aggressive garbage collection strategies. Additionally, we are working with a number of existing scientific codes which we hope to integrate into our next generation ground data systems including both radar and trajectory packages. Finally, we are looking to support parallelized scientific code via component replication and high performance software connectors.

References

- [1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *ICSE*, 2002.
- [2] B. A. Allen et al. A component architecture for high-performance scientific computing. *The International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.
- [3] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
- [4] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Trans. Software Eng.*, 31(3):256–272, 2005.
- [5] C. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *ICSE*, pages 721–730, 2006.
- [6] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.
- [7] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *First Workshop on Self-Healing Systems*, 2002.
- [8] D. E. Perry and A. L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, October, 1992.