

Kyle C. Miller

System Engineer, MISR Project (Multi-angle Imaging Spectro-Radiometer)
Science Software Systems Group (381)

Why is building a large science software system so painful? Weren't teams of software engineers supposed to make life easier for scientists? Does it sometimes feel as if it would be easier to write the million lines of code in Fortran 77 yourself? The cause of this dissatisfaction is that many of the needs of the science customer remain hidden in discussions with software engineers until after a system has already been built. In fact, many of the hidden needs of the science customer conflict with stated needs and are therefore very difficult to meet unless they are addressed from the outset in a system's architectural requirements. What's missing is the consideration of a small set of key software properties in initial agreements about the requirements, the design and the cost of the system. These key software properties are somewhat unique to the domain of scientific programming. Therefore, software engineers tend to overlook them until it is too late to include them efficiently. Early consideration of this set of science-critical software properties would promote better science and prevent the prolonged waste and frustration involved with a bad marriage between a science team and an expensive, long-lived software system.

After a brief introduction to a concept from the realm of software architecture known as "Reference Architectures," the set of ingredients which are critical to science software systems will be explained. These ingredients include properties of software such as design visibility, operational visibility, version traceability, planned requirements flexibility, long-term portability, reconfigurability, output simplicity, and user modifiability. Examples from various JPL EOS (Earth Observing System) software projects will be used to highlight the benefits of including these ingredients as well as the dangers encountered when they were overlooked. The presentation will conclude with the suggestion that such a set of key ingredients for science software should be polished and then adopted for use as a reference by scientists and software engineers who collaborate in the future.



Key Ingredients Needed When Building Large Data Processing Systems for Scientists

Kyle C. Miller

System Engineer, MISR Project

Earth Science Data Systems Sect. 381



Contributors From EOS Projects

(Earth Observing Systems)

- AIRS: Atmospheric Infrared Sounder
 - Steve Friedman, Evan Manning
- ASTER: Advanced Spaceborne Thermal Emission and Reflection Radiometer
 - Bjorn Eng
- MISR: Multi-Angle Imaging SpectroRadiometer
 - Mike Smyth
- TES: Tropospheric Emission Spectrometer
 - Susan Paradise, Douglas Shepard, Sirvard Akopyan



Large Science Software

- Challenges to a Science Team
 - Complexity: Millions of lines of code; many contributors.
 - Performance: Constant flood of satellite data through ambitious science algorithms on a limited budget.
 - Operational Robustness: Data Production done by External Organizations.
 - Access: Public promised instant access to validated data.
 - Longevity: Experiment evolves over decade(s).
- Pair Science Teams with Software Teams
 - Software Team engineers a hardware/software system.
 - Software Best Practices[1] are used to attack “challenges.”
 - The Science Team can expend more energy on science.



The Problem

- Best Software Practices, while critical to success, don't alone handle the challenges.
- Focusing on the obvious hurdles of complexity, performance and robustness in the initial design excludes features assumed from experiences with small science software.
- Resulting system can be too complex to understand, too costly to modify, too difficult to verify, too expensive to maintain, etc...
- Delays in data production and data quality improvement equate to wasted scientific opportunity.



Software Architecture

- The practice of good high-level software design.
- Rework to fix errors made at the high-level accounts for half the money spent on software development.[2]
- Reference Architecture: A Proven Template that can be Tailored to get a good design for a software system for a particular domain.[3]
- JPL first-of-kind experiments defy attempts to standardize software solutions.
- Yet, several architectural issues are common and are critical to scientist satisfaction. Reference Ingredients?



Key Architectural Ingredients

- Output Product Simplicity
- Version Trace-ability (in Output Products)
- Internal Operational Visibility
- Long-Term Design Clarity/Portability (vs. Optimization)
- Targeted Investments in Flexibility
- Design Visibility (at all levels)
- Reconfigure-ability & User Modifiability



Planned Flexibility & Output Simplicity

- AIRS: Table-Driven IO
 - Format of Products is specified in a single table.
 - Parameter Names, Data Types, Data Structures, and groupings are all defined in the table.
 - Underlying Read/Write Code is Generic, and is configured by this single table.
- Impact
 - Cheap, quick, accurate fmt. changes and additions.
 - Simplicity and Consistency of format is encouraged.



Planned Flexibility & Design Clarity

- **ASTER: Centralized Variable Naming**
 - Build command can run C Preprocessor on all components of the system.
 - Variable Names in C, Fortran, Perl, SQL and database can be modified with one action.
- **IMPACT:**
 - Names in all parts of the system are kept consistent with current usage and documentation.
 - Clarity increased for maintenance programmers and scientists.



Version Trace-ability

- **MISR: Version Annotation**
 - Product Filename Convention includes version #.
 - Software Executables are “branded” at build-time.
 - Software Version, Date, Computer Name, OS, Compiler flags, Configuration Management system parameters.
 - Software Writes Trace Info into each Product File.
 - Complete Executable Info + List of all Inputs.
- **Impact: Faith in Data Quality**
 - Users can correlate a file with a quality document.
 - Lengthy validation process can be iterative.
 - Data production mistakes are caught sooner.



Internal Operational Visibility

- **ASTER: Intermediate Temporary Files**
 - Production Software always dumps data to temporary files at each intermediate algorithm step.
 - Intermediate Files can be saved for the science team when requested.
- **Impact:**
 - The science team may easily perform low-level debugging/analysis without involving programmers.
 - Verification of production changes occurs sooner.



Portability vs. Optimization

- **ASTER: Parallel Database Population**
 - Frequent database population activity took days.
 - Clear, science-oriented design was left alone when the software team addressed performance problems.
 - Top-level design was modified to farm out pieces of the task to all available computers on the network.
- **Impact:**
 - Operations not hampered by slow database task.
 - Complex optimizations limiting future hardware options were avoided.



Reconfigure-ability & Portability

- **TES: Object Oriented Science Software**
 - C prototype of primary science retrieval algorithm given to Software Team.
 - Science and Software Teams collaborated to change top-level to C++ OO design.
 - Same code can be configured to run standalone for the scientists OR in production mode with trappings.
 - Production mode executable initiated by separate OO strategy system that handles parallel data processing plan on many small Linux systems.



Requirements Flexibility



- **MISR: Reuse of OO Software for AirMISR**
 - Cost-prohibitive software development effort to process Airborne validation experiment data skirted by elegant reuse of existing Object Oriented software components.
- **TES: Risky Algorithms attacked with OO**
 - Portions of the science code which were well-understood in the prototype were left in traditional structured, procedural C which scientists are comfortable with.
 - Ray Tracing Component identified as most risky. Special effort invested in developing with C++ OO model which will be easy to change.
 - Science Team appreciates OO after some initial education.

User Modifiability

- All Projects: Provide Sandbox Environment
 - Scientists can modify algorithmic software.
 - Scientists can run production software on real data.
 - Ease of use varies with clarity of software design.
- Impact:
 - Scientists can take over algorithm maintenance.
 - The high-level design evolves in anticipation of the requirements the science team will likely add in the future.



Common Conclusions



- Modest software development investments made to accommodate the unique needs of scientists seem to reap surprisingly large benefits.
- Costly mistakes often surround interfaces which were ill-defined or which weren't respected.
- Methods for Visualizing and communicating a system's Architecture (High-level) are dearly needed, especially in the realm of OO, which holds so much promise for science software otherwise.