P1.45 A PROTOTYPE FOR THE GIFTS INFORMATION PROCESSING SYSTEM

Raymond K. Garcia*, Steven A. Ackerman, Paolo Antonelli, Ralph G. Dedecker, Steven Dutcher, H. Ben Howell, Hung-Lung Huang, Robert O. Knuteson, Erik R. Olson, Henry E Revercomb, Maciej J. Smuga-Otto, and David Tobin.

CIMSS / SSEC / Univ. of Wisconsin, Madison

Abstract

The Geosynchronous Imaging Fourier Transform Spectrometer (GIFTS) instrument is expected to generate data at rates exceeding 1TB/day. The design of a proof-of-concept prototype ground data processing system is presented, which is responsible for turning raw interferograms from the instrument into calibrated, geolocated spectra in near-realtime. This prototype is intended for processing data generated during ground tests, as well as for testing algorithm variations, design alternatives, and implementation strategies. It embodies the core philosophy and architecture ideas from design studies carried out previously for a production-level GIFTS data system. These include a component approach which uses UML-RT to capture and communicate design decisions, a processing pipeline with pluggable algorithm stages, a flexible architecture implementable on a variety of platforms but targeted mainly for a distributed computing facility such as a cluster, and focus on the careful management and annotation of metadata with a view on complete reproducibility of processing and on fault tracing.

1. Introduction

The best way to write reusable software is to place oneself in the position of having to reuse the software in various contexts. The University of Wisconsin Space Science and Engineering Center (UW-SSEC), in collaboration with the National Oceanographic and Atmospheric Administration (NOAA), is working to implement software architecture for the data processing needs of current and future imaging Fourier transform spectrometers, using object-oriented techniques and real-time toolsets. The motivating application for the software system is the validation of the Geostationary Imaging Fourier-Transform Spectrometer (GIFTS) instrument. Other applications of the software suite being developed include processing systems for experimental Fourier Transform Spectrometer (FTS) sounders, such as the Atmospheric Emitted Radiance Interferometer (AERI) (Knuteson, 2004a), the Scanning High-resolution Interferometer Sounder (Scanning HIS), and NASA's NOAA Polar Orbiting Environmental Satellite Suite (NPOESS) Airborne Science Testbed Interferometer (NAST-I) (Revercomb et al., 1998). Many aspects of the software address risk reduction needs for a planned FTS-based operational atmospheric sounder being proposed for the GOES-R Hyperspectral Environmental Suite (HES). This paper presents an overview of the prototype implementation of the GIFTS Information Processing System (GIPS), focusing on science data processing components implemented in C++, and covers lessons learned thus far over the course of the development effort.

2. Background

The processing of future imaging FTS data to extract products including absolute radiance spectra, vertical profiles of temperature, water vapor and trace gases, and wind fields is as much a problem in computer science as it is in atmospheric remote sensing. Thus, the design of a processing system must take into account not only the physics of radiative transfer, radiometric calibration and cloud clearing, but also the computing mass of the data and need to assure proper provisioning and tracking of metadata. While this paper is mostly concerned with the computer science aspects of the problem, it starts with an overview of requirements stemming from the atmospheric science.

2.1 *Science Background*

In order to extract accurate absolute radiances from an atmospheric infrared FTS, a detailed model of the instrument's internal "background radiation" must be maintained. This is built from characterization experiments and theoretical models, and refined with periodic radiance measurements from a set of self-calibration blackbody sources. Such calibration data is interleaved with nominal earth observations. The resultant model is used to estimate time-matched reference radiances that form an absolute basis of comparison (Revercomb et al. 1988). Given the resultant absolute radiance measurements, an extensive radiative transfer coefficient library is consulted using a combination of statistical and physical retrieval algorithms. This permits the extraction of vertical atmospheric profiles for temperature, water vapor, and other atmospheric

constituents (Huang 2001). Such measurements, in turn, can be mosaiced together as accurate and contiguous geolocated image planes, which are used for wind vector extraction using feature tracking (Velden et al. 2004).

2.2 *Existing instruments and their processing pipelines*

Historically, developing software for research interferometers is a process of perpetual prototyping. There is always a better - or at least a different - way of treating the data that promises improved accuracy, further noise reduction, or faster rendering of products. Two decades of experience with the AERI, HIS, Scanning HIS and other FTS instruments show that there is always need for improvement in processing, visualizing and distributing derived products.

Dealing with the variety of experimental configurations of any one of these research instruments requires backwards compatibility and often presents the software team with ad-hoc changes to processing requirements. Largely this has been handled by lowering barriers on software development and testing. This involves reliance on simplified "flat" file formats, preference for scripting languages such as Python, and packaging the processing stages as separate executables which can be strung together in short order, creating a complete processing pipeline on a host computer or cluster. Such methods must be expanded upon or even rethought in order to handle future FTS applications effectively.

Testing algorithms is expensive. Regression testing and noise characterization is time-consuming but necessary; debugging handshakes between successive stages of a data-processing pipeline can be tedious and often requires that processing code include logging or monitoring capacity. When dealing with multiple similar instruments, it is substantially more cost-effective in the long term to take proven algorithm implementations, and insert adapters for the respective instrument platforms, in order to recoup the cost of verifying the software implementing the science algorithms.

Algorithms for these instruments have the potential for stateful behavior, and thus implementing them as simple context-free library functions shifts the burden of ensuring correct deployment on the surrounding code. For instance, for the Scanning HIS instrument, calibration source observations are taken every 15-30 seconds. When operating in a batch-processing mode, a windowed fit model is generated from the calibration radiance views over several minutes' time in order to process all data within the window. Reprocessing requires recovery of the local calibration window – the context surrounding a given observation. For a real-time calibration requirement, this window can only be retrospective. Identifying the best strategy for folding the calibration data with the observation data to extract calibrated radiances is as much an algorithm choice as the radiometric calibration equation itself.

A further complication for aircraft-based experimental interferometers is that the instrument itself evolves over time as understanding improves, limitations are characterized and overcome, and additional sensor data are injected into the instrument data stream. The way to meet the needs of this kind of mutable hardware system is with equally mutable software architecture.

2.3 *Requirements on the GIFTS information processing system (GIPS)*

A real-time processing system must not only execute such science data processing pipelines correctly and efficiently, but also orchestrate the use and update policies of large databases containing reference and model data that the science depends upon. It must manipulate decision-support metadata and reference data evolving at several time scales: some reference structures change on the order of seconds, others on the order of hours, still others on the order of months. It must also record auditing information in order to maintain confidence in the truth-value and traceability of the products, which are expected to rapidly step into the petabyte scale.

The current approach must address real-time constraints, dialogue with databases of varying implementation, efficient exchange of data between components, testing and validation, flexibility of deployment, and audit data recording for software and products.

2.4 *Use Scenarios*

There are several scenarios under which processing software must perform. These include:

A. Satellite real-time downlink: Emphasis for this case is on keeping up with the stream of incoming data, estimated at 1.5 Terabytes per day for a GIFTS-type instrument. It is also necessary to maintain acceptable latency and accuracy of products, in order to make them promptly available for downstream weather prediction and nowcasting applications.

B. In-flight: For experimental setups mounted on unmanned airborne vehicles (UAV), spectral resolution is traded away in order to operate with limited CPU and memory resources, limited power consumption, and

limited weight allotment for computing hardware. This is similar to the constraints imposed on satellite on-board processing systems. The databases and calculations must be "trimmed to fit" and optimized to provide real-time feedback that can be immediately used for in-flight decision support. Plans for operational networks of UAV's provide another application of this mode of use (Dovis et al. 2001).

C. Fieldwork: For post-flight processing of data taken from S-HIS type instruments on field experiments, a portable hardware system must be capable of processing and quality checking the recorded data at high precision and rapid turnaround time. This scenario in particular requires the software to be maximally flexible, so that a field technician can assemble ad-hoc algorithms and procedures into a meaningful and repeatable processing pipeline.

D. Bulk reprocessing of data and sensitivity studies: Large amounts of clustering hardware or simple compute farms can be brought to bear on the problem of large-scale reprocessing of old datasets with new reference data or algorithms, and ensemble testing with input permutations used for sensitivity analysis or algorithm tuning. Managing the quantities of data involved takes precedence over performance concerns in this case.

E. Algorithm research and development cycles: Regression testing, unit and integration testing of new algorithms requires that the software interact readily with visualization environments and debuggers, and provide rapid feedback for test pattern data sets. This scenario takes place on individual scientists' and engineers' workstations and laptops prior to releasing candidate software for production use; for a full-scale data processing system, the development environment is likely to include multiprocessing systems requiring distributed visualization capability.

The goal of GIPS is to provide for all the above uses, through the application of component-centered design philosophy tempered with an appreciation of the requirements imposed by each of these scenarios.

3. Designing with components

In our previous papers on this topic (Garcia, Smuga-Otto 2004a, 2004b, 2004c), we presented increasingly detailed top-down design studies of a system able to address the aforementioned requirements and concerns. Discussions so far included the concept of a reference database service, audit controls, monitoring subsystem, and included an interaction pattern outline for the algorithm software modules within a real-time data processing framework. For this discussion, we focus on a bottom-up realization of componentized algorithm software by way of useful and sustainable software design practices.

3.1 *Towards a component-based architecture*

Previous efforts at applying a component design to the legacy processing software largely consisted of using files for intermediate data storage and simple network protocols (e.g. UDP broadcast of metadata) to isolate and monitor processing stages in the pipeline for individual testing. This approach has been adequate for fieldwork and reprocessing of datasets from instruments like S-HIS and NAST-I; it lends itself less well to ensemble testing and development work, and is not readily adaptable to real-time processing. The evolutionary nature of much of the codebase that took this approach has complicated any efforts at generalization and extension of the codebase.

Another experimental approach to data flow systems used ideas similar to those found in the Visualization Toolkit (Schroeder et al. 1996) with implicit flow of control and highly constrained data generalizations connecting processing components. Like the "filesystem-connected" stages, the distribution of top-level control logic among the components led to difficulty in attaining proper flow of control for algorithms, which have richer interactions with their surroundings than the largely stateless algorithms that dominate VTK. Further work on this approach was made difficult by premature generalizations that did not fit the problem domain.

These early experiments led to the observation that the flow of data through the system is as much a "science algorithm" as the numerical manipulations performed on the data. Separating the concerns of marshalling data, operating on data, and directing the sequencing and movement of the data within the system would lead to a higher degree of reuse. With this in mind, parts of the S-HIS data system were rebuilt as operator modules applicable to a variety of use patterns as outlined above. Heavy reliance on design patterns including the Builder pattern (Gamma et al. 1995) allowed for great flexibility in invoking such operators on a stream of packetized data.

3.2 *UML-RT*

A significant break-through in architecting the system came with the assimilation of the object-oriented

concepts of the Unified Modeling Language – Real Time extensions (UML-RT), whose abstractions are a good fit for the concerns listed here. While the need for component-oriented approaches was noticed early on, the particular set of abstractions offered by UML-RT proved to provide a reasonable path to code that did not compromise any of the fundamental requirements.

The UML-RT view of components is centered on Capsules, Ports, and Protocols. Capsules are constructs for isolating functionality with a very clearly defined interface: Each capsule operates according to a state diagram, responding to and generating signals through its ports. The signal content on each port is prescribed by its role in a protocol, which itself can be represented with a state transition graph (Douglass 1999, Selic 2003).

UML-RT came about as an adaptation of object-oriented design philosophy to the world of real-time software, which is shaped by practicalities of mixed hardware-software systems with very tight time and resource budgets. Separability of three major concerns is inherent in capsule-based designs - processing itself, marshalling of data, and flow of control. Control is described by a state transition table appropriate to the role of the capsule. Data moving through the system is represented by the protocol descriptions for the connected ports - these are the patterns of exchange between the components of the system.

A core aspect of the UML-RT methodology is that it encourages the recursive description of an entire system. A top-level system design is divided into functional sub-capsules and their interaction protocols; these sub-capsules in turn hide smaller capsules further dividing and conquering the system requirements. This approach allows rigorous characterization and testing of resource use of the system capsule-by-capsule, and helps guarantee that the system achieves strict performance requirements.

4. The GIPS prototype

A system of "black boxes" prescribed by capsule responsibilities, protocol requirements and state charts can be represented directly in software design, and communicated using the UML-RT syntax.

4.1 *Design philosophy*

For our implementation of the UML-RT inspired design, we chose ANSI C++ together with STL, POSIX and other common open-source libraries, including ATLAS, MIT FFTW, and the Blitz++ matrix library. Since our heritage software, such as the current S-HIS processing system is written in a mixture of C, Fortran77, C++, Python, Java, and MATLAB, some backward compatibility is required in order to allow verification of the new system to take place.

The practical concerns of doing object-oriented design in C++ have shaped some of the coding conventions we have followed. C++ lacks in its language specification features such as garbage collection and pointer safety, leaving them instead to libraries such as Boost and the C++ Standard Template Library (STL). These libraries function in part as language extensions by way of C++'s strong support of compile-time generics and operator overloading. Despite its complexity, C++ was deemed to be the most mature language platform at the present time in which to develop the ideas of GIPS.

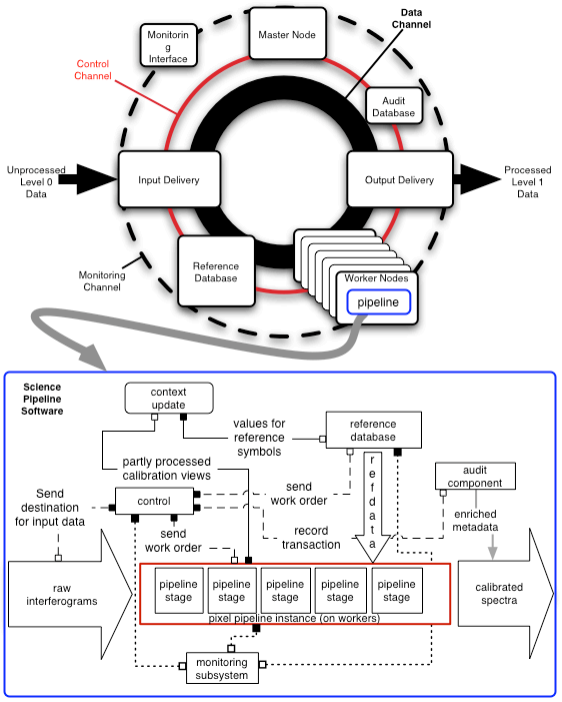

Figure 1: Diagrammatic overview of the GIFTS design, with internal details of the science pipeline exposed as a UML-RT style capsule diagram.

4.2 *Addressing design concerns*

Three critical aspects of the system that are addressed by our science software prototype are reuse, performance, and testing.

A. Reuse: Algorithm capsules, once tested, should be able to fit into a variety of enclosing frameworks without need for re-coding. Exchanges of data are represented as fully virtual protocol-role interfaces, which can be implemented using a variety of connector classes. Protocols should be kept

simple within the science processing, and defer flow-of-control decisions to the enfolding capsule's control logic where it simplifies inter-capsule protocols. Protocol descriptions should above all avoid imposing implementation details on the capsules.

B. Performance: The high-volume observation data for each detector pixel should be minimally copied. The enclosing capsule is given ownership of the exchange buffers between sibling sub-capsules. Recognize and expect that protocols will need to be revisited in order to do optimization refactorings. Favor simple data carriers (e.g. structs) over elaborate delivery containers with costly serialization requirements where the option exists.

C. Testing (unit and integration): Algorithmic functional units must be readily verified using test fittings and unit tests. Each capsule has a unit test verifying its operating state transitions.

Furthermore, the following concerns and requirements are accommodated if not fully implemented in the prototype code:

D. Auditing: Auditing is not yet directly answered in code by these design principles. Critical aspects of auditing will be managed by proper generation and use of some implementation of Universally Unique Identifiers (UUIDs). For an example implementation, see Leach et al. (2004). Such UUIDs must be registered for each distinct algorithm version within the versioning system, for each deployed software configuration item. They must also track which tool set - compilers, library versions, code generation tools, and algorithm version corresponded to the configuration item. UUIDs also need to be tracked within the data processing system to identify processing jobs, reference data, and collected as UUID tuples representing the processing history for released products.

E. Monitoring: Monitoring is allocated for by providing the capsules an iostream-compatible port following the standard output convention for C++ to accept text, values, and shorthand tags which can be used to generate monitoring logs and displays.

F. Refactoring of and testing against heritage systems: Existing applications systems and libraries can be encapsulated once their outermost protocols are extracted. Refactoring passes can then extricate control logic and data marshalling in order to improve performance and meet reuse needs. Where expedient during prototyping, heritage algorithms from other languages (e.g. Fortran77) are recoded in C++.

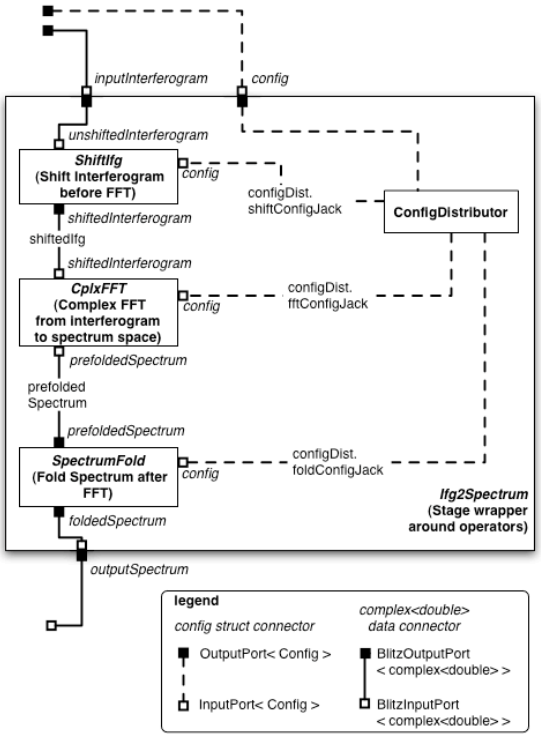

Figure 2: UML-RT capsule diagram of the interferogram-to-spectrum stage of the L0-L1 pipeline.

4.3 *The system as a network of capsules*

Our adaptation of UML-RT concepts in C++ implements connectors and capsules as objects, and protocol roles as interfaces. Connectors are meant to be provided by the system integrators and implementers, and used by capsules without knowledge of their internal construction. Capsules hold the science and other supporting algorithms and services required of the GIPS. Since capsules can be composed into larger enclosing capsules, the entire data processing system may be thus constructed by applying the same principle recursively.

With the GIPS science processing pipeline itself a capsule, a full-scale data processing system can be created so that instances of the pipeline are distributed across dozens or hundreds of processor nodes. This takes advantage of “embarrassingly parallel” nature of many of the processing tasks. Each pipeline capsule is embedded within a processing

node is provided access to observation data and reference data. The worker capsules themselves are sibling capsules to the main data processing system input and output, auditing, monitoring and reference data service capsules. The controlling logic of the overall system distributes processing jobs to the workers. In the case of a large shared memory system, building connectors moving data to the pipeline largely takes the form of implementing access to memory buffers. In the more complex case of a distributed multiprocessor target, a system of middleware (e.g. MPI, ACE, CORBA) connectors and capsules will be required to hide the movement of data structures and control signals across machine boundaries from the science pipeline.

```
// Configuration Structure for the Interferogram to Spectrum stage
struct Config_t
{
        size_t npts; // number of points in input interferogram (and output spectrum)
        int foldPoint; // point around which transformed spectrum is folded. See
                       //  SpecFoldOperator.
        Config_t( size_t n, int f ): npts( n ), foldPoint( f ) { }; ///< constructor
};

// Convenience typedefs
typedef  Connector::BlitzInputPort< std::complex< double > > JackIn;
typedef  Connector::BlitzOutputPort< std::complex< double> > JackOut;
typedef  Connector::InputPort< Config_t > JackConfig;

// Ports for the Interferogram to Spectrum stage
struct Ports
{
        JackIn &inputInterferogram; // raw interferogram input reference
        JackOut &outputSpectrum; // raw spectrum output reference
        JackConfig &config; // configuration settings reference

        Ports( JackIn &, JackOut &, JackConfig & ); // constructor
};

// The stage capsule as a class
class Ifg2SpectrumStage: Operator::AbstractOperator
{
protected:
        Ports &ports;
        explicit Ifg2SpectrumStage( Ports & ); // constructor takes reference to Ports
public:
        virtual Operator::Result operator()( ); // implementation of AbstractOperator
        virtual ~Ifg2SpectrumStage(); // destructor

        // The factory method, which can produce different Ifg2SpectrumStage instances
        // depending on what value of the Implementation enum is passed in
        static Ifg2SpectrumStage *Factory( Instrument::Implementation, Ports & );
};
```

Figure 3: C++ interface code from the file Ifg2spectrum.h

4.4 *Capsules as science algorithms*

By the strictest UML-RT definition, capsules are expected to accept signals on their ports asynchronously, effectively functioning as parallel execution threads. While this level of generality is appropriate for multithreaded and multiprocessor systems, our task of implementing a science processing pipeline within a worker capsule, we establish a practical convention, which we've assigned the name of Operator.

The Operator has an activation function that operates synchronously; in C++ this function is operator(), and returns a success or failure to complete a calculation cycle on its input.

The vectors and other data structures made available on an operator's input ports remain constant while it is active. Thread safety of the data held on the Operator's ports is the responsibility of the enclosing capsule, and may eventually be handled by thread-aware versions of connectors.

Where possible, the state of the Operator is externalized as context input and update ports. This allows an instance of a data processing pipeline to be re-used at runtime for a variety of similar jobs (e.g. reprocessing) by recovering archived context data prior to re-activating the operator.

In this fashion, the processing pipeline becomes a series of processing stage operators, which are activated in sequence by the pipeline control logic. The pipeline itself is signaled to proceed when the enclosing worker has sufficient data ready for a job to be processed.

4.5 *Developing the system in C++*

When coding a science algorithm as a major data processing stage, one starts by identifying the dependencies and command set of the algorithms within that stage's Operator. These will largely take the form of data structure declarations and specific content definitions to be delivered to the algorithms by way of one of its input ports.

```
// SHISStage is a concrete implementation of Ifg2SpectrumStage, for Scanning-HIS.
class SHISStage: public Ifg2SpectrumStage
{
private:
    // create distributor for configuration connector (external to internal)
    ConfigDistributor configDist;

    // create intermediate buffers as connectors
    Connector::BlitzSimpleConnector< std::complex<double> > shiftedIfg;
    Connector::BlitzSimpleConnector< std::complex<double> > prefoldedSpectrum;

    // and set up the operator classes using these connectors
    IfgShift::Ports shiftPorts;
    IfgShift::IfgShiftOperator ifgShift;

    CplxFFT::Ports fftPorts;
    CplxFFT::CplxFFTOperator cplxFFT;

    SpecFold::Ports foldPorts;
    SpecFold::SpecFoldOperator foldSpectrum;
public:
    explicit SHISStage( Ports &ports ): Ifg2SpectrumStage( ports ),
        configDist( ports.config ), // initialize config port distributor
        // create internal connectors (memory buffers) of correct size
        shiftedIfg(configDist.inConfig.npts), prefoldedSpectrum(configDist.inConfig.npts),
        // initialize contained capsules, first initializing their ports
        shiftPorts( ports.inputInterferogram, shiftedIfg, configDist.shiftConfigJack ),
        ifgShift( shiftPorts),
        fftPorts( shiftedIfg, prefoldedSpectrum, configDist.fftConfigJack ),
        cplxFFT( fftPorts ),
        foldPorts( prefoldedSpectrum, ports.outputSpectrum, configDist.foldConfigJack ),
        foldSpectrum( foldPorts )
    {};  // constructor - most of the wiring happens here

    virtual ~SHISStage() {}; // destructor
    virtual Operator::Result operator() ( ); // interface contract of AbstractOperator
}; // class Stage

// implementation of AbstractOperator's operator() interface
Operator::Result SHISStage::operator() ( )
{
    Operator::Result res = 0;
    // the internal capsules and connectors between them are already set up.
    // Just call them in order, and collect return codes.
    res = ifgShift();
    if (res == Operator::OK)
        res = cplxFFT();
    if (res == Operator::OK)
        res = foldSpectrum();

    return res;
} // operator()
```

Figure 4: Implementation in C++ of the abstract interface for the interferogram-to-spectrum capsule, using three internal capsules (ifgShift, cplxFFT and foldSpectrum) joined by internally defined connectors (shiftedIfg, prefoldedSpectrum). Taken from SHISStage.cc

The dependencies are then gathered into protocol roles (example is an input role for a data port) required by the Operator. The protocol roles are declared as C++ virtual abstract classes, which function as interface specifications in the C++ world. Local protocol roles are based on existing protocol role archetypes where appropriate – this lessens complexity when building distributed applications. The capsule's protocol roles are finally bundled into a Ports structure that is used to initialize operator instances (see figure 3: C++ interface code, for a simple example). The names and protocol roles for the ports can be directly represented in a UML-RT capsule diagram of the Operator (as shown in figure 2).

Unit tests are built for the operator using simple connectors that pass in the prepared input test structs and vectors. The operator is implemented such that upon activation, it cycles through a computation, recovering inputs from its input ports, writing output and monitoring data to the corresponding output ports, and then returns a pass/fail code to the upstream controlling logic (see Figure 4 for implementation code corresponding to the interface listed in figure 3).

4.6 *Connectors as data conduits*

In UML-RT component diagrams, connectors are represented as lines connecting capsule ports; each connector endpoint answers a particular protocol role, and the connector conforms to its declared protocol. Protocols are usually binary – Connectors that implement them have two ends, one acting in an input role, the other as an output.

It can be a challenge to design protocol roles for a given algorithm balancing the number and type of ports without forcing implementation details on the capsule. If too much unrelated data is grouped on one port, it may need to be separated internally to route relevant pieces to individual sub-components. At the same time, minimally revealing the implementation details of the encapsulated algorithms permits more alternatives in implementing the Operator. Thus, consistent practices for classifying the inputs (metadata, engineering data, observation data, reference data) are needed in order to provide clear and concise interfaces.

C++, with its support for both class inheritance and generic programming, provides a broad set of possibilities with which to construct connectors. The approach chosen for GIPS comes from a desire for strong type safety and compile-time error detection. The philosophy behind the design is to make it easy to use capsules and connectors properly, and catch obvious mistakes at compile time. An equally important goal is to decouple the programming of capsules, which only need to know the protocols they interact with, from the programming of connectors and glue code. Capsule implementers do not import connector headers or any other implementation details - they strictly specify the abstract protocol roles, which must be satisfied at the time of the capsule's instantiation. Separately, the implementer of the enclosing system creates for each capsule a set of connector instances compatible with the protocols. In order to speed development, a connector template library answering many of the archetype protocol roles is being assembled. It currently includes utility connectors such as raw data carriers, implementing a simple binary data in/ data out protocol, as well as connectors which use pre-existing test patterns, connectors with debugging hooks, and connectors using previously allocated memory as their data sources or sinks.

Most important for the efficient functioning of the system is a set of connectors for managing the high-volume hyperspectral data. For the current prototype effort, connectors utilizing the Blitz++ library were built, owing to its literate treatment of large multi-dimensional arrays and management of references to slices of memory. For these connectors, the input port interface specifies a getForInput() method that returns a one-dimensional immutable (C++ const) array containing the input vector of interest to the downstream capsule. Likewise, the output protocol role specifies a getForOutput() method for use by the upstream capsule. These vectors provide access to memory already allocated by the connector, and do not perform a copy when their getForInput/Output methods are called. The difference between the various kinds of connectors fulfilling this raw data protocol lies in how the backing memory is provided. The BlitzSimpleConnector allocates a vector in memory of a specific size, and allows proper upstream and downstream access. The BlitzMatrixBackedInputConnector allows stepped access to one row at a time of an entire two-dimensional blitz array.

Example uses for the matrix-backed connector are dealing with a large number of records provided simultaneously, and double buffering. If an entire set of records becomes available at once in memory (e.g. via a bulk network transfer or memory mapped file) this kind of connector may be used to loop over the entire set of records as the controlling logic invokes the relevant Operator instances for each record. Alternately, if space for only two records is allocated in the backing memory, the controlling code may be queuing data from a network connection even as the other record is being used as an input vector for an enclosed computational capsule. Bridges between language systems, such as those allowing C++ capsules to be fed data from Java arrays, or allowing

direct visualization and interaction using MATLAB or Python, can be largely constrained to be connector adaptations.

Other connector types include struct connectors, lookup connectors and monitoring connectors. Struct connectors can be used to deliver immutable algorithm configuration data to capsules, such as input spectrum size that needs to be known at capsule instantiation time. Lookup connectors fulfill the need of science algorithms to access a database of instrument characterization and self-calibration records in order to properly extract science from the interferograms. These connectors implement protocol roles designed for the purpose of retrieving and augmenting database entries. Whether the backing implementation behind the connectors is a full SQL database server, or an STL map based out of local memory - as might be the case for an embedded in-flight application or a unit test - the implementation remains opaque to the science algorithm client. Monitoring connectors currently work off the assumption that emanated messages vary from capsule to capsule, and thus provide a streamed output channel. In C++, standard iostream conventions are adopted.

4.7 *Capsules as science algorithms.*

The code in Figures 3 and 4 shows a sample algorithm interface and implementation. The intention behind abstracting the algorithms as capsules is to capture not only the specific sequence of math functions required, but also the context necessary for that sequence of functions to make sense. In part, bundling algorithms into objects also enables some aggressive optimization techniques, such as caching FFT plans assembled during object construction. Three different approaches to providing an implementation to a science capsule have been carried by the authors: atomic (function-based C/C++ code) implementations, sub-capsule implementations, and wrapping of legacy Fortran code. When an existing C/C++ library already provided some basic functionality, as in the case of the complex FFT operator, the library call was simply wrapped in a capsule. This presented a good opportunity to gather the legacy code for setting up intermediate buffers and creating an FFT plan which had previously been scattered around the source files into the capsule as well, thus freeing the new capsule-style surrounding code from such implementation details. A similar approach was taken with algorithms to fold a complex spectrum coming out of the FFT and to shift a complex interferogram going into that FFT. Once these were encapsulated, it was a fairly easy to gather them together into a higher-level capsule, by first setting up intermediate data buffer connectors, making sure that an external configuration connector was correctly distributed, and then simply firing the operator() calls for the three component capsules in sequence. Another capsule, which performs a nonlinearity correction for the SHIS instrument, was implemented by wrapping an existing well-characterized heritage Fortran binary inside a capsule. This last coding experiment demonstrated the philosophy of enabling rapid development as well as side-by-side testing of C++ versus Fortran implementations by utilizing known code in the new framework.

4.8 *Other practical considerations*

In a production system, memory leaks are not permitted. Allocation and freeing of memory blocks can incur unpredictable overhead at run-time. These kinds of behavior are strictly disallowed in hard real-time systems, and need to be characterized in soft real-time systems. The convention we select is that the enclosing capsule is the owner of all connectors and capsules that it operates. It instantiates internal connectors and delivers references for those connectors to its internal capsules; the capsule implementer may safely assume that the objects being referenced will be valid over the lifetime of a capsule instance. We favor instance variables with constructor-passed references over the use of new() and pointer passing. We also prefer libraries such as STL and Blitz++ to control heap allocation of data variables. This all should improve the stability of memory usage once a network of capsules is constructed and configured. Finally, while C++ exceptions are a welcome language feature, they should not be able to leak through capsule boundaries after they have been constructed – as this would create a mechanism separate from the well-defined ports by which capsules may be coupled to the external world.

5. Conclusion and continuing work

This paper presents an overview of practical challenges and solutions encountered so far in our C++ prototype implementation of the GIFTS Information Processing System. This is an ongoing process, and the current focus is on refining the prototype, with the goal of making it fully operational for both S-HIS data processing needs in the field, and for upcoming GIFTS thermal vacuum tests.

Source code, sample datasets and updates for this project can be found at:

http://www.ssec.wisc.edu/gifts/noaa/l0l1.html

Acknowledgments

The authors wish to acknowledge NOAA for supporting this work through Grant #NA07EC0676.

Bibliography

Alexandrescu, A., 2001: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, New York

Davis, S. P. Abrams, M. C., Brault, J. W. 2001: *Fourier Transform Spectroscopy*, Academic Press, San Diego

Douglass, B. P., 1999: *Real-Time UML: Developing Efficient Objects for Embedded Systems,* (*2nd Edition*), Addison-Wesley, New York

Dovis, F., L. L. Presti, E. Magli, P. Mulassano, G. Olmo, 2001: Stratospheric platforms: a novel technological support for Earth observation and remote sensing applications, *Proceedings of SPIE -- Volume* **4540***, Sensors, Systems and Next-Generation Satellites* V, Hiroyuki Fujisada, Joan B. Lurie, Konradin Weber Editors, pp 402-411

Gamma, E., R. Helm, R. Johnson, J. Vlissides 1995: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, New York

Garcia, R. K. and M. J. Smuga-Otto 2004a: "Design Studies for the GIFTS Information Processing System", *20th International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, American Meteorological Society, Boston, MA.

—, 2004b: Distributed Computing for the Extraction of Meteorological Products from the GIFTS Imaging Interferometer, presented at the third GOES-R Users Conference, Broomfield CO.

—, 2004c: Component-oriented design studies for efficient processing of hyperspectral infrared imager data. in *Atmospheric and Environmental Remote Sensing Data Processing and Utilization: an End-to-End System Perspective* edited by Hung-Lung Allen Huang, Hal J. Bloom, Proceedings of SPIE Vol. **5548** (SPIE, Bellingham, WA, 2004) pp 444-454

Huang, H.-L., P. Antonelli, 2001: Application of Principal Component Analysis to High-Resolution Infrared Measurement Compression and Retrieval. *Journal of Applied Meteorology*: Vol. **40**, No. 3, pp. 365–388

Knuteson, R. O., F. A. Best, N. C. Ciganovich, R. G. Dedecker, T. P. Dirkx, S. Ellington, W. F. Feltz, R. K. Garcia, R. A. Herbsleb, H. B. Howell, H. E. Revercomb, W. L. Smith, J. F. Short, 2004a: Atmospheric Emitted Radiance Interferometer (AERI): Part I: Instrument Design, *J. Atmos. Oceanic Technol.*, (Accepted for Publication)

Knuteson, R. O., F. A. Best, R. Dedecker, R. K. Garcia, S. Limaye, E. Olson, H. Revercomb and D. Tobin, 2004b: Level 0-1 Algorithm Description for the Geosynchronous Imaging Fourier Transform Spectrometer, *20th International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology,* American Meteorological Society, Boston, MA

Leach, P., M. Mealling, R. Salz, 2004: A UUID URN Namespace. http:// www.ietf.org/internet-drafts/draft-mealling-uuid-urn-03.txt

Revercomb, H. E., H. Buijs, H. B. Howell, D. D. LaPorte, W. L. Smith, L. A. Sromovsky, 1988: Radiometric calibration of IR Fourier transform spectrometers: solution to a problem with the High-Resolution Interferometer Sounder, *Applied Optics* Vol **27** No. 15, pp 3210-3218

Revercomb, H.E., D. C. Tobin, V.P. Walden, J. Anderson, F.A. Best, N.C. Ciganovich, R.G. Dedecker, T. Dirkx, S.C. Ellington, R.K. Garcia, R. Herbsleb, H.B. Howell, R.O. Knuteson, D. LaPorte, D. McRae, M. Werner, 1998: Recent Results from Two New Aircraft-based Instruments: the Scanning High-resolution Interferometer Sounder (S-HIS) and the NPOESS Atmospheric Sounder Testbed Interferometer (NAST-I), *Proceedings of the Eighth International Workshop on Atmospheric Science from Space Using Fourier Transform Spectrometry (ASSFTS8),* Toulouse, France, sponsored by Meteo-France, CNES, CNRS, pp 249-254

Schroeder, Will, with Ken Martin and Bill Lorenson, 1996: *The Visualization Toolkit*. Prentice-Hall, Boston, MA

Selic, B., J. Rumbaugh, 2003: Using UML for Modeling Complex Real-Time Systems, Rational / IBM whitepaper, http://www-128.ibm.com/developerworks/rational/library/139.html

Velden, Christopher, Dengel, Gail, Huang, Allen Hung-Lung, Stettner, David, Revercomb, Hank, and Knuteson, Robert, 2004: Determination of wind vectors by tracking features on sequential moisture analyses derived from hyperspectral IR satellite soundings. *20th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, American Meteorological Society, Boston, MA