

InSAR Scientific Computing Environment

Eric Gurrola^{*}, Gian Franco Sacco^{*}, Paul A. Rosen^{*}

Howard Zebker⁺

^{*}Jet Propulsion Laboratory
4800 Oak Grove Dr
Pasadena, CA 91109 USA

⁺Stanford University
Mitchell Bldg.
Stanford University
Stanford, CA 94305

Abstract—We are developing a new computing environment for geodetic image processing for InSAR sensors to enable scientists to reduce measurements directly from radar satellites and aircraft to new geophysical products without first requiring them to develop detailed expertise in radar processing. The environment can serve as the core of a centralized processing center to bring Level-0 raw radar data up to Level-3 data products, but is adaptable to alternative processing approaches for science users interested in new and different ways to exploit mission data. The NRC Decadal Survey-recommended DESDynI mission [1] will deliver data of unprecedented quantity and quality, making possible global-scale studies in climate research, natural hazards, and Earth's ecosystem. The InSAR Scientific Computing Environment, applied to a global data set such as from DESDynI, is expected to enable a new class of analyses that take greater advantage of the long time and large spatial scales of these new data, than current approaches [2].

We are implementing an accurate, extensible, and modular processing system and reworking the processing approach in order to i) enable multi-scene analysis by adding new algorithms, ii) permit user-reconfigurable operation and extensibility, and iii) capitalize on codes already developed by NASA and the science community. The framework incorporates modern programming methods, including rigorous componentization of processing codes, abstraction and generalization of data models, and a robust, intuitive user interface with graduated exposure to the levels of sophistication, allowing novices to apply it readily for common tasks and experienced users to mine data with great facility and flexibility. The framework is designed to easily allow user contributions, creating an open source community that will extend the framework into the indefinite future.

I. INTRODUCTION

The objectives of the InSAR Scientific Computing Environment are to develop an open-source, modular, extensible InSAR computing environment for the research community. The environment is to incorporate state-of-the-art, highly accurate algorithms to automate InSAR processing for non-experts and experts alike. To service the community and promote use and update of the code, the project will deliver documented algorithms, formats and interfaces. The specific goal is to create a code suite that the InSAR community embraces and grows with.

The approach our multi-institutional team has taken is relatively straightforward for a software development project,

but perhaps more structured than a typical research code development. We are approaching the development by applying modern software system engineering techniques and tools for configuration management and maintenance. In this work we are first collecting community-based requirements for InSAR processing methods and generalized data models. We then use these requirements to define an object-oriented framework. With the framework in place, we populate it with processing modules. Along the way we create documentation of the framework, modules, and use cases.

II. THE ISCE ARCHITECTURE

Architectural Goals

The ISCE architecture implements a computing environment that can process interferometric synthetic aperture radar (InSAR) data from all current spaceborne platforms as well as the planned DESDynI platform, and that is accurate, easy to use, flexible, and easily extensible by both experts and non-experts. From these basic principles, we have derived the following key drivers of our architecture:

1. Preserve the vast expertise and testing currently encoded in Legacy Software
2. Make that Legacy Software more lean in terms of the number of auxiliary tasks it needs to do (such as self configuration and I/O configuration).
3. Build modern object oriented structures around and behind the legacy code to manage that code and push rather than pull user configuration onto that code before executing that code
4. Implement common functions and services such as I/O through APIs to allow their implementations to change and to allow for user configuration and selection of those functions at run time
5. Build in polymorphism mechanisms to allow user selections to alter the implementations of major processing steps and common functions. Also allows just-in-time insertion of alternative functions and major components

At the core of the ISCE architecture are two legacy InSAR processing packages: ROI_PAC and STD_PROC. Both of these software packages are primarily written in Fortran, mostly using the Fortran77 version of the language with some of the transitional features leading up to Fortran90 such as structures and dynamic memory allocation. Some of the programs are written in C. Control scripts are written in Perl (ROI_PAC) or Python (STD_PROC). ROI_PAC was initially developed over a decade ago and has been used extensively by the science community to process InSAR data from several different international spaceborne radar platforms such as ERS, EnviSAT, JERS, ALOS, and TerraSAR-X. STD_PROC is currently being developed at Stanford University and is based on advances in the processing algorithms that came from processors developed at Jet Propulsion Laboratory (JPL) for SRTM and UAVSAR [3,4].

Although STD_PROC is new, we refer to it along with ROI_PAC as legacy code because they are both received as domain expert software whose functionality we wish to preserve and also because they are both built in a well-known style that is very effective at accomplishing the processing steps but not easy for non-experts to use and not very flexible or extensible for expert developers to work with. The ISCE architecture seeks to inject some modern software principles that allow for easier use and greater flexibility and extensibility.

Architectural Framework

To accomplish these goals of ease of use and greater flexibility and extensibility, the ISCE architecture surrounds the code housed in the programs and scripts of the legacy software with a city of structures that deliver services to the legacy programs as well as to the user and developer. The services delivered to the legacy code do not replace the major processing tasks of the legacy programs; rather, they replace interactions with the external world that the legacy programs handled using mostly primitive language features. The structures that deliver the services to the legacy programs replace structures currently housed inside the legacy programs, which requires modifications to the legacy programs to remove those structures and to add new wiring or plumbing to receive those services from the external structures.

It is as if the legacy program were a house that contained the power plant that converts coal into electricity for its own use. Coal must be delivered to the house because that is what it requires to produce its own electricity but the appliances in the house only require the electricity at the outlets and it doesn't matter whether that electricity is produced externally or internally and whether it is produced from energy from burning coal or from nuclear or solar energy; the electricity is the same either way.

In the same way the legacy programs require data from the external world delivered in a certain format but they are also required to pull that data from the external world, which requires them to know the formats and locations of that data in the external world. This handling of interactions with the external world, such as obtaining user command parameters and data, requires the programs to know more about the external world than they need to know, or rather imposes a fixed model of the external world, which causes inflexibility and brittleness. The only necessary part of this interaction is that the legacy program needs data at a certain point in its own internal format---it only requires an outlet at a certain point with electricity of the right voltage. This is all that should be necessary from the given program's point of view.

The responsibility for obtaining (or putting) the data from (to) the external world, including understanding data formats and conditioning data for delivery in the proper format at the proper point in the program, can be delegated to external structures in the forms of software libraries and objects and through given interfaces (the wires and pipes) to those structures. Using modern object-oriented programming

structures and patterns these services can be dynamically configured according to the conditions of the external world for the given processing run, while still being able to be attached to the same wiring and plumbing and still deliver data to the programs in the form they require and without affecting the real work of the programs in processing the data.

The restructuring of one legacy program unit into several program units is illustrated in Figure 1. It brings several possibilities for improvement of the overall software package. First, and most obvious, is that the design is more modular with a greater division of labor and responsibilities into separate specialized modules. The module representing the modified legacy program now contains only code for processing the data and hence it is simpler and easier to maintain and does not need to be changed or even recompiled whenever external data formats change as they often do. Second, the external modules that handle the user interface and the pipes can be generalized and reused for all of the processing programs in the legacy software. The legacy programs as received all contain very similar copies of the same tens to hundreds of lines of code to handle the tasks that are now handled by the external modules. Third, the modules that have been externalized from the legacy programs, as well as the wrapper around each legacy program, can now be freed from the coding style and language of the legacy code and can become specialized and built using object-oriented design principles and design patterns to further enable ease of use, flexibility, and extensibility.

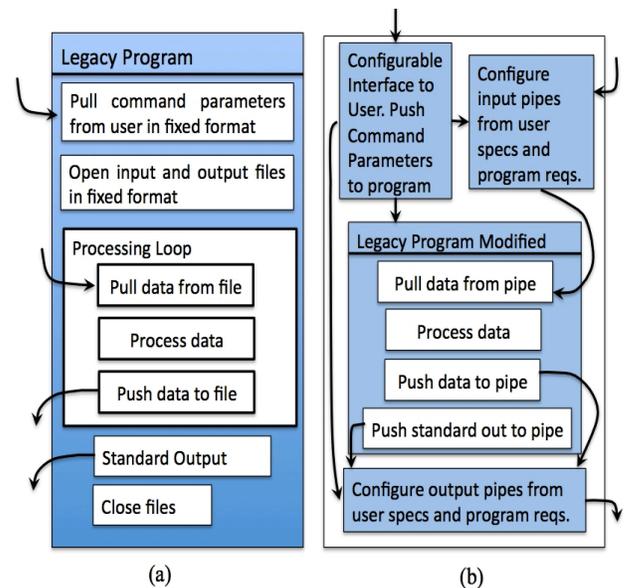


Figure 1. Schematic representation of a legacy program for data flow. (a) The flow in a legacy program before restructuring for ISCE. (b) The modified flow after restructuring for ISCE

A key step in restructuring the software is to *componentize* the legacy programs. To componentize a legacy program is to embed it in a software wrapper that satisfies the properties that a component is expected to have. There is no standard definition of these properties but we include among them a software entity that: (1) has independent integrity in the sense that it is not dependent on any particular implementation of other components with which it might interact; (2) has the ability to interact with other components in an interchangeable fashion; (3) defines “contractual” interfaces for control parameters, inputs and outputs which might include parameters, objects, and data streams; (4) contains proper initialization and finalization methods; and (5) provides introspection capabilities for its public methods and attributes.

Figure 2 shows the architecture of a component that has an embedded legacy core. A legacy core is not required for this to qualify as a component; ISCE includes components that do not contain a legacy core, but they do include the properties of a component and include other methods or class functions that contain the main function of the component. The figure shows framework components and properties upon which the component is built either through class inheritance or composition. The difference between class inheritance and composition is that the component that inherits properties and methods from a framework class *is* an object of the type of the inherited class with additional properties and methods built on top of it, whereas the component that acquires properties and methods by composition from a framework class *has* those properties and classes encapsulated in an object that it contains of the type defined by the framework class.

Figure 2 shows flow of configuration and control parameters from the top into the component initialization method. Those configuration and control parameters flow down to the component from a controlling or driving application, which is a special type of component. The configuration and control parameters are derived from user inputs, either from the command line or from input files, and defaults defined in preferences files and possibly also defined within the application itself. The component itself may also define defaults for parameters. Defaults can always be overridden by user inputs.

Runtime Polymorphism

A key feature of ISCE that is meant to satisfy the requirements for flexibility and extensibility is built-in runtime *polymorphism*, a software mechanism to alter the behavior of the software at runtime through user inputs, without requiring the software to be recompiled. Through object-oriented principles, *interfaces* and *tasks* can be defined in the software components and applications, while deferring the instantiation of the concrete software objects that implement the *tasks* and adhere to the *interfaces* until run-time, when user inputs can be used in deciding which objects are appropriate or preferred for the given task for a particular processing run. Furthermore, when done carefully, it is not even necessary for all of the concrete software objects to exist at the time that the interfaces and tasks are defined in the computing environment.

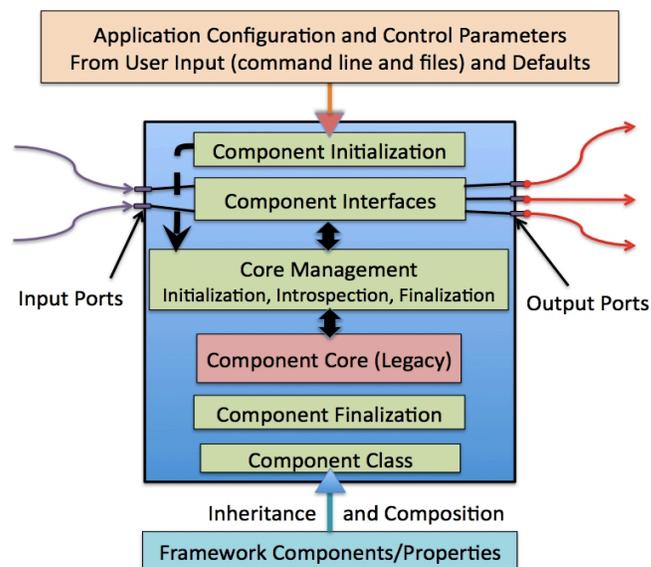


Figure 2. Architecture of a component

An ISCE developer or a user will be able to add software objects to the framework at any time after the framework is built. User inputs or default settings can select the built-in or contributed objects at any time as long as the objects adhere to the interfaces.

We are allowing for two types of polymorphism: (1) what we refer to as *facility* polymorphism where major components may be morphed at run-time; and (2) a *plug-in* type of polymorphism where lower level, common functions such as implementations of fast Fourier transforms (FFTs) may be selected across the board at run-time. Facilities define a task and an interface that are implemented by a component. Registering a Component as a Facility indicates the Component as the default Component to implement the Facility but also alerts the Application to allow the User to specify an alternate Component for that Facility at runtime.

Provenance

One advantage of the modularity and object orientation of ISCE is that we can develop objects with the dedicated task of tracing and logging the *provenance* of every data file produced by ISCE. Provenance is the ability to log and query the pedigree of a particular piece of processed data, which is an important element of scientific repeatability for the community. Provenance allows users to keep track of the versions of applications, components, and other software that were used to produce a data product, the configuration parameters used to initialize those applications and components, as well as the provenance of the input data and other output data products. Provenance will allow an investigator to explore data processing strategies, using different versions of the software or perhaps iteratively tweaking parameters while keeping a record of what was tried at every point. This fosters reproducibility of results and allows users to create a record of what was done to the data

that can be shared with the community in the form of publications or scripts, which is an important aspect of scientific discovery and refinement. ISCE supports provenance through database management and logging of processing steps and meta-data along each step of the processing chain. Given the Python-based object-oriented methods in ISCE, the code lends itself to being used within software packages with higher levels of sophistication that provide provenance capability as well. For example, several GUI interfaces, such as VizTrails, have complete provenance management, and easily accept Python applications and plug-ins as GUI-based modules. This effectively extends the ISCE utility as a scientific tool with very little effort.

III. DESCRIPTION OF THE ISCE SOFTWARE

To this point, we have been discussing the architecture in abstract terms, but now we turn to describing the software in more detail. In Figure 2, everything except the Component Core is programmed in Python, and the Component Core consists of the native language legacy code, which may be Fortran or C. To interact with the Python elements, we use the standard Python application programmer's interface (API) for *binding* C to Python, with an intermediate layer to bind Fortran to C when the legacy code is Fortran.

A more concrete picture of what the user sees in the ISCE distribution is shown in Figure 3. The user downloads ISCE from the JPL SVN server into a source directory. SVN is the acronym for Subversion, which is an open source Version Control System (VCS) that is meant to be an improved version to the familiar CVS system, with many of the same commands. After downloading ISCE, the user can view the source code, run a simple SVN command to update a copy of the source code anytime that it changes in the JPL SVN

server, and build and install the software for local use.

To build the software, the user runs SCons, a Python based build system that works well in building code written in multiple languages. SCons uses *SConstruct* files, supplied in each directory, that tell SCons to build and install the software in directories specified by the user through a user-specified global configuration file. The user is then ready to process radar data downloaded from data servers such as those maintained by the agencies that manage the radar platforms. Certain users will be given privileges to use SVN to contribute new software and to fix any bugs that might be found in the ISCE software so that the ISCE can continually be improved and enhanced by the user community.

Figure 3 shows the structure of the source directory that is downloaded. The mainline ISCE applications and components are contained under the Applications and Packages directories, where Packages are collections of logically related Components, Legacy Cores, and other support software. The Packages are *iscesys*, which contains the ISCE system or framework components and properties as well as several APIs; *isceobj*, which contains class definitions for several objects used by the components; *mroi pac*, which contains the ROI_PAC recasting into components; and *stdproc*, which contains the STD_PROC recasting into components. Figure 3 shows a branch called Contrib, which points to a separate directory where the user's software, or the contributed software, is housed.

A component by itself does not actually do anything. It must be instantiated in another type of component called an *application* that has the responsibility of collecting the user inputs and of managing its components from their initialization to the flow of data through them to their finalization. Figure 4 shows the architecture of an application.

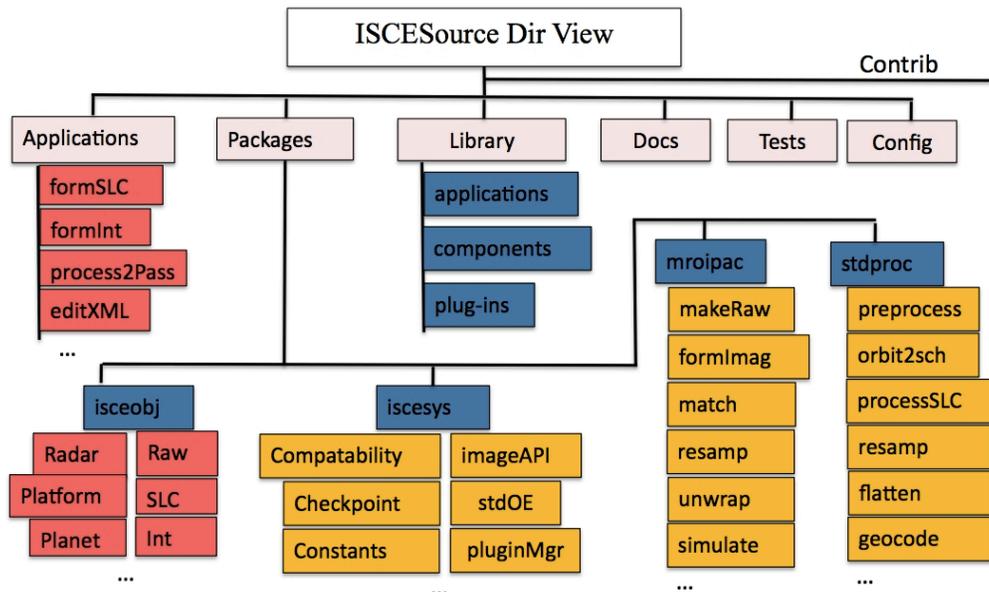


Figure 3. The structure of the ISCE source directory.

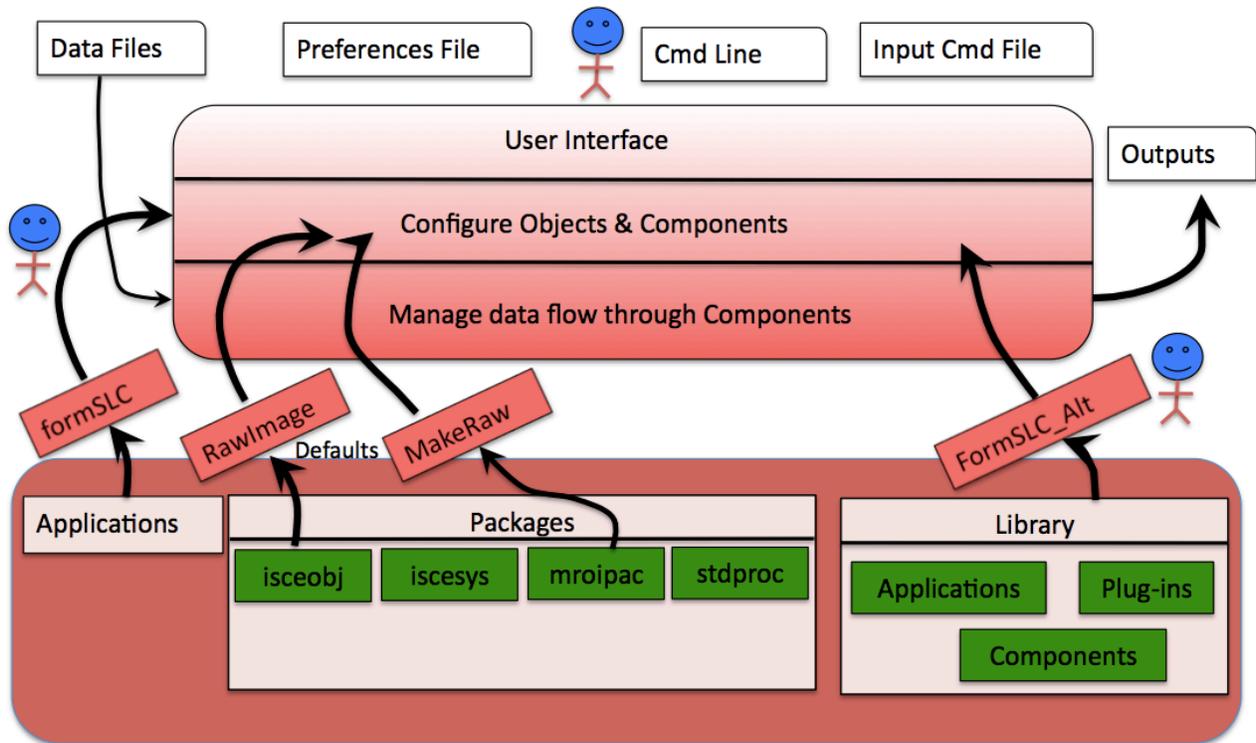


Figure 4. Application architecture. The blue people indicate points where the user selects input parameters through input files and the command line as well as the components that are instantiated for a particular run.

IV. DESCRIPTION OF THE API SOFTWARE

In this first year of development, we have recast approximately half of the ROI_PAC modules into cores of Python Components as described in the previous section. We have also created the framework elements that support the ISCE architecture. In particular, we have coded, documented, and tested the key framework APIs that allow us to control processing flow among ISCE modules. These are the Image, Control, and StdOE APIs. We now describe these software elements.

Image API

The image API provides a set of library functions that provide the legacy software and new programs developed by users with a reliable and versatile way of performing input and output operations on images. The image API consists of a set of C++ classes that contain an abstraction of a real world image, concrete methods to access data from sources (such as, but not limited to, files on disc), and a memory buffer to hold a given portion of an image that can be passed between the C++ and Fortran programs. The C++ classes allow for very general and flexible configuration of the objects instantiated from them without specific regard for the types of images and

memory buffer specifications currently in the Fortran programs of ROI_PAC and STD_PROC.

We have developed the Image API with a number of features to meet our goals of extensibility and flexibility. One of the key features ensuring extensibility is the use of an object-oriented language: In the future versions of this software, the concrete sources of data may evolve to new types not anticipated at this time. Through object-oriented class inheritance mechanisms, new data accessor methods can be layered on top of those currently available without requiring us to rewrite code that currently works.

We have built-in flexibility by providing a variety of methods for accessing data from the sources such as sequential access to full lines of data, random access by line number, as well as single pixel access. The legacy code specifies which way it needs to access the data and uses library functions that call the appropriate class methods that have been configured when they were created to conform to the properties of the abstract image (such as its width and height) and the data sources.

We have also allowed for flexibility in the number of supported band interleaving schemes. Radar images in the legacy code most often consist of complex numerical values at each pixel organized in range lines and cross-range position.

The legacy code's in-memory representation of the data often treats the pixels as a single band of complex numbers but sometimes represents the pixels as two bands of real numbers (either magnitude and phase or real and imaginary) and uses different band interleaving schemes, such as interleaving by pixel, interleaving by line, or band sequential interleaving. Future code may be written with unanticipated concepts of bands and we have written the Image API to be very general without regard to what is currently found in the legacy code. The Image API allows users to control the conversion from one representation to another on input and output in an efficient manner to allow maximum user flexibility.

The Image API allows machine dependent internal representations of the numerical values to be converted on the fly so that data files created on one machine can be used in our software without first creating a new file conforming to the internal representation of binary data on the machine that is running the ISCE software. One particular internal representation issue, for example, that often causes difficulties is the endianness, or the ordering of bytes representing a numerical value from least significant to most significant byte or the opposite.

The Image API exploits I/O caching, improving efficiency and speed. The legacy code usually operates on one range line of data at a time in a sequential fashion, which is not necessarily the most efficient method of accessing data from a file. Cached I/O allows the data accessor to be optimized to load larger amounts of data at a time and feed the data to the legacy code in the chunks that it requires. The data accessor is responsible for determining when file I/O operations are performed without the legacy code being involved in that process. While modern disk controllers often perform their own caching to minimize disk usage, they don't know how image data are typically addressed and utilized. The Image API implements caching locally around a selectable collection of image lines (order of 100 lines), such that local operations can occur without reliance on the controller cache. We have found this to greatly improve our throughput for certain image manipulations.

Control API

The Control API consists of a set of classes, features and methodologies that the ISCE framework utilizes to guarantee an easy, correct, reproducible, extensible and reconfigurable way of passing data among the different computing modules.

Generally speaking, modules contain parameters or attributes that need to be set appropriately before they can perform their function properly. The control API provides methods for setting and examining these attributes through *set* and *get* methods.

All ISCE modules inherit from a ComponentInit base class. This class, as the name suggests, allows the initialization of the parameters of the subclass that inherits it by passing an initializer object to it. The ISCE framework provides a set of default initializers that permit initialization from file, from a dictionary (an object consisting of a set of (key,value) pairs) or from another object. Expert users could provide their own

initializers, as long as they conform to the architecture specifications.

The ComponentInit class provides a set of convenience methods to allow the user to explore how to use a component, to debug his usage of the component, and to document the state of the control parameters through the following built-in capabilities: (1) determine which variables in the module must be set by the controlling program (i.e., those parameters that have no valid default value); (2) determine which variables have default values and what those default values are so that the user may have the option to override the default values; and (3) render the state of the component to a configurable destination such as to a file or to standard output (i.e., dump the variables and associated documentation of an object to a specified destination that may be used and stored by the user to debug a component or to document the provenance information on the component's state).

Another component of the Control API is the Checkpoint class, which provides a check-pointing capability to the ISCE framework. In general the process of check-pointing allows the user to save the state of the system at a given point during the program flow, such that the program can be resumed at a given checkpoint without having to recompute the previous stages. This feature is important in the event that the process was interrupted.

StdOE API

An essential element of any program suite is the ability to print informative messages about the status of the processing (e.g. percent completion, derived parameters that may be of interest to the user, etc.) and any error messages that occur. In conventional programming, particularly, in Fortran, coders insert write statements into their code that are sometimes compiler dependent, such that when compilers change, all the code needs to be updated. To avoid such tight and brittle connection of logging messages to the code elements, we have created a logging API we call StdOE for "Standard Output and Error." The StdOE consists of a C++ class used for reconfigurable standard output and error. With this API, it is possible to choose destinations for standard output and standard error messages. For example, the user might choose to select the terminal window for both of these output streams or might decide to send them to separate files, to a network socket, or to the input of some other process.

CONCLUSION

There will be a torrent of radar data available to the research community in the coming years as more and more spaceborne international radar systems are launched and the data become available for use, either in real time or from historical archives. With the upcoming launch of the European Sentinel-1 spacecraft and the US DESDynI spacecraft alone, there will be a great deal of data that will be useful for repeat pass interferometric SAR processing. Current tools are nearly all geared to examining individual frames or areas, and are not general enough or generalizable enough to allow researchers

to explore the richness of the data in space and time. The ISCE framework and radar processing software associated with it, described in this paper, are designed to be extensible and flexible enough to allow the researcher to ask questions and formulate new ways to answer them with relative ease in the environment.

The development to date has all the basic elements of the framework in place. We are now refining these elements, creating data and metadata objects for specific data sources and problem sets, and completing the recasting of our algorithms into the framework. We anticipate a documented and usable set of code in the coming year, with ample time for community feedback, bug fixes, and refinement during the life of the funded AIST development. And of course, we anticipate that the code will be useful enough, well enough documented, and accessible enough for the ISCE to far outlive the duration of the AIST program.

ACKNOWLEDGMENT

We thank Piyush Shanker and Cody Wortham, both PhD candidates at Stanford University, for their work on the Permanent Scatterer and SBAS algorithm development and validation as well as scripting work all contained in the Stanford legacy code at the core of ISCE. The authors would like to thank the Earth Science Technology Office at NASA for support. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with NASA, and at Stanford University under a contract with JPL.

REFERENCES

- [1] Rosen, PA, A. Donnellan, Z. Liu, B. Hager, P. Lundgren, F. Webb, S. Yun (2009). DESDynI's Ability to Estimate Source Parameters for Solid Earth Science Applications, Proceedings of the IEEE 2009 International Geoscience and Remote Sensing Symposium Conference, Capetown, South Africa.
- [2] Simons, M. and P. A. Rosen (2007). Interferometric Synthetic Aperture Radar Geodesy, in Treatise on Geophysics, G. Schubert Ed., Elsevier.
- [3] Farr, TG, PA Rosen, E Caro, R Crippen, R Duren, S Hensley, M Kobrick, M Paller, E Rodriguez, L Roth, D Seal, S Shaffer, J Shimada, J Umland, M Werner, M Oskin, D Burbank, D Alsdorf (2007). The Shuttle Radar Topography Mission, Reviews of Geophysics, Vol. 45, No. 2, RG2004.
- [4] Hensley, S., K. Wheeler, G. Sadowy, C. Jones, S. Shaffer, H. Zebker, T. Miller, B. Heavey, E. Chuang, R. Chao, K. Vines, K. Nishimoto, J. Prater, B. Carrico, N. Chamberlain, J. Shimada, M. Simard, B. Chapman, R. Muellerschoen, C. Le, T. Michel, G. Hamilton, D. Robison, G. Neumann, R. Meyer, P. Smith, J. Granger, P. Rosen, D. Flower, R. Smith (2008). The UAVSAR instrument: description and first results, 2008 IEEE Radar Conference, 6 pp.