To appear in **Software: Practice and Experience**[1]
*Submitted*: 22 May 2006; *Accepted*: 17 May 2007

# A component-based framework for radio-astronomical imaging software systems

A. J. Kemball[2], R. M. Crutcher, and R. Hasan

*National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 1205 W. Clark Street, Urbana, IL 61801, USA*

## ABSTRACT

This paper describes a component-based framework for radio-astronomical imaging software systems. We consider optimal re-use strategies for packages of disparate architectures brought together within a modern component framework. In this practical case study, the legacy codes include both procedural and object-oriented architectures. We consider also the special requirements on scientific component middleware, with a specific focus on high-performance computing. We present an example application in this component architecture and outline future development planned for this project.

*Subject headings:* Component-based software engineering; software re-use; domain analysis; scientific computing

## 1. INTRODUCTION

Component-based software engineering (CBSE) is an active area of research and development holding great promise for reducing the cost and managing the complexity of large-scale software systems (16; 32). There are however important open theoretical and practical questions in this area of computing practice. These include foundational discussions concerning the definition of components, their properties, and communication patterns (3; 30), as well as the best component engineering techniques by which CBSE can be used to

[2]E-mail: akemball@uiuc.edu

[1]This is a preprint of an article accepted for publication in Software: Practice and Experience, Copyright C 2007 (John Wiley & Sons, Ltd.)

routinely construct reliable, large-scale software systems at low cost using standard generic component composition techniques that are similar to those used in established engineering disciplines (1).

We consider the practical application of CBSE techniques in a specific problem domain, namely imaging software systems used in radio astronomy. In this branch of astronomy, images are formed using computationally-intensive inverse imaging techniques. Several large community codes have been developed in recent decades to support the analysis and imaging of radio astronomy data of this type. The existing codes were primarily developed for interactive, single-user use and are not always optimized for high-performance computing as a result. As long-term software development efforts, they are also prone to architectural drift, complexity, and rising maintenance and evolution costs. Efforts to evolve the architecture and features of existing community codes are referenced in the text below. In this paper, we describe a new component-based framework tailored to meet the current and future computational and scientific challenges of this particular scientific field, and outline our practical experience with applying CBSE in this domain.

We adopt the definition that components are units of independent deployment and third-party composition; these components interact and are composed within a standard provided by a component model (8).

The following important open questions concerning the application of CBSE in constructing complex software systems relate directly to our case study: i) optimal techniques for arriving at the true generic component decomposition in a given problem domain; ii) the specification and implementation of domain component frameworks; and iii) component adaptation techniques. The use of CBSE in this context is closely related to re-use based software engineering (RBSE) (21), particularly in so far as it concerns the re-factoring and adaptation of legacy software resources within a modern component framework.

The premise that there is an optimal component decomposition in a given problem domain is related to the underlying concept and challenge of generic programming (23; 24). In generic programming, a goal is to partition algorithmic building blocks in a form that can be re-used directly and generically in the widest possible range of higher-level applications. Analogously, an optimal CBSE component decomposition in a given problem domain can be considered that decomposition which allows maximal re-use in higher-level application development across the domain. We consider strategies for how to arrive at this optimal decomposition in the problem domain that is the subject of this study. We also describe a component framework which allows re-use and interoperability of existing software codes developed using both procedural and object-oriented design methodologies and include an analysis of component adaptation techniques used for casting legacy code into this component

architecture. In closing, we consider the special requirements posed by scientific CBSE, where high-end, parallel, and grid computing are important application drivers, particularly where these requirements relate to component decomposition, middleware, or component model choices.

The paper is structured as follows: we briefly describe the problem domain in Section 2, the component architecture in Section 3, and illustrate an example application in Section 4. Related work is summarized in Section 5. The strengths and weaknesses of the component framework and planned future work are considered in Section 6.

## 2. IMAGING SOFTWARE SYSTEMS IN RADIO ASTRONOMY

This section describes the technique of radio interferometry and reviews current community codes and future computing challenges in this scientific domain.

### 2.1. Radio interferometry

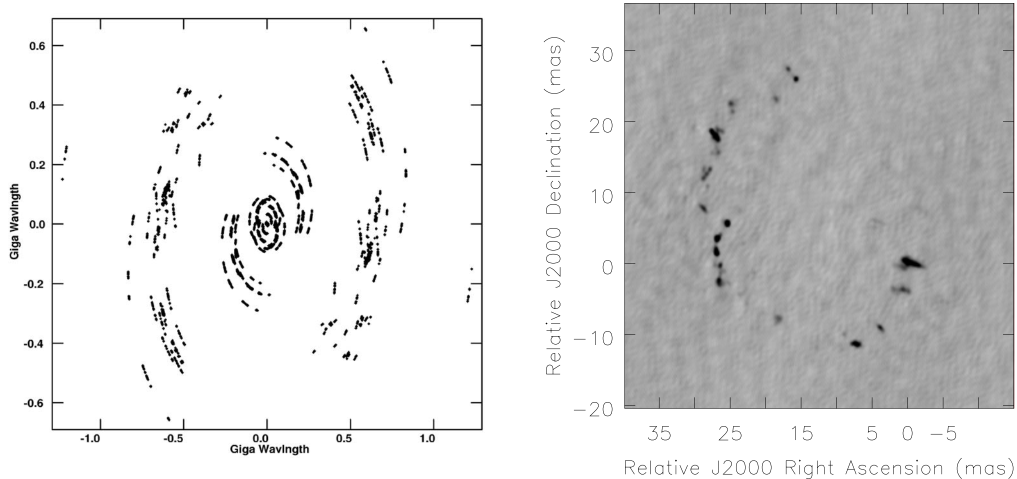
Radio interferometers, which comprise arrays of radio telescopes such as the Very Long Baseline Array (VLBA)[1] or the Combined Array for Research in Millimeter-wave Astronomy (CARMA)[2], form astronomical images using a computationally-intensive inverse imaging technique. The fundamental principle underlying the inverse imaging method is that the correlated signals between antennas in the array and the image of the astronomical source being observed are related by an integral equation of Fourier transform type (33). Further details of this mathematical relationship are described by (14). The primary responsibility of imaging software systems in radio interferometry is to calibrate the observed correlated data for instrumental and propagation contamination effects, and to transform these data into an astronomical image using image reconstruction and deconvolution algorithms.

The correlated antenna data are referred to as visibility data; they are sampled in a plane, defined by convention as the $(u, v)-$plane, orthogonal to the direction to the astronomical source. Earth rotation causes each antenna baseline to trace an arc in the $(u, v)-$plane over time. The fidelity of the image reconstruction is dependent on the spatial density of the visibility-plane sampling and the accuracy with which the instrumental antenna and

---

[1] http://www.vlba.nrao.edu

[2] http://www.mmarray.org




(a) $(u, v)$-plane coverage.

(b) Radio-interferometric image

Fig. 1.— This figure shows the $(u, v)$-plane sampling (a) and associated reconstructed image obtained from VLBA observations of the 43.1 GHz SiO maser emission toward the late-type star TX Cam. The radio emission has been averaged over source velocity to produce a two-dimensional projected image (b); the $(u, v)$-plane coverage is plotted in units of $10^9$ wavelengths and the image in angular coordinates of milliarcseconds on the plane of the sky.

atmospheric calibration can be determined. Image formation using this technique typically requires a complex data reduction sequence of multiple computational steps.

An example of $(u, v)-$plane coverage, and an associated radio-astronomical image, are shown in Figure 1.

### 2.2. Community codes

As the scientific quality of radio-astronomical images is strongly dependent on these image reconstruction techniques, several imaging software systems have been developed in the community since the advent of routine radio interferometry in the 1970's. These software packages typically have lifespans of at least a decade (and frequently much longer) and are often designed with a specific instrumental or scientific focus. These objectives may evolve over time, subject to any limits imposed by low-level instrumental assumptions built into the package that may govern the extent of future evolution.

Community codes have been developed using both procedural and object-oriented software engineering approaches. We consider three representative community-code packages in this study, which are listed in Table 1. This list is confined to those packages studied in this paper, however, and is not intended as a complete list of all available community codes in this scientific discipline.

The AIPS[3] package started development in the mid-1970's and is supported by the National Radio Astronomy Observatory (NRAO) (13). This package has primarily, but not exclusively, been used for reduction and analysis of data from the centimeter-wavelength radio interferometers operated by the NRAO, such as the Very Large Array (VLA)[4], and the VLBA. The code is written primarily in Fortran 77, with limited operating system-specific code implemented in the C language. The AIPS package was developed using a procedural programming model. The package is structured as a set of individual task-oriented applications; these can be launched from a command-line environment with basic scripting capability. A Python binding has recently been developed to the AIPS tasks and data format (18).

The MIRIAD[5] project was started at the National Center for Supercomputing Appli-

---

[3]http://www.aoc.nrao.edu/aips

[4]http://www.vla.nrao.edu

[5]http://www.astro.umd.edu/ teuben/miriad

cations (NCSA) in the late 1980's, and was initially targeted at the millimeter-wavelength Berkeley-Illinois-Maryland Array (BIMA)[6]. Subsequent development proceeded in a loosely-coupled collaborative consortium of universities and academic institutions. MIRIAD is in use by other millimeter-wavelength arrays, such as CARMA, which is currently in commissioning. The package has also been extended for use by centimeter-wavelength arrays, such as the Australia Telescope Compact Array (ATCA)[7]. MIRIAD was developed within a procedural programming model, and is implemented primarily in Fortran 77, with some low-level code, primarily for I/O, implemented in the C language. This package has a similar task-oriented architecture as AIPS; applications can be launched from a command-line parameter-setting environment or directly from an operating system shell.

The AIPS++[8] package was developed by a consortium of academic institutions and observatories, including NCSA, starting in the early to mid-1990's and intended as a successor to the earlier generation of packages such as AIPS and MIRIAD. At the outset, AIPS++ employed an imaging model targeted at generic interferometry (14), in an attempt to encompass the instrumental needs of all telescopes operated by the consortium partners. AIPS++ was the first community code developed in this discipline to use object-oriented techniques, and is primarily implemented in C++ with a scripting interface to the underlying objects and methods provided by the Glish interpreted language (29). Some computational kernels in this package are implemented in Fortran 77 for efficiency. This package is currently undergoing re-structuring both to provide a more task-oriented interface as well as to provide further bindings in addition to Glish (15; 20).

For the most part, the development costs of the community codes listed in Table 1 are not known accurately, primarily as a result of the informal academic software engineering approaches used. However, we can estimate an approximate commercial replacement cost based on the size of each package measured as the number of source lines of code (SLOC). Using the default COCOMO I cost model (5) reported by SLOCCount, $COST = 2.4\ (KSLOC)^{1.05}$, the nominal collective commercial replacement cost for this sampling of community codes is of order $\sim 480$ person-years. The deployment of the packages in the user community over several decades represents a significant investment in user and integration testing. Given limited development budgets in astronomy instrument construction, these prior investments argue strongly for aggressive software re-use in this problem domain. In this paper we consider optimal re-use strategies for packages of disparate architectures brought together within

[6]http://bima.astro.umd.edu

[7]http://www.narrabri.atnf.csiro.au

[8]http://www.aips2.nrao.edu

a modern CBSE framework.

### 2.3. Future challenges

Modern imaging software systems for radio astronomy face significant current and future challenges. Moore's Law (22) has enabled exponential increases in data output rates from the electronics and instrumentation used in constructing modern radio telescopes; this in turn allows increasingly challenging science goals to be addressed, with associated pressures on the computational complexity of imaging and calibration algorithms. This, in turn, requires an increasing focus on high-end computing (HEC) in data reduction software systems for radio astronomy imaging. For example, the current VLA, which was commissioned in the late 1970's, has a peak data output rate of $\sim 35$ kBps; by contrast the future Square Kilometer Array (SKA), currently under design and development, currently anticipates a mean output data rate of $\sim 17$ GBps with an associated peak computational requirement of 1 PFlop [9].

## 3. COMPONENT ARCHITECTURE

The primary goal of this work is to develop the framework and architecture for a next-generation radio astronomy imaging software system that facilitates re-use of existing community code investments, but which also provides a platform for developing solutions to the most demanding computational science and engineering challenges in this scientific discipline. Geometric advances in available community HEC resources and data-handling capacity are enabling breakthrough applications across the physical sciences (2), and astronomy is no exception (9). An additional goal of this framework is to allow the re-use of shared cyber-infrastructure (CI) developed for grid and distributed HEC environments to enable these scientific breakthroughs across the physical sciences.

### 3.1. Component decomposition

We have chosen a component-based architecture for this study first because of the inherent advantages offered by CBSE but also because it offers a unifying framework for the re-use of existing community codes developed using both procedural and object-oriented paradigms. For any given problem domain, there exists an optimal decomposition into

[9] http://www.skatelescope.org

software components which captures the true generic decomposition of the problem space, and which is maximal in the sense of allowing the greatest degree of direct component re-use in application development across the domain. This decomposition is unknown a priori and the challenge of CBSE and generic programming is to determine the optimal software engineering approach to arrive at this decomposition.

For the problem domain considered here, our starting point in determining the generic software component decomposition are the established unified mathematical models for image formation in radio-astronomical interferometry (14). These generic models abstract away telescope-specific instrumental differences and provide a unified common mathematical framework for domain analysis encompassing a broad range of current and future telescopes. The mathematical model needs to be complete in the sense that no instrumental effect can be removed without compromising the image formation process, as opposed to there being no remaining contemplated instrumental effect which could possibly be added. As such, the model specification process is bounded and limited. This expresses the general principle that completeness is achieved when there is nothing left to remove rather than nothing left to add[1]. A unified imaging model also maximizes the re-use potential of existing diverse community code modules which may each only partially fulfill the optimal generic component decomposition for this domain as a whole.

The design process we have followed for determining the optimal component decomposition involves factoring the common mathematical model arising from the domain analysis into sets of abstract data types (ADT) (data components) and functional operations (functional components). Each ADT conforms with an interface design pattern defined in terms of a set of elemental methods which provide for: i) component initialization from a specified set of input representations (e.g. XML); ii) parameter accessors; iii) basic operations fundamental to the data type; and iv) component conversion to a set of specified output representations. The functional components capture the imaging and calibration operations needed for image formation within the imaging model framework. Their interface design pattern similarly includes methods for initialization and conversion to and from different representations, but for functional components these data are the initialization parameters for the functional methods provided by the component. The data and functional components each fall within a hierarchical structure, with increasing domain-specificity at higher levels of the hierarchy. Components at the lowest levels of the hierarchy have the greatest overlap with shared CI components and services, as shown in Figure 2. The user interface is shown at the highest level of this layered architecture. Increasingly complex applications

[1] "Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove"; quotation by writer Antoine de Saint-Exupery (1900-1944)

Table 1: Sample community codes for radio astronomy imaging

| Package Name | Development languages (ordered by prevalence) | Size (MSLOC[1]) | Reference |
|---|---|---|---|
| Astronomical Image Processing System (AIPS)[2] | Fortran 77, C | 0.6 | (13) |
| Multi-channel Image Reconstruction, Image Analysis, and Display (MIRIAD)[3] | Fortran 77, C | 0.2 | (28) |
| Astronomical Information Processing System (AIPS++)[4] | C++, Glish(29), Fortran 77 | 1.0 | (12) |

[1]: MSLOC=$10^6$ SLOC, as measured by SLOCCount (written by David A. Wheeler)
[2]: Modified version of base release 15OCT97
[3]: Release v4
[4]: Modified version of code base v1.8 #667

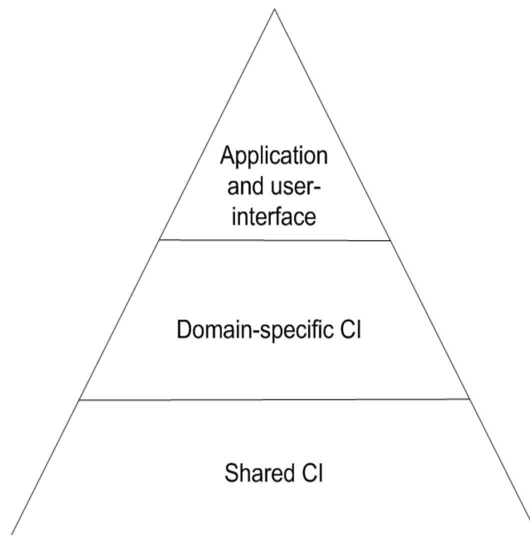



Fig. 2.— Tiered cyberinfrastructure architecture.

are composed at higher levels using components from the lower levels of the architecture.

The operations which are implemented as part of each ADT interface are constrained to be low-level operations, closely coupled to the data type itself. Higher-level operations are implemented in functional components in which the ADT will appear in the method signature.

As noted in the previous section, the existing community codes are implemented in both procedural and object-oriented paradigms and each has an implicit mathematical model for imaging which is typically a subset of the common domain model. As a result, the re-usable resources from the community codes – be they procedures, objects, libraries, executables, requirements, or designs – represent varying degrees of partial fulfillment of the generic domain component interfaces adopted for the framework described here. Given that the existing community codes collectively represent large investments in development and user testing, the union of their interfaces and functionality, where they match ADT and functional components in the generic decomposition adopted here, provide a good test of design completeness for this CBSE framework. In this sense, our re-use of the existing software products is in the broadest possible sense (21).

### 3.2. Component framework

We adopt a layered component framework shown in Figure 3. In this diagram, the interface for generic domain components, both functional or ADT, is shown at level A. The software resources targeted for re-use are shown at level D; these package codes are not in component form and do not match the ADT or functionality expressed in the domain component interfaces. We employ component adapters in layer C to map the re-usable code in each package to the generic domain component interface. In general, each adapter provides, using white-box component adaptation (32) in this layer, a partial fulfillment of the domain component interface, governed by the subset of functionality implemented in the underlying community code. Specific package implementations of the domain component interfaces are represented in this architecture at level B. The architecture allows the domain component interface to be fulfilled by composition using several package component implementations. It also allows multiple component implementations in B for a given domain component interface, where this is useful in order to assess different imaging algorithms implemented in the underlying community codes and in order to test equivalent components for correctness or performance by direct inter-comparison. Domain components in this architecture also implement their own functionality directly if re-use from lower layers is neither possible nor cost-effective.

### 3.3. Component model

The vital intellectual products of the framework are the generic domain decomposition as well as the component adapters which facilitate legacy code re-use. The choice of a middleware component technology is separate from the domain analysis, but needs to be made judiciously in order to achieve the overall goals and requirements discussed in Section 2. As there is a specific need in this development for support of HEC and parallel computing, we have adopted the Babel binding (19), which is also used by the Common Component Architecture (CCA[2]) (4), a community scientific CBSE initiative with an emphasis on HEC. Additionally, this software also provides optimized multi-dimensional scientific array support, and a language-neutral peer-to-peer component binding which supports our target languages of C, C++, FORTRAN, and Python. These were further considerations in its adoption. Babel specifies component interfaces in terms of a scientific interface definition language (SIDL). Further detail on the CCA project goals and vision is provided in the summary of related work in Section 5 below.

The need to deploy scientific components in an HEC or grid environment places constraints on the optimal middleware choice, as discussed above, but also on the component model. To ensure scalability in this environment the component designs need to be constrained so that any significant use of memory cache or locality management, I/O, and computation are predictable and configurable. This enables both scalability in a given HEC environment, but also future portability across different HEC or grid hardware architectures, an architectural approach described by (17). On each host, we also favor deployment of applications containing multiple components in a single process, to allow in-process virtual function dispatch as is possible in Babel, over client-server method invocation, in order to have predictable latencies. These special emphases on HEC are a unique requirement on the component framework we describe here; current community code designs are generally not optimized for HEC needs.

## 4. EXAMPLE APPLICATION: IMAGE COMPONENT FITTER

As an example application in this framework we consider the image analysis problem of automatically locating and fitting source components in astronomical images (7; 36). Individual components in radio-interferometric images are usually modeled as having a Gaussian form. Extracting their properties is an important element of the quantitative scientific inter-

[2]http://www.cca-forum.org

pretation of radio-interferometric astronomical images. Such measurements allow the determination of source sizes, brightness, and proper motions over time, amongst other scientific applications. Automatically finding and fitting a collection of components in an astronomical image is challenging because of the blending of individual components and the non-uniform noise statistics across the image due to residual calibration and deconvolution artifacts. Each two-dimensional Gaussian component has six free parameters while each three-dimensional Gaussian component has nine; fits to these components need to be well-constrained for convergence as a result, especially for heavily blended components. Automated component extraction algorithms need to set these constraints reliably without user interaction.

We consider here a recent algorithm for automatically locating, deblending, and fitting two- or three-dimensional components in astronomical images (D. Perley et al, in preparation), which was developed as a research project in the AIPS++ package.

In Figure 4, we show the component architecture of an application in our framework to apply this algorithm to astronomical images stored in the FITS format (35). This image-fitting component application re-uses FITS I/O capabilities from the MIRIAD package and the image-fitting algorithm itself as implemented in the AIPS++ package. They are connected through generic domain component interfaces within our component framework. In this particular instance, the re-used FITS I/O capabilities were written in FORTRAN 77, the FITSImageIO adapter code in FORTRAN 90, and the other components in the architecture in C++. The Babel middleware automatically supports peer-to-peer bindings from the component SIDL specification in all supported languages, including C, C++, FORTRAN 77 and 90, Java, and Python. In our component framework we generate client bindings for C++, FORTRAN 90, and Python; as a result the components in Figure 4 are all accessible from these compiled language or scripting environments.

As an example of the use of this component application, we consider a magnified region of the radio-interferometric image shown in Figure 1, centered on the left-most midpoint of the projected shell of emission. The selected region contains three components, as plotted in the left panel of Figure 5. The algorithm automatically located, deblended, and fitted all three Gaussian components above the specified noise threshold in this image; these components are plotted in the center panel of Figure 5. The right-most panel of this figure contains the residual image obtained by subtracting image 5(a) and image 5(b).

Our development of initial applications in this component framework, including this example application, has raised several practical issues and lessons learned; these are described in further detail in the remainder of this section.

The component re-factoring approach we have adopted is based on white-box re-use (32),

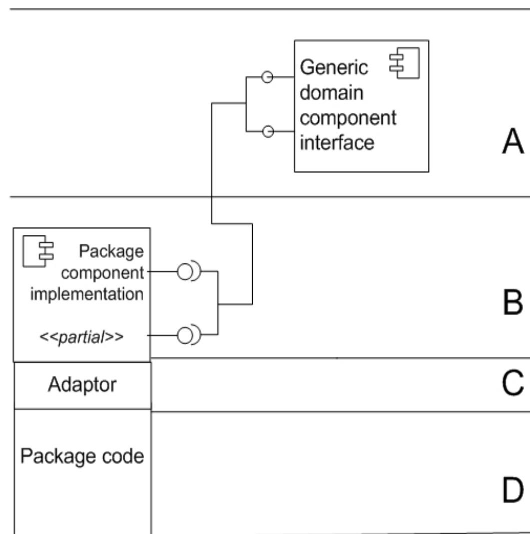



Fig. 3.— Layered architecture of the component-based framework.

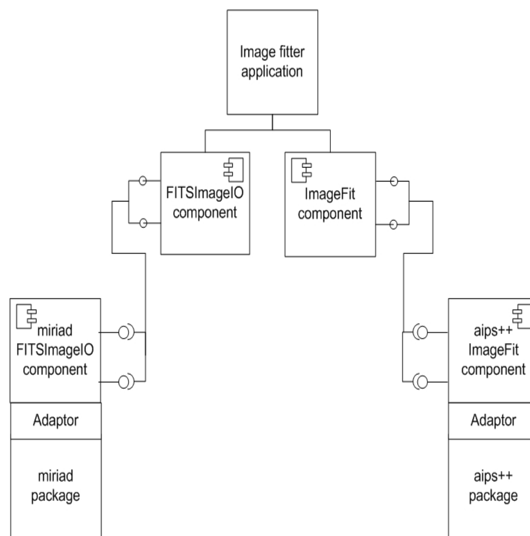
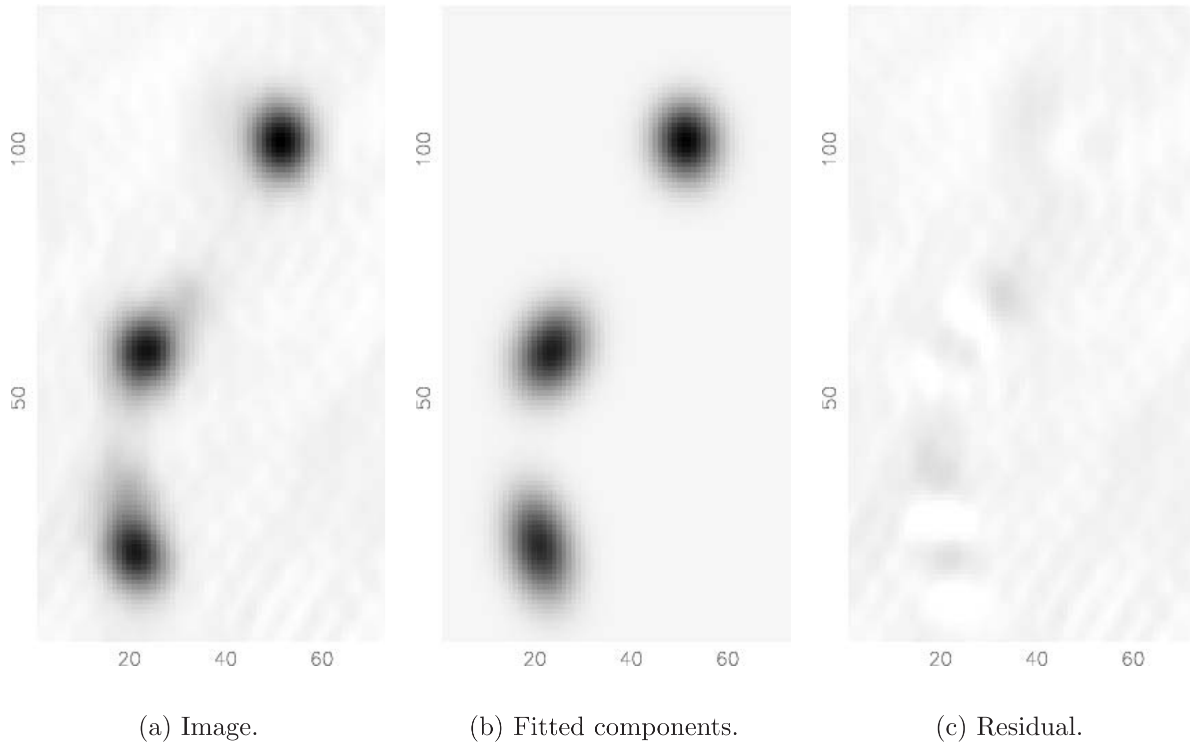



Fig. 4.— Component interaction diagram for the example image fitter application.




(a) Image. (b) Fitted components. (c) Residual.

Fig. 5.— The left panel (a) shows a magnified region of the radio image in Figure 1. Fitted Gaussian source components, as determined by an automated fitting algorithm, are shown in the center (b). The residual obtained by subtracting the fitted components from the original image in (a) is shown in the right panel (c).

although this is confined substantially to the component adaptation layer. White-box re-use offers the greatest flexibility in a common build environment and we have implemented this for both the legacy packages and component framework using the GNU Autotools system (34). The build system needs to provide robust and portable support for shared and dynamic library generation so that the components can be made available for dynamic loading into a scripting layer. This places a greater premium on efficient decomposition of the legacy package libraries into smaller, less dependent shared libraries, than they may ordinarily produce in their intrinsic build systems. This optimization was performed, where needed, as the legacy packages were integrated into the common build system.

Our component interfaces are first defined in terms of Babel SIDL files. The interfaces are designed using the interface patterns for data and functional components described in section 3.1. The component interfaces are implemented as stand-alone classes or modules, depending on the choice of object-oriented or procedural implementation language, that can be built both outside of the Babel component middleware model and also automatically parsed during the build using the PDT and Chasm tools (27) to generate the binding code required for integration in the component middleware model. This approach separates component implementation from component middleware integration and was found to be valuable in development and testing.

Our example application, though intentionally simple in scope, illustrates that CBSE is an efficient and scientifically-effective method for re-factoring and re-using legacy community codes within a modern component architecture. This approach offers several practical advantages over traditional programming: i) it allows efficient re-use of independently-developed and tested scientific capabilities from separate legacy codes; this is substantially more cost-effective than traditional development; ii) the requirement that components be independent units of deployment forces a cleaner separation of interface from implementation, and the specification of more generic component interfaces than might arise in traditional class or module design; this, in turn, facilitates a greater likelihood of future component re-use; and iii) the use of a component binding model optimized for scientific languages and data types, such as Babel, allows efficient integration of multi-language implementations in a parent component architecture; we found this component model to hold advantages over commercial component middleware in this scientific problem domain.

## 5. RELATED WORK

A clear overarching vision for the role of CBSE in addressing pressing problems in large-scale scientific computing is presented by the CCA and Babel projects (4; 19). Challenging

problems in high-performance computing now frequently require the multi-scale coupling of increasingly complex codes which each address a different physical or data analysis process, and which are likely to be developed by separate groups. These large codes represent significant prior investments of resources, and capture unique expertise in each individual area; however, they will likely have been developed using a diversity of languages, architectural models, and software engineering methodologies. The development of a common component model for high-performance scientific computing holds the promise of managing the escalating implementation complexity of large scientific codes, and opens the possibility of establishing repositories of re-usable components, further lowering future costs in this area and opening new scientific opportunities.

Several studies have considered the general problem of incrementally migrating large monolithic legacy software systems to a component-based architecture. Piecemeal migration of a large document archiving and retrieval software system using an integration pattern language approach is described by (11). Further important design patterns for component and language integration for in-situ legacy migration are described by (37). Both papers cover the issue of component adaptation as part of these studies. This specific question is considered in further detail by (6), where the traditional approaches to component adaptation of copy-paste, inheritance, and wrapping are reviewed, and a new method presented for layered indirection known as component superimposition.

The problem of dynamic component adaptation is considered by (25), in the context of maintaining interoperability between evolving web service interfaces published by separate providers. While the underlying technological implementation differs in this instance, the analysis is broadly applicable to components. A framework for dynamic correctness and performance testing of multiple component versions is presented by (26).

## 6. DISCUSSION AND CONCLUSIONS

In our case study, we find that CBSE is a good design paradigm for re-using and combining both procedural and object-oriented legacy codes, and for developing a modern framework for radio astronomy imaging software systems. Re-factoring in terms of generic domain component interfaces also produces a component model and architecture that has several key advantages in this problem domain. The generic component interface decomposition is a vital intellectual product in and of itself; this process encourages community standardization and interoperability within the problem domain as well as with other scientific components, an important goal recognized and targeted by the CCA forum (4). The refinement of standardized generic domain component interfaces allows open development of

separate implementations that can inter-operate reliably; this is an important requirement in the academic research community.

Components separate interface from implementation more strongly than other design paradigms. Components therefore uniquely enable extensibility and composition, an important requirement in this science domain in which new applications are driven by rapidly evolving research and scientific priorities. Re-use and extensibility are long-standing general goals in software engineering, but have proven difficult to achieve in practice. Our case study in this problem domain, which includes both procedural and object-oriented legacy code, confirms prior experience regarding impediments to software re-use and extension. Procedural libraries present barriers to re-use if their shared data are poorly encapsulated, as is commonly the case. Large object-oriented libraries have not proven to be inherently good repositories of re-usable and extensible software (10; 31; 32); in practice more knowledge of their implementation is required for re-use than interface specifications alone, as is our experience in this study. In both cases, non-generic assumptions or designs are a general impediment to re-use; our approach of allowing partial fulfillment of a common generic domain component interface has proven to be a workable strategy to offset this difficulty. The requirement that components be independent units of deployment forces a cleaner separation of interface and implementation, and we believe that component re-factoring is a promising approach in this problem domain.

The component engineering approach we have adopted in this study, namely the identification of generic domain data types, expressed as data components, and domain operations, expressed as functional components, cross-checked for completeness against the union of existing legacy implementations, has proven a good means by which to arrive at an optimal component interface decomposition. This is further aided by the adoption of a common design pattern for the component interfaces of each type.

In this case study, Babel has proven to be a good component middleware choice. It is well-suited to scientific problem domains, such as radio astronomy imaging, due to the support for multi-dimensional arrays, FORTRAN bindings (a language not in common general use), good interoperability with HPC, and peer-to-peer language bindings. The latter property is particularly useful in providing developer choice in component implementation and in providing a general scripting interface using Python. This scripting binding also fulfills the role of the command language integration pattern described by (37).

Our component adaptation strategy is based on white-box re-use in an adaptation layer. White-box re-use is vulnerable to evolution in the underlying re-used code but we have mitigated this through primary isolation in the adapter layer. This approach matches the component wrapper and explicit import/export integration patterns described by (37). The

wrapper layer provides a white-box indirection layer which exports a stable domain interface and which can absorb evolution in re-used black-box code beneath; it also provides a centralized point for customization, type conversion, decoration, or message interception (11; 37). On balance, we have found this component adaptation approach to be a good strategy in this case study.

In summary, the component framework described in this paper has proven to be a workable architecture and design for this problem domain and for the goals outlined at the start of this paper. In future work we plan to continue adding applications to this framework with a strong focus on the most computationally demanding and data-intensive applications.

## REFERENCES

Apperly, H. The component industry metaphor. In *Component-Based Software Engineering,* Heineman GT, Councill, WT (eds.). Addison-Wesley, 2001; 21-32.

Revolutionizing science and engineering through cyberinfrastructure. Atkins DE. (ed.). http://www.cise.nsf.gov/sci/reports/atkins.pdf [28 September 2005].

Maurer, J. A conversation with Roger Sessions and Terry Coatta. *ACM Queue* 2005; **3**(7):16-25.

Bernholdt DE, Elwasif WR, Kohl JA, Epperly TGW. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02),* 2002.

Boehm, BW.*Software engineering economics.* Prentice Hall, 1981.

Bosch, J. Superimposition: a component adaptation technique. *Information and Software Technology* 1999; **41**(5):257-273.

Colomer, F, Reid, MJ, Menten, KM, Bujarrabal, V. The spatial and velocity structure of circumstellar water masers. *Astronomy and Astrophysics* 2000; **355**:979-993.

Councill, B, Heineman, GT. Definition of a software component and its elements. In *Component-Based Software Engineering,* Heineman GT, Councill, WT (eds.). Addison-Wesley, 2001; 5-19.

Djorgovski, SG. Virtual astronomy, information technology, and the new scientific methodology. In *Proceedings of the Seventh International Workshop on Computer Architecture*

*for Machine Perception* Di Gesú, V, Tegolo, D. (eds.). IEEE Computer Society:Los Alamitos, 2005; 125-132.

Emmerich, W. Distributed component technologies and their software engineering implications. In *Proceedings of the International Conference on Software Engineering,* ACM, 2002; 537-546.

Goedicke, M, Zdun, U. Piecemeal legacy migrating with an architectural pattern language: a case study. *Journal of Software Maintenance: Research and Practice* 2002; **14**(1):1-30.

Glendenning, BE. Creating an object-oriented software system – The AIPS++ experience. In *Astronomical Data Analysis Software and Systems V, ASP Conference Series Vol. 101,* Jacoby, GH, Barnes, J. (eds.). PASP: San Francisco, 1996; 271-280.

Greisen, EW. The VLA - AIPS. In *Radio Interferometry: The Saga and the Science,* Finley DG, Goss WM. (eds.). NRAO:Socorro, 2000; 57-74.

Hamaker JP, Bregman JD, Sault RJ. Understanding radio polarimetry. I. Mathematical foundations. *Astronomy and Astrophysics Supplement Series* 1996; **117**:137-147.

Harrington, S, DeBonis, D, McMullin, J, Young, W, Chiozzi, G, Jeram, B. ACS as the framework for offline data reduction in ALMA. In *Astronomical Data Analysis Software and Systems XV, ASP Conference Series Vol. 351*, Gabriel,C, Arviset,C, Ponz,D, Solano,E, (eds.). PASP:San Francisco, 2006; 259-262.

Heineman, GT, Councill, WT. *Component-Based Software Engineering.* Addison-Wesley, 2001.

Kendall RA, Apra E, Bernholdt DE, Bylaska EJ, Dupuis M, Fann GI, Harrison RJ, Ju J, Nichols JA, Nieplocha J, Straatsma TP, Windus TL, Wong AT. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications* 2000;**128**:260-283.

Kettenis, M, van Langevelde, HJ, Cotton, B. ParselTongue: AIPS talking Python. In *Astronomical Data Analysis Software and Systems XV, ASP Conference Series Vol. 351*, Gabriel,C, Arviset,C, Ponz,D, Solano,E, (eds.). PASP:San Francisco, 2006; 497-500.

Kohn S, Kumfert G, Painter J, Ribbens C. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing,* SIAM, 2001.

McMullin, JP, Schiebel, DR, Young, W, DeBonis, D. AIPS++ framework migration. In *Astronomical Data Analysis Software and Systems XV, ASP Conference Series Vol. 351*, Gabriel,C, Arviset,C, Ponz,D, Solano,E, (eds.). PASP:San Francisco, 2006; 319-322.

Mili H, Mili A, Yacoub S, Addy, E. *Reuse-Based Software Engineering.* Wiley-Interscience, 2001.

Moore, GE. Cramming more components onto integrated circuits. *Electronics* 1965; **38**(8);114-117.

Musser DR, Stepanov AA. A library of generic algorithms in Ada. In *Proceedings of the 1987 Annual ACM SIGAda International Conference on Ada,* Brosgol BM. (ed.). ACM, 1987; 216-225.

Musser DR, Stepanov AA. Generic programming. In *Lecture Notes in Computer Science Vol. 358,* 1989; 13-25.

Ponnekanti, S, Fox, A. Interoperability among independently evolving web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware,* Springer-Verlag:New York (NY), 2004; 331-351.

Rakic M, Medvidovic, N. Increasing the confidence in off-the-shelf components: a software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability,* ACM:New York (NY), 2001; 11-18.

Rasmussen CE, Lindlan KA, Mohr B, Striegnitz J. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. In *Proceedings of the 2nd Annual Los Alamos Computer Science Symposium,* Technical Report FZJ-ZAM-IB-2001-10, ZAM:Jülich, 2001.

Sault RJ, Teuben, PJ, Wright, MCH. A retrospective view of Miriad. In *Astronomical Data Analysis Software and Systems IV, ASP Conference Series Vol. 77,* Shaw RA, Payne, HE, Hayes, JJE. (eds.). PASP: Provo (UT), 1995; 433-437.

Schiebel DR. Event driven programming with Glish. In *Astronomical Data Analysis Software and Systems IX, ASP Conference Series Vol. 216,* Manset N, Veillet C, Crabtree, D. (eds.). PASP: San Francisco, 2000; 39-48.

Sessions, R. Fuzzy boundaries: objects, components, and web services. *ACM Queue* 2004; **2**(9):40-47.

Sullivan, KJ. Designing models of modularity and integration. In *Component-Based Software Engineering,* Heineman GT, Councill, WT (eds.). Addison-Wesley, 2001; 341-354.

Szyperski, C. *Component Software.* Addison-Wesley, 1997.

Thompson AR, Moran JM, Swenson Jr GW. *Interferometry and Synthesis in Radio Astronomy.* Wiley: New York, 2001.

Vaughan, GV, Elliston, B, Tromey, T, Taylor, IL. *GNU Autoconf, Automake, and Libtool.* New Riders:Indianapolis (IN), 2000.

Wells, DC, Greisen, EW, Harten, RH. FITS - A flexible image transport system. *Astronomy & Astrohpysics Suppl Ser.* 1981; **44**, 363-370.

Williams, JP, de Geus, EJ, Blitz, L. Determining structure in molecular clouds. *Astrophysical Journal* 1994; **428**:693-712.

Zdun, U. Some patterns of component and language integration. In *Proceesings of the 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004),* 2004; 1-26.

---

This preprint was prepared with the AAS LaTeX macros v5.2.