Meta-Component Architecture for Software Interoperability

Ly Danielle Sauer
Sandia National Laboratories
ldsauer@sandia.gov

Robert L. Clay
Sandia National Laboratories
rlclay@ca.sandia.gov

Rob Armstrong
Sandia National Laboratories
rob@ca.sandia.gov

Abstract

Most existing software is one-of-a-kind, monolithic, non-interoperable, and consequently, non-reusable. In addition, this software is difficult to maintain, improve, and scale. More importantly, this software is vital to many enterprises and institutions. Thus, enterprises must continuously make trade-off decisions between developing new software and maintaining existing software. The meta-component architecture (Component Mill) presented in this paper will enable enterprises to continue using existing software while providing a mechanism to migrate the software into a format (meta-component) that supports software integration and reuse. This architecture provides the blueprint for realizing an environment that supports exposing existing software for reuse with other (heterogeneous) software while allowing software development based on reuse. The meta-components are independent of any component model used in component technologies. Thus, this architecture provides components that are, in principle, executable in any component technology.

1. Introduction

In general, software development remains a handcrafted activity that obeys laws that can notcannot be generalized [5][6]. Software developed in this manner could lead to products being delivered behind schedule, over budget, with substandard quality, and with one of a kind solutions. Generally, this software is not interoperable and can not be reused. Moreover, existing software may contain critical information or may be core to the enterprises competitive edge. At Sandia National Laboratories (Sandia), the complexities of these problems are growing with the addition of multi-physics scientific computing software developed to execute on massively parallel and high performance computers. This problem is having a profound impact on Sandia's mission resulting in an initiative being instituted to address this problem. The initiative is intended to provide an integrated computing environment that delivers collective tools to designers, analysts, and others for better, faster, and cheaper management of the stockpile in the nuclear weapon complex. Some tools currently being used are critical to the success of Sandia's mission, while other tools are being created to address the initiative. Many existing tools do not interoperate for reasons including programming languages, poor design, operating system platforms, or program approaches (high performance and parallel computing versus distributed computing). Completing this initiative will require an infrastructure that overcomes the software integration barriers among these heterogeneous tools. The meta-component architecture (Component Mill) is being used to realize Sandia's software interoperability initiative. In general, the Component Mill is also applicable to industry, especially in the areas of 1) exposing existing data and systems for reuse with other heterogeneous software, 2) having a component-based development environment that leads to better and cheaper software, or 3) developing reusable software that can be executed on multiple component technologies.

Meta-components, the abstract encapsulation of components, are core to the Component Mill to expose legacy data and systems for reuse and integration. These meta-components can in principle be integrated with or used by other (legacy and new) software regardless of the component technologies, development methodologies, execution platforms, or programming languages. In addition to meta-components, the Component Mill uses some key concepts (design patterns, distribution, and high performance and parallel programming approaches) to support these capabilities. The meta-component architecture also provides an environment for creation of new components from scratch or through composition of components introduced into the environment or exposed data and systems. These components are in turn reusable for creation of other components.

This paper introduces the meta-component architecture. It starts with an examination of work related to reuse based on components and then shows how the Component Mill can be used to fill the hole in the tapestry of component-based reuse and integration solutions. The paper proceeds to describe the Component Mill approach (Section 3) and then presents the meta-component architecture (Section 4). Then, the application of the architecture to Sandia's initiative is discussed (Section 5). The paper concludes with plan to complete and validate the architecture (Section 6).

2. Related Work

Choose one from column A, one from column B, and one from column C; presto, you have an application. Why should software development not be this easy? After all, this is how the personal computers are constructed, by selecting off-the-shelf components. It is this hope of building software programs from pre-fabricated components that has lured the software community towards component software. The benefits of reusability and interoperability have driven both the research and commercial software communities to develop a component-based approach to software development. Today, there are many solutions for supporting component-based software development, ranging from research-based (Common Component Architecture [2], Component Software Engineering [7]) to industry-based (JavaBeans [12], Enterprise JavaBeans [12], DCOM [1], COM [3], ActiveX [4]) to consortium-based (SCIPIO [11], OpenDoc [9], CORBA Components [10]). The ones with the most promise and some industry momentum are Enterprise JavaBeans, CORBA Components, and COM/ActiveX. The following sections provide a quick glance into these technologies and discuss their limitations while indicating how the Component Mill addresses these constraints.

2.1. Enterprise JavaBeans

Enterprise JavaBeans (EJB) [12] is Java's new component architecture for the development and deployment of reusable Java server components. EJB is governed by the *Enterprise JavaBeans Specification*, which defines multiple roles in the EJB environment, describes how EJB interoperates with clients and existing systems, indicates how EJB is compatible with CORBA, and defines the responsibilities for other components in the system. In particular, an Enterprise JavaBean is a component (ebean) that executes in an EJB container that runs within the EJB server. A component is a Java class that implements the business logic. An EJB container hosts ebeans and makes the required services (i.e., transaction, security, persistence, etc.) available to ebeans through the interface defined in the specification, thus, freeing the component to concentrate on the business logic. An EJB server is the high-level process or application that manages EJB containers and provides access to system services. An application client accesses enterprise beans executing in an EJB container through the ebean's home and remote interfaces. The home interface lists the available factory methods for locating, creating, and removing instances of the enterprise bean, where the remote interface lists the business methods defined in the enterprise bean.

EJB is one of the first major component architectures with industry support. In a year since its announcement, there have been numerous vendors supporting the various aspects of the specification, ranging from tools to support development of enterprise beans (e.g., VisualCafe and VisualAge) to EJB servers (e.g., Iona, Inprise, and Weblogics). One major advantage is the specification of well-defined interfaces for the definition of components and deployment environments. This results in an architecture that supports component creation through composition, ease of use for component developers, and reusability. However, there are limitations that may not make EJB the component architecture of choice. One of the most critical requirements of a component is an independent unit of software that can seamlessly interoperate with other components regardless of the developing language, platform, developing methodology, and program types. Currently, EJB must be developed in Java, but this restriction may be removed when Sun has developed the Java to IDL mapping. However, this approach adds an extra layer to the component model, which may not be desirable. Another disadvantage of EJB is the stringent concurrency requirements. Applications assembled from enterprise beans and executed in EJB containers are restricted to one process and a single thread. These limitations make EJB a less than ideal solution for addressing Sandia's initiative of providing an integrated computing environment of heterogeneous tools and supporting component-based software development.

2.2. CORBA Components

OMG is working on extending the OMA platform to include an infrastructure for programmers to assemble software from off-the-shelf software components into sophisticated, distributed applications. This infrastructure is known as CORBA Components (CCM) [10]. Changes to the OMA, as well as additional features are required to support a component-based development model. As of the March 1999 revised submission to the CORBA Components RFP, CCM required an extension to the IDL grammar to support new meta-types and grammar. It also called for the addition of a component model, a component implementation framework, a container programming model, packaging and deployment, a component meta-model, and a mapping to Enterprise JavaBeans. CORBA component development begins by specifying the business logic and any external interfaces using the extended IDL (called CIDL). The business logic implementation uses the component implementation framework as the programming environment. Once the code is completed, the CIDL compiler generates the appropriate supporting infrastructure, which is used as input to create the package for distribution. Similar to Enterprise JavaBeans, CORBA components are executed in a container that uses CORBA services to support the components.

As specified, CCM is a component model that supports component composition, programming language and operating system independence, and interoperates with Enterprise JavaBeans. A disadvantage is that CCM requires IIOP as the single standard for a data transfer prototype. From the specification perspective, CCM addresses our requirements for developing and deploying components; however, it does not address component-based high performance computing. In particular, high performance computing components require knowledge of the target component locations. In addition, the connection among high performance components must be very efficient (e.g., a direct procedure call). CCM is built upon CORBA distribution mechanism, which is location transparent and connection speed is not a priority. This contrasts with high performance components where locality and speed are critical. The Component Mill fills this hole in the spectrum of component architectures by supporting both distributed and high performance components, as well as, the interaction among these types of components.

2.3. DCOM & ActiveX

Distributed Component Object Model (DCOM) [1] is Microsoft's integration infrastructure solution for implementing components that reside on different hosts. ActiveX [4] defines a set of services for supporting component documents. DCOM and ActiveX only support component development on Windows (effectively). Furthermore, it does not support programming language interoperability. ActiveX is not a component architecture for general-purpose use, but is designed for the sharing of document components. Thus, ActiveX using DCOM is not the solution needed to address Sandia's initiative due largely to the strict Windows requirements. The Component Mill architecture provides the mechanism to assist in removing these platform restrictions.

2.4. Common Component Architecture

Common Component Architecture (CCA) [2] is a developing component architecture specification that supports high performance computing, which targets network of workstations, distributed memory multiprocessors, clusters of symmetric multiprocessors, and remote resources. The specification is being developed by a group of representatives from DOE laboratories and academia with the aim of defining a foundation for definition of standardized sets of domain-specific component interfaces and for the interoperability among frameworks and toolkits developed by different teams from different institutions.

The CCA working group has defined an overall architecture for high performance components. In particular, the specification specializes the interface definition languages (IDL) to include grammar for specifying attributes specific to scientific computing software programs (e.g., matrix). CCA components use *ports* to support surface interfaces that allow fast and collective interconnects among components. These components are deployable into any CCA compliant framework. Currently, the CCA working group is prototyping the architecture and learning from the prototypes to improve the architecture.

The CCA is providing a mechanism for reusing high performance computing software. However, the CCA does not address component-based software development in an environment where applications execute (for the most part) on a single processor, as well as where the location of the components that makeup the application can be located anywhere (location transparent). Since CCA provides a solution for high performance computing, the Component Mill adopts this work as a baseline for addressing massively parallel and high performance components.

2.5. Summary and Opportunities for the Component Mill

The various component-based methods have been used to develop a component model to create components. However, these improvements do not directly address integration of different models and heterogeneous software, thus providing few opportunities for software developed based on reuse. This requires an architecture and sound methodology devoted to these principles.

While different component architectures have incompatible languages, domains of application, and functionality, most specify well-known and overlapping design patterns for their mechanisms of composition. Moreover, most implementations of various component architectures can *support* many mechanisms for composition even if a mechanism is not *admitted* by the component model specification.

Here we propose just such a system that we call the Component Mill. The Component Mill architecture seeks to create an infrastructure for integrating heterogeneous software and supports component-based software development. An abstract component (meta-component) is the underlying foundation of this architecture expressing the design patterns for composition of which its component model is capable. The following section describes the specifics of this architecture and explores its implications for interoperability.

3. Component Mill Approach

The basic premise of the Component Mill is to provide an architecture that addresses the integration of heterogeneous software (existing and new data and systems) and supports a development environment based on reuse. One option to achieving interoperability is to transform heterogeneous software into an agreed upon format producing software that interoperates in that common domain. Recognizing that component composition relies on implementation of particular interfaces, a second option is to adopt the interfaces of heterogeneous software into a common, interoperable ground, leaving the implementations in their original language, domain, and bindings. A disadvantage of the former method is that existing software must be re-written using the new format. This can be an expensive and time consuming approach, especially in view of the fact that the common format chosen for interoperability may itself become obsolete. The latter method, in general, does not change the intrinsic makeup of the existing software. Software communication interfaces (functionalities) are adopted, and are exposed for reuse. The exposed capabilities (services) are packaged into components that are composable with other components, supporting integration and reuse. The Component Mill supports both approaches (re-engineering and adoption) of exposing existing software. Figure 1 depicts the Component Mill's elements and their relationships to support re-engineering and adoption techniques for reuse.

As depicted in Figure 1, the Component Mill uses the adapter design pattern (adapter1, adapter2, adapterN) to expose existing software (Existing Application, Legacy Data, and Existing High Performance Application) into services (ser1, ser2, and serN) that are used in the creation of components (com1 and com2). The Component Mill also supports component construction from scratch (com3) or through composition of existing components (com4). Component creation is supported by the Component Constructor (architectural element of Component Mill).

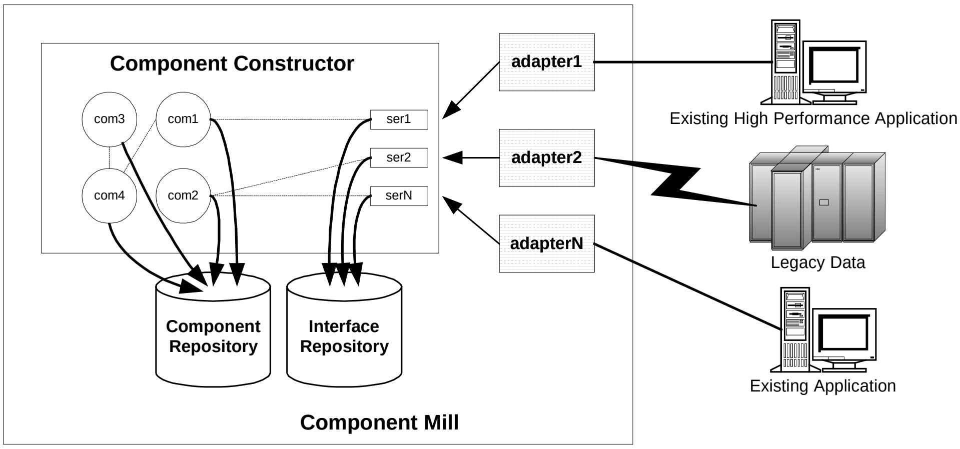

Figure 1. Exposing Existing Software for Integration and Reuse

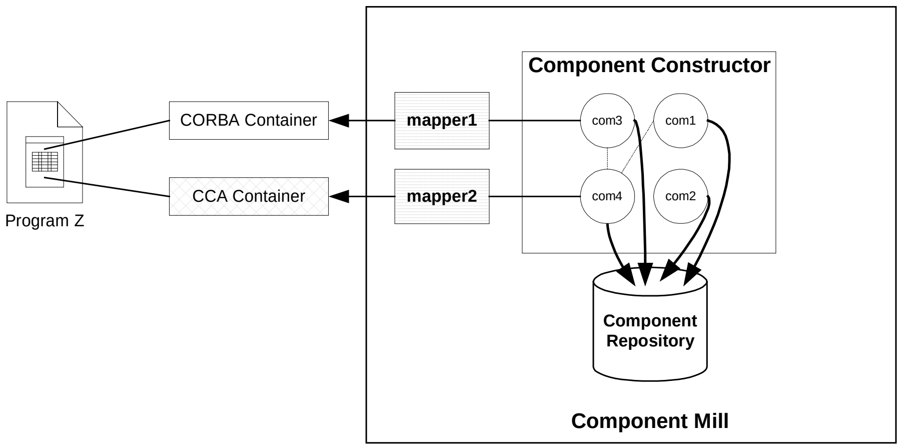

Figure 2. Use of Meta-Components To Support Technology Independence

The exposed services and components are stored in repositories (Interface Repository and Component Repository, respectively) for continuous reuse. These components (com1, com2, com3, and com4) are reusable to create new software and the components are composable for integrating existing software. Thus, the Component Mill supports software integration using the adapter design pattern, the concept of components, and persistence storage.

In addition, the Component Mill architecture follows the same approach as CORBA Components and Enterprise JavaBeans in separating components (business logic) from execution environments and common services (containers). Components created by the Component Constructor are free to execute in any component-based runtime environment (e.g., Enterprise JavaBeans, CORBA components, etc.), thus, providing technology and platform independence. Figure 2 illustrates elements of the Component Mill that support this capability.

As shown in Figure 2, the Component Mill specifies a mechanism for bridging between the Component Mill components and the execution environment of choice. For instance, deploying com3 in CORBA Container only requires a representation of the Component Mill components to the CORBA Container mapper (mapper1). Thus, deploying components in a different execution environment is a matter of supplying a mapper for that particular component technology. The mapper bridges between exposed service interfaces or created meta-component interfaces to the particular component technology, thus, it is a shadow mapper. A major advantage of this approach is the isolation of the infrastructure from the turmoil of component maturity. Also, this makes the environment realized by the Component Mill easier to maintain and improve. A realization of the Component Mill does not have to stake the future on one component technology; it can hedge its bets. In this architecture, the components are meta-components and can be mapped to any component technology as long as the mapper exists. Whether or not a mapper exists depends on a number of factors and is discussed in more detail in Section 4. Another benefit of this approach is that components are "thin" components that contain only its business logic and the necessary surface interfaces for external interactions and introspection. The executing environment or the mapper represents all the necessary mechanics and services. To summarize, the Component Mill is an architecture that supports integration of heterogeneous software and provides an environment for software development based on reuse. With the same importance, Component Mill components can be used with any component-based technology. A more in-depth description of the Component Mill is providing in the next section.

4. Component Mill Architecture

As discussed in Section 3, the Component Mill is an architecture that supports integration of heterogeneous software and provides an environment for software development based on reuse. In addition, Component Mill components are executable with any component technology, providing a technology independent architecture. Figure 3 depicts the architectural elements of the Component Mill and the relationships among these elements.

Figure 3 shows that the core of the Component Mill architecture is the *Meta-Component Model*. This model is a formal abstraction of software components to a point that it is independent of any component technologies based component representation. The *Component Constructor* uses the meta-component model to create components. Components are defined as being independent, deployable units of software that consist of a collection of standard interfaces exported to the external environment (applications or components). In addition, components must be composable with other components through their interfaces. The Component Constructor creates components from scratch, through composition of other components or exposed services. The *Component Adapter* is a specification and interface implementation for adopting existing software for reuse. There is an adapter for each exposed capability. This software may or may not be created from an existing component model. If the software is not already a component, the adapter makes it one. All adapters are stored in the *Interface Repository* and make the exposed services visible to the Component Constructor. Analogously, all created components are stored in the *Component Repository*, and the Component Constructor uses the Component Repository to determine which components are available for composition to create a new component. The Component Constructor also uses the *Component Locator* to discover components that are managed by other federated Component Constructors, where the components are executable in any component technology. This is realized using the *Component Generator*, which is a specification and interface implementation for mapping the Component Mill component representation to the particular component technology. In summary, these architectural elements (Meta-Component Model, Component Constructor, Component Adapter, Interface Repository, Component Repository, and Component Generator) together with their interactions are the Component Mill. Together, these elements support the goal of software integration and reuse. The remainder of this section provides a detailed description of these elements.

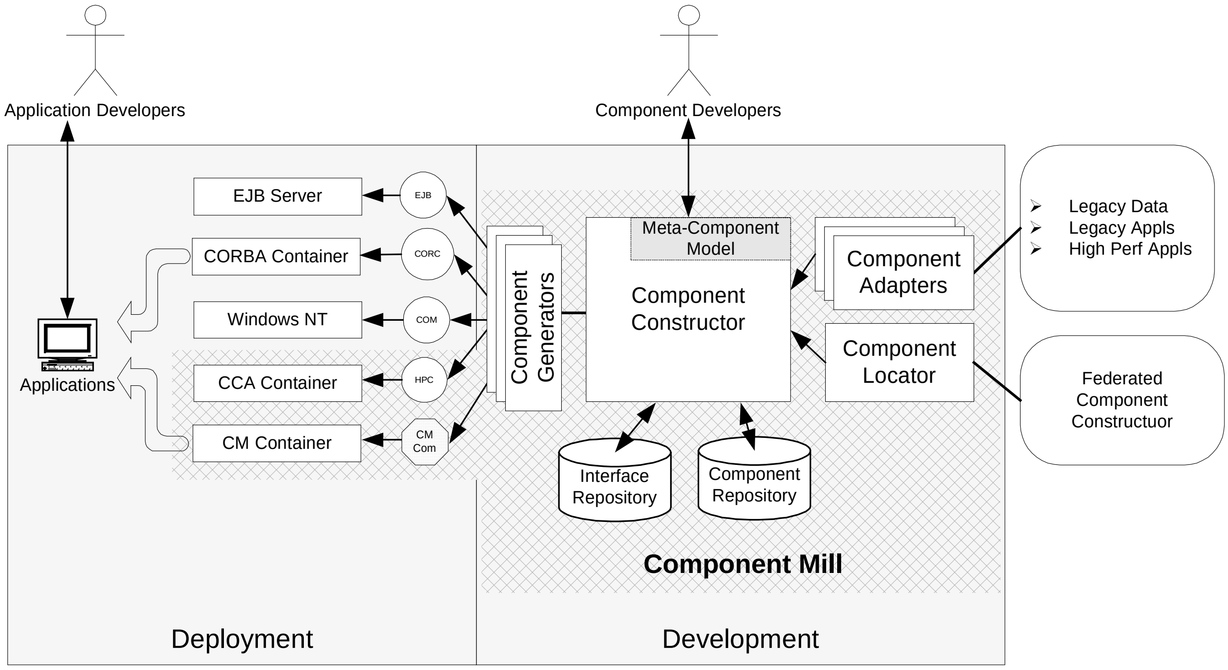

Figure 3. Component Mill

Though the Component Mill *can* map from one component model to another using the meta-component representation of composition design patterns, this begs the question of compatibility between composition patterns. In other words, regardless of the power of the concept, the Component Mill cannot express a composition mechanism that a particular component model cannot support. This is less restrictive than it might at first appear however. For example, Java Beans has exactly one mechanism for composition: a Notifier/Listener design pattern. At first glance it would appear that a CORBA3 component needing a Facet/Receptical (aka Uses/Provides) design pattern would be fundamentally incompatible. The Notifier/Listener pattern specification in Java Beans is too weak to support the Facet/Receptical pattern in CORBA3, though the converse is not true. While the JavaBeans *specification* is indeed too weak to support this pattern, Java is not. However, the Component Mill can map the Facet/Receptical design pattern to a particular representation in Java not covered by the Beans specification, but *supported* by the language in which all Beans are written: Java. By fashioning the Component Generator element to always do this Facet/Receptical translation consistently, CORBA3 components will still interoperate with CORBA3 components, even when mapped to a JavaBeans world.

The Component Mill only requires that a particular component system (EJB, JavaBeans, CORBA3) supports a particular design pattern and does not rely on the component system's specification to have included that pattern as a composition mechanism. This relaxes the design space considerably and effectively permits almost any full-featured component system to participate in the Component Mill architecture.

4.1. Meta-Component Model

The Component Mill, using the Component Constructor, supports creation of components from scratch, through composition of existing components or exposed services. The Component Constructor uses some formal abstraction representation to capture and create components. The representation must capture the semantics of the component surface interfaces without depending on any component technology based representation. More importantly, the representation must be sufficiently rich such that the component interfaces can be mapped to any component technology representation of a component, hence "meta" components. The Meta-Component Model describes this formal representation. Key parts to the model are: 1) the component identity, 2) the component standard external interfaces, 3) the component factory, and 4) the component definitions. Specifically, the Meta-Component Model uses ports to represent the component external interfaces. The Meta-Component Model supports four basic ports: Facet, Connector, Events, and Inspector. Facet is an interface that exposes a set of business logic to the external environment and allows clients or applications to interact with the component. A component may have several facets, each representing a different user perspective of the component. The Connector is an interface that provides interactions between components. This interface supports composition of components to create new components. In the default case, this port is implemented as a direct connect port, which implies that a connection between two components must be as efficient as a direct method call. This is required to support high performance component connections. The Connector ports are specialized into distributed connector ports to support remotely distributed connections between components. Events are ports that emit particular events to, or receive specific events from a connected port. Lastly, an inspector is a port that dynamically reads the external environment to determine its capabilities based on this information. The Component Constructor uses the Meta-Component Model to define components.

4.2. Component Constructor

The Component Constructor is core to the architecture. Using the Meta-Component Model, it provides the infrastructure for construction and creation of software components from scratch, through composition of existing components or exposed services. After suitable pre-configuration of the legacy software, the Constructor provides the users the capability to automatically browse legacy data, legacy services, and existing components to create new components dynamically. That is, amorphous legacy software is converted to a component by adapting facets to the original software. The Constructor uses the Interface Repository to retrieve exposed services. Similarly, the Constructor depends on the Component Locator to retrieve existing components from the local or remote Component Repository. The Constructor uses the Component Generator to export software components stored in the Component Repository into various component forms (i.e., Enterprise JavaBeans, CORBA Components, COM, etc.). Once components are exported, they can be deployed in the supporting server. For example, Enterprise JavaBeans components are deployable into EJB Container and are supported by the EJB Server.

The Component Constructor is part of a decentralized federation of Component Constructors. This implies that each Component Constructor manages its own set of federated Component Constructors. To be part of a federation, the Component Constructor must be able to import from at least one other Component Constructor in the federation. The Component Constructor that exports to a remote Component Constructor has an export contract with the remote Component Construct. The remote contract is an agreement between an importing and exporting Component Constructors.

4.3. Component Adapter

The Component Mill architecture supports both re-engineering and adoption approaches to integration. Re-engineering integration is supported through the re-construction of existing software into components using the Component Constructor. Adoption integration is realized with the Component Adapter. A Component Adapter is a reification of the object adapter [8] that provides the logic for mapping legacy software into a representative that is understandable by the Component Constructor. There must be an adapter for each exposing capability. All adapters are registered with the Component Mill and are stored in the Interface Repository, which is used by the Component Constructor to make the exposed services visible to the users. Specifically, each adapter must consist of multiple elements. The first and required element is the IDL that describes service signatures that are either retrieved legacy data or maps to one or multiple combinations of legacy system services. Another item is the necessary logic to export the mapping of the IDL to the legacy data and systems. An optional item is the logic to support distribution (i.e., stub and/or tie files) if the legacy data or systems are located remotely and the package is described using XML. The Component Mill architecture supports both static and dynamic deployment of the adapters. The static deployment allows the adapters to be loaded into the Interface Repository during the Component Constructor activation process. Static deployment requires that the location of the adapter be specified to the Component Constructor.

4.4. Repositories

The Component Constructor uses the Component Repository to store the components for tracking and for Component Constructor federation activities. Similarly, the Interface Repository tracks the component adapters that are adapted by the Component Constructor. In particular, they record the adoption interfaces and the associated package information. The tracked interfaces are used by the Component Constructor to provide the users with exposure to legacy data and systems.

4.5. Component Generator

The Component Constructor uses the Component Generator to export components stored in the Component Repository into representations supported by the various existing component technologies (i.e., Enterprise JavaBeans, CORBA Components, COM, etc.). If the components are to be deployed on the Component Mill supported containers (CM and CCA Containers) then either none or very little mapping is required. The Component Generator is an interface specification that allows Component Mill components to be deployed on any component technology.

The Component Generator uses the adapter pattern to map the Component Mill component representation into a particular component model used by the particular component technology. During the component generation phase and if the exporting components contain exposed services, the Component Generator automatically produces the appropriate bindings to the legacy data and systems using the data provided by its adapter. Similarly, if the generated component consists of other components, the Component Generator generates the necessary binding to the sub-components regardless of the location. In addition, each extracted component has built-in authenticating and authorizing information to control access to the component.

4.6. Component Locator

The Component Constructor uses the Component Locator to retrieve components from the local Component Repository and any federated Component Constructors within the search path. One possible implementation is to use intelligent agents to actively search the Component Mill federation space to provide the current Component Constructor with information about remote components. Another option is to the conventional federation approach to determine remote components.

5. Component Mill Application

Sandia's initiative to provide an integrating computing environment to the nuclear weapon designers and analysts is being fulfilled with the Component Mill, which serves as a blueprint for integrating the heterogeneous software and providing an environment for reusing these components. The realization is planned in stages, with the first stage being to develop the Meta-Component Model for component abstraction and creation. The second stage is to adopt legacy data and systems to expose their services for reuse, where this adoption is governed by the Component Adapter interface specification and recommended adoption techniques. The third stage requires development of a mapper using the Component Generator to a particular component technology. At this point, with some manual manipulation, the instantiation of the Component Mill will be realized. The fourth stage involves implementing the Component Constructor to automate component creation and adoption of existing software and mapping to particular technologies. The fifth stage introduces the Component and Interface Repositories to store components and exposed services. The last stage adds the Component Locator for broader reuse. Currently, Sandia is developing stages one and two concurrently with plans to complete the instantiation of the architecture within the next year

Sandia is also planning to deploy the realized infrastructure in increments. The first phase is to create reusable components either through adoption of existing capabilities or by creating new components. In the second phase, Sandia's development process will be modified to include reusing components and integrating existing capabilities. The benefits (software reuse and integration) of the Component Mill will be realized with completion of the second phase.

In the first phase of deployment, Sandia is scheduling to adopt the various existing software into components, as well as to create components from scratch as needed. An example of reusable capability is the ConversionCentral service in PRE. PRE is a middleware that uses CORBA to provide an easier and more restrictive environment for Sandia's application developers to take advantage of distribution. ConversionCentral is one of PRE's common services that supports conversion of one data format (GIF) to another format (BMP). Some software products need the capability of the ConversionCentral but do not want to inherit the entire PRE infrastructure. PRE ConversionCentral service can be made available to these products by providing *ConversionCentralAdapter* to the Component Constructor. The *ConversionCentralAdapter* is an instantiation of the Component Adapter specification. Once the service is exposed to the Component Constructor, it can be used to create the *ConversionCentral* component, which is stored in the Component Repository. An application can then use the *ConversionCentral* component regardless of its execution environment. For instance, if *ApplicationX* is an EJB program, then *ConversionCentral* is converted to enterprise bean using the *CMtoEJBGenerator*, which is created using the Component Generator specification. Thus, with the Component Mill, the PRE ConversionCentral service is reusable regardless of the executing technology without needing to maintain multiple versions of the ConversionCentral component or changing the original capability.

6. Conclusions and Future Work

The Component Mill is an architecture based on meta-components to create an environment for software integration and reuse. Core to this architecture is the Component Constructor, which uses the Meta-Component Model to create components from scratch, through composition of existing components or exposed services. Legacy data and systems are exposed using the Component Adapter, which uses the concept of adapter design patterns. The adapters are stored in the Interface Repository for retrieval by the Component Constructor. Analogously, all components created by the Component Constructor are stored in the Component Repository for reuse. The Component Constructor relies on the Component Locator to determine availability of components for reuse. The Component Locator searches its local repository and migrates to remote repositories that are within its visibility. Components are deployable in any component technologies supported with the Component Generator. If existing technologies are not sufficient, then the Component Mill specifies two containers tailored to provide the necessary services that addresses Sandia's needs. One of the containers is used to host massively parallel and high performance components (CCA Container) while the remaining components are executable in the CM Container. All these architectural elements interact in a specific manner to support the Component Mill architecture, which is used for software integration and reuse.

The architecture has positive implications for both Sandia and industry. For industry, the Component Mill is useful to expose legacy data or applications that are critical to the success of companies that have reached their limits in both scalability and modification improvements. At Sandia, the Component Mill is being used as a blueprint for realizing the vision of providing an integrated computing environment to the designers, analysts, and support personnel for stockpile stewardship of the nuclear weapon complex. Within the next year, Sandia is planning to provide an instantiation of the Component Mill through incremental development. Validation will be done at each increment, where the components and each of the architectural elements will have built-in mechanisms for validation. Validation criteria includes the numbers of reuses per component, number of heterogeneous software interoperating, and the scalability of the integration.

After the first instantiation of the Component Mill, lessons learned will be incorporated in the architecture. The first instantiation will not include the Component Locator or a federation of the Component Constructor. These will be added in the second year. The first implementation of the Component Locator will use simple federation policies. In concurrence with the first instantiation development, intelligent agents are being designed and implemented to enhance the federation policies. Other future work will include automation of the component-based software engineering process and development of intelligent agents to more effectively support maintenance and management of the environment.

7. References

[1] R. Abernety, R. Morin, and J. Chahin, *COM/DCOM Unleashed*, Sams Publishing, 1999.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, Toward a Common Component Architecture for High-Performance Scientific Computing, In proceedings of *Super Computing*, 1999.

[3] D. Box, *Essential COM*, Addison-Wesley, Reading, MA, 1998.

[4] Chappell, D. *Understanding ActiveX and OLE*, Microsoft Press, 1997.

[5] B.J. Cox, There is a Silver Bullet: Reusable Components, *Byte*, 15, 10, (October), 209-218, 1990.

[6] B.J. Cox, Planning the Software Industrial Revolution, *IEEE Software*, 7,6, (November), 25-33, 1990.

[7] Developing a Handbook for Component-Based Software Engineering, *Proceedings of the International Workshop on Component-Based Software Engineering* held in conjunction with *21st International Conference on Software Engineering (ICSE)*, Los Angelas, CA, USA, May 17-18, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[9] A. Meadow and J. Feiler, *Essential OpenDoc: Cross-Platform Development for OS/2, Macintosh, and Windows Programmers*, Addison-Wesley, Reading, MA, 1996.

[10] Object Management Group, *CORBA Components*, OMG TC Document orbos/99-02-05, March 1999.

[11] SCIPIO Consortium, *SCIPIO*, (available at http://www.scipio.org/).

[12] Sun Microsystems, Enterprise JavaBeans Specification, Version 1.1, August 9, 1999, (available at http://www.javasoft.com/products/ejb/docs.html).

[13] Sun Microsystems, JavaBeans Specification, Version 1.01, July, 1997, (available at http://www.java.sun.com/beans/docs/spec.html).