Approaches to Constructive Interoperability

Grace A. Lewis
Lutz Wrage

December 2004

Carnegie Mellon
Software Engineering Institute

Approaches to Constructive Interoperability

Grace A. Lewis
Lutz Wrage

September 2004

Integration of Software-Intensive Systems (ISIS) Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

Table of Contents

List of Figures

List of Tables

Abstract

Interoperability between systems requires the capability for users to exchange information (syntactic interoperability) and a common understanding of its meaning or how to act upon it (semantic interoperability). This report will discuss several current approaches to constructing systems of systems that have interoperability requirements, with respect to syntactic and semantic interoperability. The areas examined include Model-Driven Architecture, Service-Oriented Architecture, Web services, Open Grid Services Architecture, and Component Frameworks. These initial discussions assume that the interoperating systems agree on a common approach. Reaching an agreement can be challenging, especially when legacy systems are involved. Techniques and recommendations for reaching an agreement between systems that use differing technologies are also briefly explored.

1 Introduction

Interoperability is much more than the capability for exchanging data between systems. Also required is a shared understanding of that information and how to act upon it.

> *Interoperability is the ability of a collection of communicating entities to (a) share specified information and (b) operate on that information according to an agreed operational semantics [Brownsword 04].*

The ability to exchange information is *syntactic interoperability* and the ability to operate on that information according to agreed-upon semantics is *semantic interoperability*; both are needed to solve the interoperability problem.

Organizations trying to achieve system of systems interoperability usually concentrate on syntactic interoperability, via techniques such as common messaging standards and interchange formats. For example, it is commonly assumed that if systems can exchange XML (eXtensible Markup Language) files and no errors occur during assembly and parsing of the files, then interoperability has been achieved. But this approach leaves out the most important problems, which deal with semantic interoperability: What are systems supposed to do with the XML files once they are received? How does a system developer obtain the information to interpret the exact meaning of each of the data elements contained in the XML file? So ultimately, even perfect syntactic interoperability is insufficient.

To achieve semantic interoperability, system developers usually go through a laborious and time-consuming process of engineering every inter- and intra-system information exchange a priori. This generally results in system interfaces that are fragile and relatively inflexible. A change in something as simple as a single field within a message may require a significant reengineering effort to numerous systems in order to maintain interoperability. Unfortunately, this approach is not adequate to respond to the increased demand for distributed, dynamic, composable systems that require (1) automated processes for locating services and (2) negotiating appropriate service contracts in the absence of complete information.

Figure 1 shows the System of Systems Interoperability (SoSI) Model, developed by the Carnegie Mellon® Software Engineering Institute (SEI) as part of an independent research and development project. The SoSI Model presents three types of interoperability [Morris 04].

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office.

1. Programmatic: interoperability between different program offices or organizations tasked with the development of a system

2. Constructive: interoperability between the organizations that are responsible for the construction (and maintenance) of a system

3. Operational: interoperability between the fielded systems
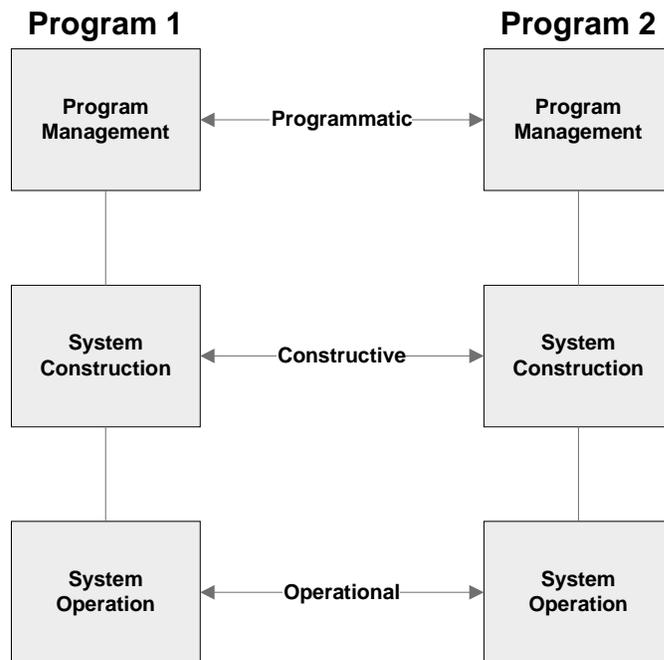
Figure 1:   Different Types of Interoperability

The focus of this report is constructive interoperability. Constructive interoperability can be defined as the process by which multiple system development entities interact, such that resultant constructed systems can interoperate. Technology as well as management activities take place in the constructive interoperability process. Examples of technology activities are technology selection, model sharing, and system construction. Management activities include the collaborative interactions between system development entities such as project management, contract management, resource allocation, and configuration control.

From a technology perspective, there are many current approaches to constructing systems with interoperability requirements. Each has particular advantages and disadvantages with respect to interoperability, and each works well in some circumstances but not others. These approaches include

- Model-Driven Architecture (MDA)
- Service-Oriented Architecture (SOA)
- Web services

- Open Grid Services Architecture (OGSA)

- Components Frameworks

This is the first in a series of reports covering constructive interoperability, both at the syntactic and semantic level. It will focus exclusively on the technology aspects of constructive interoperability. It will not cover the management aspects of constructive interoperability, which are mostly driven by activities in programmatic (organizational) interoperability. Sections 2 to 6 of this report will discuss each of the above approaches with respect to syntactic and semantic interoperability. Section 7 will illustrate the problem of constructive interoperability. The details of future work and upcoming reports in the area of constructive interoperability will be included in Section 8.

2 Model-Driven Architecture (MDA)

The goal of the MDA is to make it easier for software developers to separate business and application logic from underlying execution platform technology. The major benefit of this approach is that it raises the level of abstraction in software development. Instead of writing platform-specific code in some high-level language, software developers focus on developing models that are specific to the application domain but independent of the platform. Although MDA includes the term "architecture," this does not mean that MDA defines a particular software architecture or an architectural style. MDA is a broad conceptual framework that describes an overall approach to software development.

The Object Management Group (OMG) has developed the fundamental concepts of Model-Driven Architecture and, at the time of writing, working groups are defining new standards that are necessary to realize the MDA concepts in practice. While the MDA concept is a vendor- and technology-neutral approach [OMG 03], MDA is compatible with

- established OMG standards such as
  - CORBA (Common Object Request Broker Architecture)
  - UML (Unified Modeling Language)
  - MOF (MetaObject Facility)
  - XMI (XML Metadata Interchange)
- other industry standards such as Web services
- component frameworks such as
  - Sun's J2EE (Java 2 Platform, Enterprise Edition)
  - Microsoft's .NET

At the core of MDA lies the idea of describing business and application logic in a platform-independent model or in a set of related models and to utilize tools to generate all platform-specific implementation code from these models. In this way, all code that depends, for example, on the middleware, will be generated instead of written by hand as is usually the case today. Ideally, it should then no longer be necessary to involve middleware experts in the software development effort because all knowledge about the middleware-specific implementation details will be included in the MDA tools and code generators. Other expected benefits of the MDA approach for the development process include higher developer productivity, reuse of domain models and platform-independent models, and a more consistent development process. Applications that are developed using this approach are expected to be more portable and to interoperate better across platforms. It is important to note that these benefits

depend on the availability of MDA tools which are just emerging, so there exist little or no data to confirm or refute their actual delivery.

What distinguishes MDA from the current practice of model usage in software development is an emphasis on automatic model transformations. This emphasis extends to code generation because code can be viewed as a very detailed, executable model of a system. Future MDA tools will incorporate transformation capabilities where the transformations are described in a vendor-neutral manner based on OMG standards. This is in stark contrast to currently available tools where mechanisms for code generation are proprietary. The platform-independent models allow the developer to reuse the same model to generate implementations to run on various platforms. MDA tools also should have the capability to generate code to bridge different platforms. A simple example is a three-tiered Web application where each tier runs on its own platform (e.g., a relational database, a J2EE application server, and a Servlet engine). More complex situations would also involve different operating systems, different middleware, and so on. MDA tools should be able to generate code for various languages and platforms and also to generate code that integrates parts of an application into one coherent whole.

Models and Transformations

A model for MDA must be a formal model in the sense that it is described in a language with well-defined semantics so that an automated tool can process the model. Examples of acceptable models are UML class diagrams and state charts, entity-relationship diagrams, or even source code. Ad hoc box-and-line diagrams, on the other hand, do not qualify. Formal models make it possible to define transformations of models that can be executed automatically.

The OMG's MOF plays a prominent role in MDA as it provides a standard repository for models and other meta-data with standardized interfaces to access its content from CORBA or from a Java application. An MOF repository can contain models as well as models of models (metamodels). A metamodel essentially defines a language to describe models. There is, for example, a UML metamodel that defines UML in terms of MOF constructs. MOF metamodels themselves are described in a language that includes a subset of UML class diagrams plus the Object Constraint Language (OCL), so it should be fairly easy to use for developers who are already familiar with UML.

MDA prescribes the use of three kinds of models: Computation Independent Models (CIM), Platform Independent Models (PIM), and Platform Specific Models (PSM). A CIM or domain model highlights the environment and the requirements of a system. The structure and operation of a system are described in the PIM, independent of execution platform technology details. The details of how the system makes use of the platform are described in a PSM. In this chain of models, implementation code is another, very platform specific, PSM. Model transformations convert a PIM or PSM of a system to another model of the same system, for example, PIM to PSM, or PSM to implementation code. Transformations may use additional

information as indicated by the empty box in Figure 2. This additional information could specify, for instance, a particular architectural style or data access pattern to be used in the PSM [OMG 03].

Within MDA there are no objective criteria that determine when a model is platform independent or platform specific. This depends greatly on the viewpoint of the model developer so in practice there is a whole spectrum with PIM and PSM being the extreme cases.

Current MDA tools define model transformations in a vendor-specific manner, such that it is not possible to exchange transformations between tools from different vendors. There is ongoing work at the OMG to define a declarative transformation description language QVT (Queries, Views, and Transformations). QVT is a standardized metadata repository that will support vendor-neutral definition of model transformations. This work is still in its very early stages.
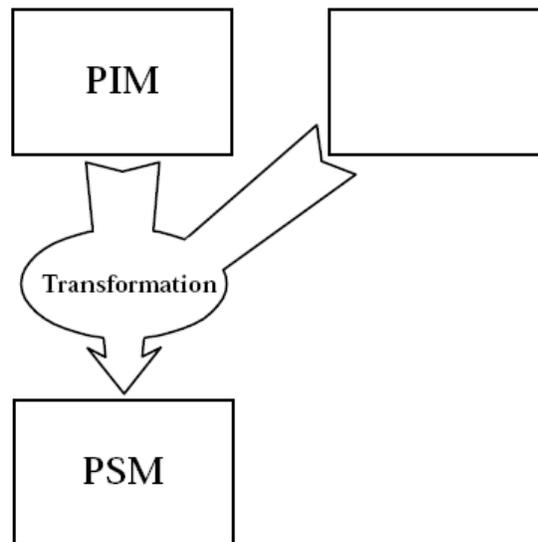
Figure 2:   Model Transformation

MDA Tools

At the time of this writing, there are many tools on the market that claim to support MDA, but this support can only be incomplete because parts of the MDA concept are not yet finalized. In addition, there is no agreed-upon specification of exactly what an MDA tool should incorporate and so there is no standard notion of MDA conformance of a tool. Tool vendors have to rely on their own interpretations of the MDA approach to make decisions about their tools' capabilities. The OMG site contains a list of companies that are committed to MDA and their products [OMG 04].

Most tools available so far seem to be designed towards generation for one execution platform only, mostly J2EE.

MDA and Interoperability

There are two major aspects of MDA that relate to interoperability.

1. Interoperability of applications across platforms:
   Application interoperability in an MDA context means that software applications can work together independent of the execution platform of each individual application. The mechanism to achieve this kind of interoperability is bridge code that an MDA tool generates based on information about the (a) interoperating parts in the model (b) target platforms, which is encoded in the models and in the transformation definitions. Bridge code enables syntactic interoperability. However, semantic interoperability is not necessarily guaranteed because current MOF-based metamodels cannot completely specify the execution semantics of models.

   An approach to achieving semantic interoperability would include a complete specification of data and operations on that data, which define both syntactic and semantic aspects. This information could be part of additional information for model transformations, as shown in Figure 2, or it could be stored as part of the model. For example, if two models specify a data element *currency,* the transformation would obtain all information related to the type *currency* from the data specification or model and generate the equivalent data type, documentation, and permitted operations on this data type. Current tools, however, do not support the definition of semantics at the required level of detail and, as a result, different tools may generate implementations with different semantics.

   Another aspect is the scope and completeness of models. In some cases it may be possible to create a model that is complete in the sense that a tool can generate all application code from the model. Other tools only allow specification of models that comprise only certain aspects of the system and require that developers implement some business logic directly in the implementation programming language. In this case, even if the same model is used to generate different parts of the application, interoperability cannot be guaranteed by the tool because programmers may base their implementations on conflicting assumptions.

2. Interoperability of MDA tools:
   This aspect relates to the degree to which an MDA tool environment is open and allows the developer to exchange models with other tools that may be provided by different vendors. So far, MDA provides the basic mechanisms for enabling such model exchange through XMI. XMI defines a way to represent metamodels and models as XML Schemas and XML documents. Other concerns that will gain relevance in the future are the exchange of graphical views of models and the exchange of model transformation definitions. For current tools, interoperability is limited to syntactic interoperability through

a common exchange format. The semantics of the exchanged information are often tool specific.

Many tools define model elements that are tool specific, and it may be very difficult to reconstruct these elements in another tool. XMI will support these elements syntactically without any problem but the receiving tool will not be able to process the information in a meaningful way.

3 Service-Oriented Architecture (SOA)

The simplest way to define a service-oriented architecture is as an architecture built around a collection of services with well-defined interfaces—similar to DCOM (Distributed Component Object Model) or Object Request Brokers (ORBs) based on the CORBA specification. A system or application is designed and implemented as a set of interactions among these services.

A service is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model [Brown 02]. Common communication models are

- Web services using Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL)
- message-oriented middleware (MOM) such as IBM Websphere MQ
- publish-subscribe system such as Java Messaging Service (JMS)

What makes SOA different from DCOM or CORBA are the words *discoverable* and *coarse-grained,* present in the previous definition of a service. Services need to be able to be discovered at both design time and run time, not only by unique identity but also by interface identity and by type of service. Services are also ideally coarse-grained, that is, they usually implement more functionality and operate on larger data sets, as compared to components in component-based design. A typical example of a service is a credit card validation service.

A service can be invoked in several ways, as shown in Figure 3 [Brown 02, ServiceArchitecture 04]. A service consumer can

1. directly invoke a service provider
2. use a directory service to find a service provider based on some criteria. The directory service returns the location of the service so the service consumer can invoke the service provider.
3. use a service broker to pass on its request to one or more directory services
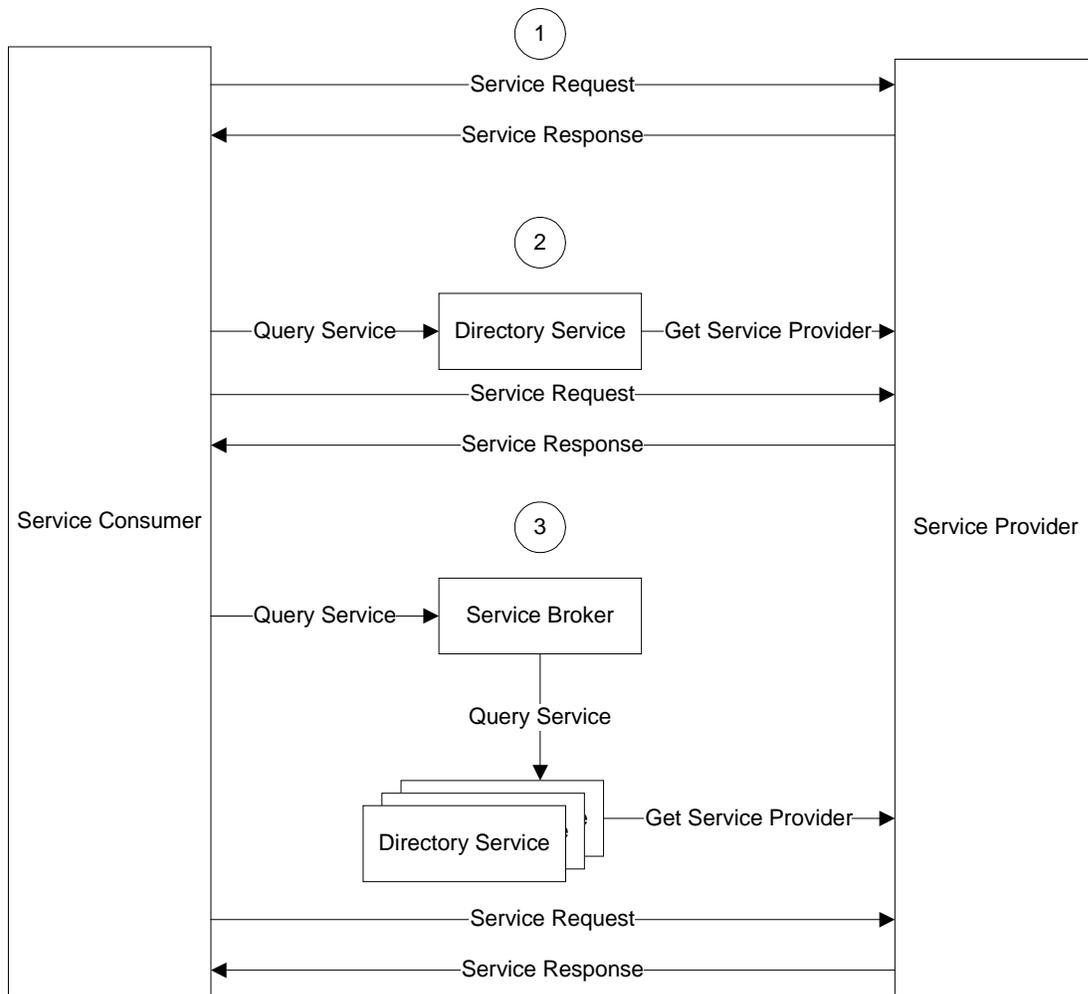
Figure 3: Forms of Service Invocation

Examples of service-oriented architectures are Web services using SOAP and UDDI (Universal Description, Discovery and Integration Service), HP's E-Speak [Karp 00], and Sun's Jini [Sun04b]. There is a more detailed discussion on Web services in Section 4.

SOA and Interoperability

In a service-oriented architecture, interoperability is simply defined as the ability of the service to be invoked by any potential client of the service [Stevens 03]. This definition of interoperability has a much narrower scope than the one being used in this report. Nonetheless, there are several attributes of an SOA that make this a possibility:

- Common payload and protocol: Each service provides an interface that is invoked through a payload format and protocol that is understood by all the potential clients of that service.[1]

---

[1]   Payload is the term used by most messaging technologies to refer to the actual data being exchanged.

- Published and discoverable interfaces: Each service has a published and discoverable interface that allows systems to search for services that are best suited for their purposes.

- Loose coupling: Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges.[2]

- Multiple communication interfaces: Services can implement separately defined communication interfaces. For example, a service could have a Web services adapter, an IIOP (Internet Inter-ORB Protocol) adapter, and an MQSeries adapter to serve clients of these three different types.

- Composability: Because services are coarse-grained reusable components that expose their functionality through a well-defined interface, systems can be built as a composition of services and evolve through the addition of new services.

From a syntactic point of view, service-oriented architecture is very promising. The challenge lies in determining the number of adapters to implement and determining the right granularity of service interfaces, because it is not always known how systems will use the services. It is important to keep in mind that services are executed across a network as an exchange of a service request and a service response. If service interfaces are too coarse-grained, clients will receive more data than they need in their response message. If service interfaces are too fine-grained, clients will have to make multiple trips to the service to get all the data they need.

From a semantic point of view, service-oriented architectures by themselves do not offer any guarantees. Semantic interoperability depends on how the interface to a service is described and how the meaning of the information is shared with potential clients of the service. There is a great amount of research being done in this area because this is the difficult problem: How to know exactly what a service offers? How to interact with this service? What quality of service (QoS) does it offer? Some of these questions will be covered in Section 4 on Web services.

---

[2]   Service orientation encourages loose coupling, but does not guarantee it. A loosely coupled architecture is good for systems that do not require near-real-time responses.

4 Web Services

In its simplest definition, a Web service is an instantiation of a Service-Oriented Architecture where all of the following apply.

- Service interfaces are described using Web Services Description Language (WSDW).

- Payload is transmitted using Simple Object Access Control over HTTP (Hypertext Transfer Protocol).

- Universal Description Discovery and Integration (UDDI) is used as the directory service.

Other combinations of technologies are possible, but this is the most common instantiation and the reason why the terms SOA and Web services are often used interchangeably.

The growing success of Web services is due to a number of factors, including those below.

- Systems can interact with one another dynamically via standard Internet technologies.

- Services are built once and reused many times.

- Services can be implemented in any programming language.

- Service consumers do not need to worry about firewalls because communication is carried over HTTP.

- Systems can advertise their capabilities for other systems to use. For example, Amazon Web Services allows systems to access catalog data, manage the shopping cart, and initiate the checkout process via Web services [Amazon 96].

- Standards such as BPEL4WS (Business Process Execution Language for Web Services), WS-Security, WS-Routing, WS-Transaction, WS-Coordination, and WSCL (Web Services Conversation Language) are working toward the automatic discovery and composition of Web services.

Web Services and Interoperability

The reason why many vendors and users associate Web services with interoperability is because interoperability is simply defined as the capability to implement a service in multiple programming languages and to communicate using well-known and platform-independent protocols and standards. This definition of interoperability, like the SOA definition, has a much narrower scope than the one being used in this report.

The Web Services Interoperability (WS-I) group is attempting to provide guidance on the usage of Web services standards. Established in early 2002, WS-I is an open industry effort chartered to promote Web services interoperability across platforms, applications, and programming languages. This organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services [WS-I 04]. WS-I also recently announced the availability of its tools for testing interoperability with the WS-I Basic Profile for use of Web services.

From a syntactic point of view, Web services are very promising and are experiencing tremendous growth because of their reliance on well-known standards and organizations like WS-I. The current challenge is that these standards are emerging and therefore there is still considerable room for different interpretations of the standards by parties implementing Web services. This is especially true of SOAP because of the available choices in formats, envelopes, and transport protocols (see Section 4.2).

From a semantic point of view, there are many limitations because Web services can currently only be discovered based on keywords. Therefore, the ability for run-time discovery, a requirement for automatic Web Service composition, is limited. The Semantic Web is a collaborative effort led by W3C with participation from many researchers and industrial partners who wish to tag information on the Web in such a way that it can be interpreted by software agents looking for specific types of information. The combination of Web services with the Semantic Web is called Semantic Web Services. A Semantic Web Service is a Web Service whose description is in a machine-understandable language with formal semantics. The idea is to be able to describe Web services in such a way that applications can automatically coordinate information exchanges and hence improve interoperability. The Semantic Web Services arm of the DAML (DARPA Agent Markup Language) program is developing a Web Service Ontology based on OWL (Web Ontology Language) called OWL-S (formerly DAML-S), as well as supporting tools and agent technology to enable automation of services on the Semantic Web [Sycara 03]. OWL is intended to be used when the information contained in documents must be processed by applications instead of humans [W3C 04b]. With ontologies such as OWL-S, or others described using OWL, there is a much greater chance of semantic interoperability, but these ontologies are still emerging and primarily being used in research environments.

The next subsections will briefly describe the technologies behind Web services from an interoperability perspective.

4.1   Web Services Description Language (WSDL)

WSDL is used to describe what a Web Service can do, where it resides, and how to invoke it. It is XML-based and supports simple and complex transactions defined by message exchange patterns [W3C 04a].

From an interoperability perspective, WSDL defines the interface to the Web Service. If interfaces are well defined, then the chances of interoperability increase. There are many vendors that are releasing WSDL interoperability tests against the WS-I Basic Profile. If there is conformance to the WS-I Basic Profile, there is an even better chance for interoperability. The message exchange patterns defined in Part 2 of the WSDL working draft are also a plus for interoperability because they contain pre-defined sequences of messages that make it easier to interact. Development tools such as Sun Java Studio, Cape Clear CapeStudio, and BEA Cajun automatically generate WSDL documents. Tools such as these promote interoperability because they avoid the errors that appear when developers try to create WSDL documents by hand. There are also WSDL repositories such as www.salcentral.com and www.xmethods.com that contain tested WSDL documents as well as tools. But regardless of all these advances, WSDL still does not address the semantic issues of interoperability mentioned earlier.

4.2   Simple Object Access Protocol (SOAP)

SOAP defines a framework to construct XML-based messages that can be used to exchange information between nodes in a decentralized, networked environment. SOAP messages are defined as XML Infosets. An XML Infoset is an abstract description of the contents of an XML document [W3C 03].
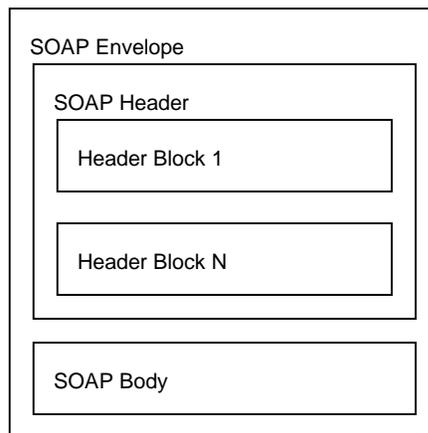
Figure 4:   SOAP Message

As shown in Figure 4, a SOAP message consists of header and body information. A SOAP message travels between SOAP nodes on a SOAP message path from an initial sender through one or more intermediate nodes to an ultimate receiver. Each node on the path may process the message in some way based on information in the header blocks. The message body is processed by the ultimate receiver. SOAP does not define how messages are transported between nodes and how they are routed, but relies on an underlying protocol for this. There is one standard protocol binding to HTTP, but other protocols such as e-mail could be used to convey SOAP messages.

While SOAP messages are inherently one-way, applications can build more complex message exchange patterns on top of them. Examples are request/response or remote procedure call.

From an interoperability perspective the following issues are relevant:

- translation between platform-dependent types and SOAP data types: There is no guarantee that the receiver implements a data type in a manner that is compatible with the sender; it might not implement that data type at all.

- semantics of conveyed information: This is outside the scope of SOAP as it defines only the message format and which node must or may process the message. The processing itself and the meaning of data contained in the message headers and body is application-dependent.

- SOAP protocol bindings: Different protocol bindings can implement different features. For example, HTTP implements a request/response pattern such that an application can make use of this feature when exchanging SOAP messages over HTTP. E-mail, as another example, only supports one-way messages and therefore an application that exchanges SOAP messages via e-mail must contain additional code that matches responses to requests.

There are groups interested in testing SOAP interoperability. SOAPBuilders, for example, is an open group of SOAP developers defining interoperability test suites that check custom data types for compatibility [Cohen 02].

4.3 Universal Description, Discovery and Integration Service (UDDI)

UDDI is an XML-based distributed directory that enables businesses to list themselves, as well as dynamically discover each other [OASIS 02]. Businesses register and categorize the Web services they offer and locate Web services they want to use. UDDI itself is a Web Service. The directory contains three types of information, similar to a phone book:

- white pages: contains basic information such as name, address, business description, and type of business

- yellow pages: follows a categorization based on U.S. government and United Nations standard industry codes

- green pages: contains technical information about the services that receive exposure through the business directory that will help a client connect to the service

From an interoperability perspective, the goal behind UDDI is to allow businesses to dynamically discover each other. Only business services are described in the registry. UDDI works in two ways: (1) a developer queries the registry, obtains information on how to access the service, and writes a client to access the service, or (2) a client uses the registry as a Nam-

ing Service, obtains the endpoints[3] for the desired service, and binds to one of the returned URLs dynamically. The second method is more aligned to the UDDI goals but currently the first method is the most used because the algorithms on how to decide which is the best service when more than one URL is returned are still not very reliable and usually require human intervention. The first method works very well when the provider of the service is known, but the problem is that it is static and will work as long as the provider does not change. To help in this matter, Version 3 of UDDI provides Subscriptions and Notifications that allow client programs to automatically receive notification of changes made to registered services. This still does not make it dynamic because the client program has to be modified when a notification is received.

Having a centralized registry of services, whether public or private, is necessary for dynamic composition of systems.[4] A problem that applies to public registries is deciding who is responsible for the quality of the information. Another problem that applies to both public and private registries is the need for a common taxonomy or ontology to describe services. Dynamic composition of systems will be challenging until these two problems are addressed.

---

[3]   This is the term used by UDDI to refer to the location of the Web service in the form of a URL.
[4]   This is not to be interpreted as a requirement for the registry to be physically centralized. What is necessary is to have a known place where services are discovered and located, even if the underlying structure of the registry is distributed. This should be transparent for the users of the registry.

5 Open Grid Services Architecture (OGSA)

Grid computing is a form of distributed computing that involves coordinating and sharing computing, application, data, storage, or network resources across dynamic and geographically dispersed organizations [Grid 04].

The Open Grid Services Architecture (OGSA) is an SOA for the Grid. It is a non-proprietary effort by Argonne National Laboratory, IBM, the University of Chicago and other institutions, that combines Grid computing with Web services. The goal of this architecture is to enable the integration of geographically and organizationally distributed heterogeneous components to form virtual computing systems that are sufficiently integrated to deliver desired QoS [Foster 02].

OGSA defines the mechanisms for creating, managing, and exchanging information among entities, called Grid Services. The Open Grid Services Infrastructure (OGSI) defines the standard interfaces and behaviors of a Grid Service [GGF 03]. The Globus Toolkit is an open source implementation of Version 1 of the OGSI Specification. Release 3.2 is available for download from the Globus Alliance Web site [Globus 04, Sandholm 03].

As stated previously, OGSA represents everything as a Grid Service. Grid Services are stateful transient Web Service instances that are discovered and created dynamically to form larger systems [Foster 02]. Transience is what allows for the dynamic creation and destruction of services and has significant implications for how services are managed, named, discovered, and used—that is what makes a Grid Service different from a Web Service. A Grid Service conforms to a set of conventions, expressed as WSDL interfaces, extensions, and behaviors, for such purposes as

- discovery: mechanisms for discovering available services and for determining the characteristics of those services so that they can be invoked appropriately
- dynamic service creation: mechanisms for dynamically creating and managing new service instances
- lifetime management: mechanisms for reclaiming services and state in the case of failed operations
- notification: mechanisms for asynchronously notifying grid service clients of changes in state

As OGSA evolves it will include interfaces for authorization, policy management, concurrency control, and monitoring and management of potentially large sets of Grid Service instances.

OGSA and Interoperability

Interoperability is a requirement for Grid computing. The ultimate goal behind Grid computing is the capacity to leverage resources to carry out massive calculations or distributed operations on demand. The problem is that access to a particular resource requires a set of knowledge and technologies that might be totally different from those of the next resource. There is an obvious need for standardization in this area, and this is what OGSA is trying to do.

Because OGSA is based on Web services, it carries with it all the advantages and disadvantages with respect to interoperability covered in the previous section. OGSA adds capabilities for discovery of services and lifetime management, which are both crucial to the construction of systems on the fly. It also makes Web services stateful, which is important for Grid computing. OGSA is backed by the Global Grid Forum (GGF) and has several working groups exploring issues such as an architecture roadmap, infrastructure, security, and database access and integration. On the security front, the idea is to expose the technologies used within a particular hosting environment as part of its policy so that "secure interoperability" can be achieved.

If two services are OGSA-compliant, the chances of interoperability from a syntactic interoperability perspective are very high. But OGSA still does not totally solve the semantic interoperability problem. There is an operation called FindServiceData that can be performed on a Grid Service. This allows a client to discover more information about a service's state, execution environment and additional semantic details, in essence, to learn more about the service. This is important for interoperability, but unless there is a common ontology to describe Grid Services, reaching semantic agreement will be a problem.

6 Component Frameworks

Component-based development (CBD) has received much attention in the software engineering community. Using CBD, large software systems can be assembled from independent, reusable components. Two component frameworks that support this model are the Java 2 Platform, Enterprise Edition (J2EE), and Microsoft .NET.

Even though the scope of this report is system-of-systems interoperability, and not the interoperability between components to form a single system, these two component frameworks are addressed for two reasons: (1) because there is a general belief that systems developed using the same component framework will interoperate seamlessly and (2) because there is growing interest in the interoperation between systems developed using J2EE and systems developed using .NET.

6.1 Java 2 Platform, Enterprise Edition (J2EE)

Developed by Sun Microsystems, the J2EE defines a standard for developing component-based multi-tier enterprise applications [Sun 04a]. J2EE provides a set of APIs (Application Program Interfaces) to implement availability, security, reliability, and scalability into applications developed under this component framework. Components are mainly developed using the Java language and deployed in containers that transparently provide services to those components, such as lifecycle management, transaction management, access control, and others.

Many vendors have application servers that implement the J2EE specification, such as JBoss, BEA WebLogic, and IBM WebSphere. J2EE runs on a range of operating systems, including Windows, Sun Solaris, UNIX, and Linux. Sun also provides a Compatibility Test Suite to ensure consistent implementation across vendors. Only vendors that pass this test receive certification.

There are several technologies and APIs that are a part of J2EE:

- JavaServer Pages (JSP) and servlets
- Enterprise JavaBeans (EJB)
- Java Naming and Directory Interface (JNDI)
- Java Messaging Service (JMS)
- Java Database Connectivity (JDBC)

- Java Transaction API (JTA)

The current version of J2EE (v1.4) natively supports standards such as SOAP, WSDL, UDDI, and XML. From an interoperability perspective, the J2EE specification now ensures Web services interoperability through support for the WS-I Basic Profile.

6.2   Microsoft .NET

Microsoft .NET is a development environment for creating distributed enterprise applications. The main component of .NET is the .NET Framework, which consists of two main parts: the common language runtime (CLR) and the .NET Framework class library. The CLR allows programs to be written in many different programming languages because it translates them into Intermediate Language (IL). IL is the syntax used to send, receive, and manage .NET signals. The .NET Framework class library includes ASP.NET for developing Web applications and Web services, Windows Forms for user interface development, and ADO.NET for connection to databases [Microsoft 04].

Other components of Microsoft .NET include

- Visual Studio .NET development system
- Windows Server 2003
- Active Directory directory services
- Windows Server system components such as SQL Server 2000 and Exchange Server 2003

From an interoperability perspective, .NET supports standards such as SOAP, WSDL, UDDI, and XML.

6.3   J2EE, .NET, and Interoperability

From a syntactic point of view, the assertion that two systems can interoperate seamlessly because they were built using the same component framework is not always true. In the case of J2EE, because it is a standard, there can be differences between different application server implementations that can cause problems. This is why Sun has a J2EE certification program. For .NET this is less of a problem because of its proprietary nature (Microsoft provides full support for the .NET Framework and there are versions of the Framework that run on most versions of Windows).

In the case for interoperation between component frameworks, there are a number of ways in which to implement J2EE to .NET constructive interoperability:

- Web services

- runtime bridges such as Borland's Janeva, Intrinsyc's J-Integra for .NET (Ja.NET), and JNBridge's JNBridgePro

- message-oriented middleware such as IBM MQseries, Microsoft Message Queue (MSMQ), BEA MessageQ, and Tibco Enterprise Message Server

- a shared database

- integration brokers such as IBM MQSeries Integrator, Mercator CommerceBroker, Microsoft BizTalk Server, and webMethods Enterprise Services Platform

When data exchange between systems is involved, three main challenges exist, mainly because of data type incompatibilities between the languages.[5] A typical example occurs when Java is used for J2EE components and C# for .NET components. These challenges are listed below [Microsoft 03].

- Primitive data type mappings: Even though the same data type may exist in both languages, it cannot be guaranteed that they will map exactly. This is especially true with floating point numbers and strings.

- Non-existent data types: It is possible that a data type in one language does not exist in the other. Typical examples are the specialized data types that represent collections of elements, such as vectors.

- Complex data types: Complex data types that are composed of other data types have to be exposed to the other party so that the proper mapping can be made.

Extensive testing must be done to assure that these problems do not exist.

From a semantic point of view, component frameworks are no different from the approaches discussed before. If there is no common understanding of the data being exchanged, then semantic interoperability has not been accomplished. If the applications are wrapped as Web services, then the semantic interoperability discussion in Section 4 applies.

---

[5] This problem can be extended to Web services as well because underlying components can be implemented using any programming language.

7 Illustrating the Problem of Constructive Interoperability

The previous sections have presented some modern technology approaches to address interoperability requirements between systems and have included a brief discussion on how these approaches relate to syntactic and semantic interoperability. These discussions have been based on the assumption that the interoperating systems have an agreement on the use of a common approach. Here we include a discussion of the more general case when systems use or expose different technologies and are faced with an interoperability requirement.

One current example is interoperability between systems based on different component frameworks. A commonly proposed solution to the problem of making a J2EE application interoperate with a .NET application is to wrap each application as one or more Web services as described in section 6.3. Both applications now use a common technology as an interoperability enabling mechanism. In the development of a system using an MDA approach there may be a requirement to interoperate with certain Web services that already exist independently of the new system. In this situation MDA tools should be available to generate the necessary bridges that allow the new application to call the Web services.

Mary Shaw wrote a paper in 1995 where she listed a series of techniques for dealing with architectural mismatch between components [Shaw 95]. These techniques can be applied to systems and present options for constructing interoperable systems. Most techniques are generally applicable as they can help achieve syntactic as well as semantic interoperability.

1. Change a system's form to another system's form: One system is modified in such a way that it matches the technology or data and operational semantics used/exposed by other systems.

2. Publish an abstraction of the system's form: Systems provide a high-level API for use by other systems. To achieve semantic interoperability the semantics of the exposed operations must match the semantics expected by other systems using it.

3. Transform on the fly: An external mechanism intercepts the interaction between systems and converts from one form to another. Gateways can translate between communication protocols, for example. The transformations must be compatible with the intended semantics of the communication.

4. Negotiate to find a common form: Systems negotiate on the fly to find the optimal common form (the way some modems find the fastest common protocol). This may re-

quire introduction of a third-party entity to act as negotiator between systems and of a protocol for the systems to interact with the negotiator. The simplest instance of this is a negotiator that allows a system to choose among pre-defined alternatives.

5. Make systems multilingual: Systems have the ability to interoperate with multiple other systems because they provide several interfaces or can interact with multiple external interfaces. Services can implement separately defined communication interfaces. For example, a service could have a Web services adapter, an IIOP (Internet Inter-ORB Protocol) adapter, and an MQSeries adapter to serve clients of these three different types.

6. Provide systems with an import/export converter: Systems interact with an external entity that provides conversion services between forms or use extensions that translate to and from other forms on demand. This technique is used in word processors to read and write documents created using a different word processor.

7. Introduce an intermediate form: Systems agree on an intermediate common exchange format (e.g., XML) or introduce a mediator system.

8. Use adapters or wrappers: Adapters and wrappers are pieces of code that encapsulate components and hide their internal details. Systems can build wrappers around them so that they can interact with other systems.

9. Maintain parallel consistent versions: Parallel consistent versions of a system are built so that it can interoperate with other systems. A fairly common case occurs when a system exists in a UNIX and a Windows version to work with other systems in the same environment.

This list of techniques is not complete and they all have advantages and disadvantages. Some techniques will make sense for some systems and will not make sense for others. Some will require the modification of more than one system and some will require the introduction of an additional system or component. Aspects to consider when deciding on a technique include

1. Cost and schedule: Most of the listed techniques require the construction of additional system components or interfaces, thus affecting cost and schedule.

2. System performance: The introduction of any type of mediator between systems will affect performance.

3. On-the-fly requirements: If there is a requirement for systems that are composable on-the-fly, only techniques where interfaces are not decided a priori will be acceptable. On-the-fly transformations and negotiations fall into this category.

4. Flexibility: For systems that have volatile interoperability requirements, a technique where these changes can be isolated from the system itself, such as a wrapper, will provide a better option.

5. Need to reach agreements before building the systems: Some of the techniques will require a higher degree of negotiation between the entities constructing the systems or the

organizations in charge. Introducing an intermediate form, for example, can take a long time that is easily underestimated.

6. Ease: Some techniques will be easier to adopt than others. This is especially true for legacy systems where some technologies may not be available or where modification may prove difficult. Adopting an XML intermediary data representation will be much easier between modern systems running on current platforms than in a situation where there is no off-the-shelf XML parser available on a legacy platform.

7. Diversity: Most interoperability scenarios relate to legacy systems. If this is the case and there is no need or possibility to replace a legacy system, then the selected approach will have to accommodate diversity. Approaches where an intermediate form or an external converter or adapter is introduced will allow the legacy system to remain a part of the system of systems while exposing a more modern interface.

Constructive interoperability is therefore an interesting problem. Selecting the appropriate technique for making systems converge on a common approach is an important aspect of the process and should be made explicit so that entities constructing interoperable systems can plan for the effort.

8 Conclusions and Future Work

A look at some current approaches to constructive interoperability has shown that there is a large emphasis on syntactic interoperability and less work in semantic interoperability. Nonetheless, there is a recognized need for semantics and dynamic composition of systems of systems.

None of the approaches presented in this report implies that they should be used in isolation. These approaches can be combined to form systems-of-systems. For example, one could have systems developed under a certain component framework wrapped as Web services; or one could use MDA to build abstract models for systems and then generate instances of the system on different platforms that communicate using the OGSA architecture. The combination of approaches does not solve the semantic interoperability problem, but it does exploit each approach for its own advantages. Hopefully the advances in ontologies and the Semantic Web could eventually make this a dynamic process in which these systems are composed on the fly.

It is important to state at this point that the information in this report corresponds to what is known about these approaches at the date of this publication. Many standards organizations, vendors, and consortia are working on some of the issues mentioned in this report; also, advances in technology will no doubt make this report outdated. Regardless, the report presents valid issues that have to be considered in system-of-systems interoperability.

This is the first in a series of reports covering constructive interoperability, both at the syntactic and semantic level. An experimentation setup has been established and there is work in progress on a model problem that uses each of the approaches described in this report. Future reports will use the results and lessons learned from this work to describe the experience and provide guidance on when and how to use these approaches when interoperability is a requirement.

There are also plans to investigate the effects of changes in communication protocols on operability. For example, what effort is required to change from using SOAP over HTTP to the Globus Tookit?

Finally, there are plans for a series of directed experiments in order to assess performance, reliability, security, and scalability in systems built using these approaches. Some of the questions that are expected to be answered are as follows:

- What are the response times for communicating between systems and how does it vary depending on the underlying communication technology?

- What is the overhead caused by the parsing of the XML documents used in some of these approaches?

- How secure are the systems created using these approaches? What security infrastructure does each of the approaches provide?

- How do these approaches handle the voluntary or involuntary removal of systems?

- How do these approaches scale? What happens as systems are added?

Appendix A Summary of Approaches to Constructive Interoperability

It is difficult to compare the approaches presented in this technical report as they are different in nature. Table 1 presents a summary of the discussion for each of the approaches. The description of the information contained in each of the columns follows.

- Approach: name of the approach.
- Organizations: organizations responsible or supportive of the development and implementation of the approach.
- Type of Technology: very broad classification of the technology proposed or implemented by the approach.
- Associated or Supporting Technologies: technologies that are associated with the approach or that are required (or suggested) for its implementation.
- Elements that Promote Syntactic Interoperability: aspects of the approach that if used correctly can help achieve syntactic interoperability.
- Elements that Promote Semantic Interoperability: aspects of the approach that if used correctly can help achieve semantic interoperability.

Table 1: Summary of Approaches to Constructive Interoperability

Approach	Organizations	Type of Technology	Associated or Supporting Technologies	Types of Systems where the Approach is a Good Fit	Elements that Promote Syntactic Interoperability	Elements that Promote Semantic Interoperability
Model-Driven Architecture (MDA)	OMG	Model Development and Code Generation	UML, CWM, MOF, XMI	The approach can be used to develop any type of system.	• Generated bridge code • MOF Metamodels • Model interchange language XMI	Achieving semantic interoperability by using MDA is not guaranteed because MOF cannot completely define the semantics of models (e.g., execution of state machines and the interpretation of XMI is partly tool specific). The inclusion of data specifications that specify both syntactic and semantic aspects of the data, as additional information for the transformations or as part of a MOF-based metamodel, can help achieve semantic interoperability but cannot guarantee it.
Service-Oriented Architecture (SOA)	Concept implemented by many organizations: HP, IBM, Sun Microsystems	Architectural Approach	Web Services, Message-Oriented Middleware (MOM), Messaging Services	The concept can be applied to any type of system, but makes the most sense when applied to systems where asynchronous communication is acceptable.	• Common payload and protocol • Published and discoverable interfaces • Loose coupling • Multiple interfaces • Composability	Even though the concept includes published and discoverable interfaces, there are no elements that guarantee semantic interoperability. It will depend on how the service is classified and described, and how this information is shared with potential clients of the service.

Table 1: Summary of Approaches to Interoperability (continued)

Approach	Organizations	Type of Technology	Associated or Supporting Technologies	Types of Systems where the Approach is a Good Fit	Elements that Promote Syntactic Interoperability	Elements that Promote Semantic Interoperability
Web Services	OMG	Communication Infrastructure for Services	WSDL, SOAP, XML, UDDI	The technology can be applied to any type of system, but makes the most sense when applied to systems where asynchronous communication is acceptable. Web services are currently used in the development of new systems, to wrap legacy systems, or to expose parts of a system to the public.	• Capability to implement a service in multiple programming languages • Services communicate using well-known and platform-independent protocols and standards. • Consumers do not need to worry about firewalls because communication is carried over HTTP. • The Web Services Interoperability (WS-I) group is attempting to provide guidance on the usage of Web services standards.	There are many limitations because Web services can currently only be discovered based on keywords. The work on the Semantic Web, the OWL Web Ontology Languages, and other ontologies continues towards overcoming these limitations. Standards such as BPEL4WS (Business Process Execution Language for Web Services), WS-Security, WS-Routing, WS-Transaction, WS-Coordination, and WSCL (Web Services Conversation Language) are working towards the automatic discovery and composition of Web services.

Table 1: Summary of Approaches to Interoperability (continued)

Approach	Organizations	Type of Technology	Associated or Supporting Technologies	Types of Systems where the Approach is a Good Fit	Elements that Promote Syntactic Interoperability	Elements that Promote Semantic Interoperability
Open Grid Services Architecture (OGSA)	The Globus Alliance (Argonne National Laboratory, University of Chicago, IBM, and others), Global Grid Forum (GGF)	Communication Infrastructure for Grid Services	Globus Toolkit, WSDL, SOAP, XML	The goal of this architecture is to enable the integration of geographically and organizationally distributed heterogeneous components to form virtual computing systems that are sufficiently integrated to deliver desired Quality of Service (QoS).	OGSA is based on Web services and therefore brings all its elements that support syntactic interoperability.	OGSA adds capabilities for discovery of services and lifetime management, which are both crucial to the construction of systems on the fly. An operation called FindServiceData allows a client to discover more information about a service's state, execution environment, and additional semantic details. But unless there is a common ontology to describe Grid Services, reaching semantic agreement will be a problem.
Component Frameworks	Sun Microsystems, Microsoft	Development Approach	J2EE, .NET	Component frameworks are used mostly to build systems composed of business objects modeled as object-oriented components.	Systems built using components frameworks can be used in conjunction with Web services, runtime bridges, message-oriented middleware, a shared database, and integration brokers. There is extensive literature and product support to do so.	If there is no common understanding of the data being exchanged, then semantic interoperability has not been accomplished even if the integration is seamless.

Appendix B   Acronyms

API	Application Program Interface
BPEL4WS	Business Process Execution Language for Web Services
CBD	Component-Based Development
CIM	Computation Independent Model
CLR	common language runtime
CORBA	Common Object Request Broker Architecture
DAML	DARPA Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DCOM	Distributed Component Object Model
EJB	Enterprise JavaBeans
GGF	Global Grid Forum
HTTP	Hypertext Transfer Protocol
IIOP	Internet Inter-ORB Protocol
IL	Intermediate Language
ISIS	Integration of Software Intensive Systems
J2EE	Java 2 Platform, Enterprise Edition
JDBC	Java Database Connectivity
JMS	Java Messaging Service

JNDI	Java Naming and Directory Interface
JSP	JavaServer Pages
JTA	Java Transaction API
MDA	Model-Driven Architecture
MOF	MetaObject Facility
MOM	message-oriented middleware
MSMQ	Microsoft Message Queue
OCL	Object Constraint Language
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
OMG	Object Management Group
ORB	Object Request Broker
OWL	Web Ontology Language
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
QVT	Queries, Views, and Transformations
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SoSI	System of Systems Interoperability
UDDI	Universal Description, Discovery and Integration Service

UML	Unified Modeling Language
WS-I	Web Services Interoperability
WSCL	Web Services Conversation Language
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

References/Bibliography

URLs are valid as of the publication date of this document.

[Amazon 96]   Amazon.com.Web Services, 1996.
http://www.amazon.com/gp/aws/landing.html

[Berners-Lee 01]   Berners-Lee, T.; Hendler, J.; & Lassila, O. "The Semantic Web." *Scientific American*, (March 2001).
http://www.scientificamerican.com/article.cfm?articleID=000
48144-10D2-1C70-84A9809EC588EF21&catID=2

[Brown 02]   Brown, A; Johnston, S.; & Kelly, K. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications*. Rational Software Corporation, 2002. http://www-106.ibm.com/developerWorks/rational
/library/4860.html

[Brownsword 04]   Brownsword, L. et. al. *Current Perspectives on Interoperability* (CMU/SEI-2004-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004.
http://www.sei.cmu.edu/publications/documents/04.reports
/04tr009.html

[Cohen 02]   Cohen, F. *Understanding Web Service Interoperability: Issues in Integrating Multiple Vendor Web Services Implementations.* developerWorks, 2002. http://www-106.ibm.com
/developerworks/ webservices/library/ws-inter.html

[Foster 02]   Foster, I.; Kesselman, C.; Nick, J.; & Tuecke, S. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. 2002.
http://www.globus.org/research/papers/ogsa.pdf

[GGF 03]	Global Grid Forum—Open Grid Services Infrastructure Working Group. *Open Grid Services Infrastructure (OGSI) Version 1.0.* June 2003. http://www.gridforum.org/Public_Comment_Docs /Archive_Comments.htm
[Globus 04]	The Globus Alliance. http://www.globus.org/ogsa/ (2004).
[Grid 04]	Grid.org. United Devices, Austin TX. http://www.grid.org (2004).
[Karp 00]	Karp, A. *E-speak E-xplained*. Hewlett-Packard Laboratories, 2000. http://www.hpl.hp.com/techreports/2000/HPL-2000-101.html
[Krill 04]	Krill, P. *Web Services Interoperability Tools Released*. Info-world.com., March 2004. http://www.infoworld.com/article/04/03/17/HNwsitools_1.html
[Microsoft 03]	Microsoft Corporation. *Application Interoperability: Micro-soft .NET and J2EE.* December 2003. http://msdn.microsoft.com/library/default.asp?url=/library /en-us/dnpag/html/jdni.asp
[Microsoft 04]	Microsoft Corporation. *Microsoft .NET,* 2003. http://www.microsoft.com/net/
[Morris 04]	Morris, E.; Levine, L.; Meyers, C.; Place, P.; & Plakosh, D. *Systems of Systems Interoperability* (CMU/SEI-2004-TR-004) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications /documents/04.reports/04tr004.html
[OASIS 02]	OASIS UDDI. *UDDI Version 3.0*. July 2002. http://www.uddi.org/
[OMG 03]	Object Management Group. *MDA Guide Version 1.0.1.*, 2003. http://www.omg.org/docs/omg/03-06-01.pdf
[OMG 04]	Object Management Group. *Committed Companies and Their Products*. http://www.omg.org/mda/committed-products.htm (2004).

[Sandholm 03]	Sandholm, T. & Gawor, J. *Globus Toolkit 3 Core – A Grid Service Container Framework.* 2003. http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf
[ServiceArchitecture 04]	Web Services and Service-Oriented Architectures. http://www.service-architecture.com/ (2004).
[Shaw 95]	Shaw, M. "Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging," 219-221. *Proceedings of the IEEE Symposium on Software Reusability,* Seattle,WA, April 1995. http://www-2.cs.cmu.edu/~Vit /paper_abstracts/Packaging.html
[Stevens 03]	Stevens, M. "Service-Oriented Architecture Introduction." http://www.developer.com/services/article.php/1010451 (2004).
[Sun 04a]	Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee/ (2004).
[Sun 04b]	Sun Microsystems. Jini Network Technology. http://wwws.sun.com/software/jini/ (2004).
[Sycara 03]	Sycara, K. *Autonomous Semantic Web Services* (presentation), 2003. http://www-2.cs.cmu.edu/~softagents /presentations/Caise-final-brief.pdf (2004).
[WS-I 04]	Web Services Interoperability Organization. http://www.ws-i.org/ (2004).
[W3C 03]	W3C. *SOAP Version 1.2 Part 1: Messaging Framework.* June 2003. http://www.w3.org/TR/SOAP/
[W3C 04a]	W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C Working Draft 3, August 2004. http://www.w3.org/TR/wsdl20/
[W3C 04b]	W3C. *Web Ontology Language (OWL).* June 2004. http://www.w3c.org/2004/OWL/

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2004	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Approaches to Constructive Interoperability	F19628-00-C-0003

6. AUTHOR(S)

Grace A. Lewis, Lutz Wrage

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2004-TR-020

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2004-020

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Interoperability between systems requires the capability for users to exchange information (syntactic interoperability) and a common understanding of its meaning or how to act upon it (semantic interoperability). This report will discuss several current approaches to constructing systems of systems that have interoperability requirements, with respect to syntactic and semantic interoperability. The areas examined include Model-Driven Architecture, Service-Oriented Architecture, Web services, Open Grid Services Architecture, and Component Frameworks. These initial discussions assume that the interoperating systems agree on a common approach. Reaching an agreement can be challenging, especially when legacy systems are involved. Techniques and recommendations for reaching an agreement between systems that use differing technologies are also briefly explored.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Constructive Interoperability, Model-Driven Architecture, MDA, Service-oriented Architecture, (SOA),Web services, Open Grid Services Architecture, (OGSA), Component Frameworks	56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL