12-2013

Understanding Patterns for System-of- Systems Integration

Rick Kazman
Carnegie Mellon University, rkazman@sei.cmu.edu

Claus Nielsen

Klaus Schmid

Software Engineering Institute

Understanding Patterns for System-of-Systems Integration

Rick Kazman
Claus Nielsen
Klaus Schmid

December 2013

TECHNICAL NOTE
CMU/SEI-2013-TR-017

Software Engineering and Acquisition Practices

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0000474

Table of Contents

List of Figures

List of Tables

Abstract

Creating a successful system of systems—one that meets the needs of its stakeholders today and can evolve and scale to sustain those stakeholders into the future—is a very complex engineering challenge. In a system of systems (SoS), one of the biggest challenges is in achieving cooperation and interoperation among systems through some form of system integration. Previous work has approached the information system integration challenge in a generic way, not specific to a SoS context, or has provided only a limited range of solutions. This technical report discusses how an IT architect can address the SoS integration challenge from an architectural perspective; it also illustrates the breadth of potential solutions to the challenge through a categorization of SoS software architectural patterns. To demonstrate the practical relevance of this work, the authors instantiate this categorization with a set of patterns described in both the research literature and by companies that support SoS platforms.

1 Introduction

Put simply, a *system of systems (SoS)* is a set of systems that are cooperating and interoperating, while the different systems are simultaneously working as independent entities (and thus not only as parts of the integrated system). This is a shorthand way to say that a SoS can be defined as Maier does, below [Maier 1996].

A *system of systems* is an assemblage of components which individually may be regarded as systems, and which possesses five additional properties:

1. *Operational Independence of the Components:* If the system of systems is disassembled into its component systems, the component systems must be able to usefully operate independently. That is, the components fulfill customer-operator purposes on their own.
2. *Managerial Independence of the Components:* The component systems not only can operate independently, they do operate independently. The component systems are separately acquired and integrated but maintain a continuing operational existence independent of the system of systems.
3. *Evolutionary Development:* The system of systems does not appear fully formed. Its development and existence is evolutionary with functions and purposes added, removed, and modified with experience.
4. *Emergent Behavior:* The system of systems performs functions and carries out purposes that do not reside in any component system. These behaviors are emergent properties of the entire system of systems and cannot be localized to any component system. The principal purposes of the systems of systems are fulfilled by these behaviors.
5. *Geographic Distribution:* The geographic extent of the component systems is large. *Large* is a nebulous and relative concept as communication capabilities increase, but at a minimum it means that the components can readily exchange only information and not substantial quantities of mass or energy.

The paradox of systems being independent while at the same time being part of a SoS can be explained by realizing that a system in a SoS is playing simultaneously different roles in different interactions (typically based on different use cases). Thus, a SoS must be regarded as integrated only in the context of some use cases. Therefore, when one is making design decisions about a SoS, understanding those use cases is important.

As we can see in the above description of a SoS, cooperation (interoperation) among systems is an important criterion. To achieve cooperation among systems, we must perform some form of *integration* of the systems. In fact, no SoS is possible without integration. This integration problem is the focus of our current work; this report describes a primary objective of that work: *supporting software engineers, and in particular software architects, to perform the integration of systems to become a system of systems*. In this report, we describe a first step towards this end: characterizing the integration problem and illustrating the breadth of solution approaches in terms of software architectural integration patterns. It should be noted that SoS integration is not only a technical challenge, but also poses significant challenges from an organizational, managerial, and social perspective. However, in this technical note, we will focus solely on the technical perspective.

That is, we are explicitly avoiding SoS patterns that focus primarily on people, processes, and organizations in a SoS.

While this report has a strong focus on the analysis of systems of systems, this is by no means the only application area of the work described here. As systems of systems are strongly related to concepts such as ecosystems or platforms, and integration plays an important role in those contexts, the problems we investigate here are of prime importance in these contexts as well.

1.1 The Architectural Problem

Our focus in this report is on the perspective of the IT architect: we want to improve how the architect can address the integration problem in a SoS context from a software architectural perspective. An obvious question is whether this is necessary at all. Integration has already been addressed in the literature in many ways, in particular in books such as *Software Architecture in Practice* [Bass 2012] and *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [Hohpe 2003] and similar publications. However, the problem is that these publications either focus on the integration problem in a generic way, as they are not specific to a SoS context, or they provide only limited types of solutions that already make many implicit assumptions about the problem (which are not always appropriate in a SoS context). This makes it necessary to provide specialized support to the integration architect in a SoS context. Throughout this report, we will focus on the integration of systems and hence on patterns that describe the *interaction* of systems.

To characterize the architectural problem in a SoS context, we first must differentiate two fundamentally different situations: integrating a new system into an existing SoS and establishing a new SoS. While related, these fundamentally different situations bring with them different constraints. Further, design constraints are common across the SoS landscape: for example, the need to find architectural solutions that provide integration success while still allowing for an organizational independence of the individual systems. As a major objective of this report, we aim to characterize the architecting context and the design constraints of the integration problem more precisely than previous work has done.

While existing pattern collections for addressing the integration problem are useful, they do not cover the range of issues we think are important. In particular, integration in a SoS context is much broader than mere (message-oriented) information exchange. It can happen on many different levels: for example, in the database layer, the user interface (UI), and the business logic. Important developments support this multi-level integration. For example, the SOA (Service Oriented Architecture) paradigm aims at integrating and establishing interaction between separate applications via data exchange among loosely coupled services. An older example is OLE (Object Linking and Embedding), a technique introduced in Microsoft products to enable interface sharing without overly integrating the individual systems. However, in a SoS context only a small subset of such techniques is typically discussed.

1.2 Contribution of This Report

The focus in this report is mostly on classifying and bounding the problem, less on providing a complete solution by itself. Thus, we provide a classification of the SoS situation to impart an improved understanding of the architecture context. This classification also enables us to describe

relevant design constraints and the design space for systems of systems more effectively than in previous work.

We also illustrate the breadth of potential solutions to the SoS integration problem through a set of patterns. This collection of patterns is not meant to be complete, but to be illustrative of how a collection of SoS integration patterns might be created. We provide an initial set of relevant patterns that aims to outline the space of SoS integration patterns. One way we demonstrate the practical relevance of our work is by including patterns described in the research literature, as well as patterns described by companies that support platform SoS solutions, such as SAP or Salesforce [Sal 2012].

There are many patterns catalogs, and even a few focused on systems of systems (such as the Network Centric Operations Industry Consortium catalog[1]). But the point of our work is not to provide yet another catalog; it is to provide a framework for reasoning about, analyzing, comparing, and choosing patterns for the SoS context.

As future work (and hence not as part of this report), we aim to derive both a more detailed method for developing an integration architecture in a SoS context and a more comprehensive collection of integration patterns.

[1] https://www.ncoic.org/technology/deliverables/patterns/

2 Categorization of Integration Approaches

Before choosing a SoS-wide integration pattern, an architect must be clear on the form of integration to be achieved, which will influence greatly the form of integration pattern chosen. While the need for clarity of purpose may seem obvious, the choice of pattern is not a straightforward decision for two reasons:

1. According to well-established SoS taxonomies (e.g., those presented by Maier), the individual elements within systems of systems have different purposes and control structures [Maier 1996]. Systems of systems may be broadly categorized as virtual, collaborative, acknowledged, or directed [DoD 2012]. The architect must determine, early on in the design process, which kind of SoS is being planned, as this will guide key architectural decisions.
2. Architectural patterns are *collections* of design decisions. Developers seldom clearly explain the motivations for creating them and the forces involved in using them. So the architect must endeavor to match the affordances—strengths and weaknesses—of a pattern to its anticipated use.

Two aspects of design choice are central to the selection of the main SoS patterns for integration: how *data* will be shared and how *control* will be managed. Let us examine each in turn.

Data may be common and shared or private and isolated. If a SoS has common, shared data then any participating system can access any data. This is often the case in modern enterprise systems of systems that use a common data warehouse. Of course, most systems of systems will be somewhere in between—few systems will externalize all of their data and allow any other system to access it, and a SoS that shares no data is inconceivable—how would the participating systems interact?

Control within individual activity sequences that span systems may be, at one extreme, a strictly hierarchical arrangement where one system controls others.[2] At the other end of the spectrum, it may be the case that no one system may control another—each system is autonomous and exercises discretion regarding its own actions. Just as with data, there may be points within this spectrum where individuals operate within a loose hierarchy, but still exercise some individual control.

Such considerations lead us to the categorization in Table 1, where we classify the different kinds of systems of systems that appear at each endpoint of the axes. Of course, many intermediate points are possible as well, but we simply mean to illustrate the pure forms of each decision regarding control and data.

To reiterate, the systems at the extremes of either dimension are likely to be quite rare, perhaps even unthinkable. Most real-world systems of systems will require choices along these control

[2] This might seem counterintuitive at first, as a SoS is defined as one where the various systems of the SoS are managerially and operationally independent. However, here we take the perspective of individual interactions, not of the systems as a whole. For example, an online shop and a credit card payment system are independent, but within the sales transaction the online shop uses the payment system as part of the payment process. Thus, from the perspective of the specific activity sequence, the online shop wields (limited) control.

and data dimensions that reflect their burden of legacy systems, their ownership, and the quality goals that their architects are trying to achieve.

Table 1: Data and Control Choices in a SoS

Control	Data	
	Shared	**Isolated**
Hierarchical	Information system based on call-return paradigm with shared memory	Traditional information system based on call-return paradigm
None	Data-centric system (e.g., data-warehouse-based)	Agent-based system

While virtually every objective is *possible* to achieve in a computational system, given enough time and money, the choice of a specific architectural pattern will make some of the control and data objectives easier or harder to achieve. For example, service-oriented architectures or agent-based architectures tend to be less suited to systems where precise limits of resources must be taken into account. Typically, a tighter integration, going hand in hand with more control in activity sequences, is desirable when quality attributes such as performance and availability are architectural drivers (e.g., in systems with hard real-time deadlines or in safety-critical systems). In such cases, an architect would be well advised to consider architectures with mechanisms that facilitate greater degrees of control, such as an architecture based on remote procedure call (RPC). On the other hand, architectures with highly hierarchical, centralized control tend to limit creativity and exploration, and hence evolution.

Since the choice of a system-wide pattern affects so much in a SoS environment, this is a critical decision for an architect. We do not intend the categorization in Table 1 to aid directly in choosing a pattern but rather to illustrate the issues involved in such a choice. In the next section, we describe a number of characteristics that will have a direct effect on the integration and interoperation of systems within a SoS context.

3 Defining the System-of-System Development Context

As we discussed in the previous sections, the first step in engineering a system of systems is to understand the kind of system one wants to develop and its development context. Thus, it is important that we provide a set of categorizations that help to define more precisely the development context and constraints. These categories will provide one basis for categorizing the SoS integration patterns that Section 5 describes.

3.1 System-of-Systems Scenarios

There are many possible ways to describe the engineering context of a SoS development. We found the following two particularly important, as they significantly restrict the design space and can be determined early on:

- SoS scope: whether we want to develop only a new system within a SoS context or to establish a completely new SoS
- Development context: whether we must take previous development into account

3.1.1 System-of-Systems Scope

System-of-systems scope involves a fundamental decision, as it determines whether we

- must take into account preexisting decisions regarding the SoS design principles, or
- are in a position to actively make such decisions, as the SoS still requires definition

While this decision might appear as a binary decision, in practice it is often more complex. For example, a new SoS usually is not developed without first establishing systems in this context, so the decisions are actually about both the SoS and individual systems. In addition, the complexity and scope of systems of systems may vary widely, leading to correspondingly different impacts on the design space.

3.1.2 Development Context

The second dimension of our categorization is critical to understanding the development context, as we describe above, and the type of decisions we can still make after this context is determined. Especially in a SoS context, systems are often integrated incrementally and not built initially with the specific SoS in mind. This may reduce significantly the design space available to the system integrator. Thus, we differentiate the following three subcategories of development contexts:

1. **Greenfield**: *there are no preexisting implementations that restrict the design space.* Thus, whatever approach to development we choose we can implement. Greenfield implementation for a SoS means the complete SoS must be newly constructed and thus, there are no architectural constraints besides those that are a consequence of the SoS purpose. For an individual system, Greenfield means that only this specific system must be newly constructed; the SoS context still may be preexisting. In this case, the relevant architectural constraints derive from the preexisting parts of the SoS context, but no further constraints apply.

If we consider the enterprise software infrastructure of an enterprise as a SoS,[3] then a new SoS in a Greenfield scenario would be one that we are able to design totally from scratch; therefore all systems could be entirely redesigned.[4] A Greenfield system in this context would mean that an individual system (e.g., the procurement system) would be completely newly constructed.

2. **Brownfield**: *there exists something, but we can (in principle) modify the realization of it.* Here access to the existing implementation is available, and it is, in principle, possible to alter this implementation. While theoretically this situation allows for arbitrary modifications, typically only minimal adaptations are desirable due to cost considerations. Thus, this scenario favors integration approaches that require only changes of rather limited system scope. For a SoS, this means that we could adapt all the existing systems. In particular, we could introduce or replace a common backbone (e.g., middleware). A typical example of this would be the move to web services or a SOA conversion in a company. Typically, this goes along with introducing new application program interfaces (APIs) for the constituent systems and deprecating old ones. For an individual system, it means that we have the specific code and can alter it, for example, in order to introduce connections to new systems (SoS constituents).

   If we consider again the enterprise software infrastructure introduced above, a Brownfield scenario occurs in a SoS context if the company did initially build its own SoS that now must be modified (e.g., by introducing an ESB[5] backbone). A Brownfield scenario also occurs if one of the constituent systems, for which there is full access to the implementation, must be modified while leaving the remaining SoS as is.

3. **Closed Source**: *an implementation already exists, but we do not have access to change it.*

   In this case, the design space is significantly restricted, implying that mainly external adapters can be used, if the existing implementations do not already provide the required integration facilities. This implication holds both for a closed- source SoS context as well as for a closed- source system. More precisely, it usually will not be possible to address a closed source SoS context anyway. While it would be possible to modify the integration technology for the SoS, this would provide limited benefit, as all systems would need to access this new integration technology through translators. In a closed- source system context, we might integrate an existing system with the SoS by creating a facade and, through the facade, connect it without the need to alter the implementation of the system.

   If we again consider our enterprise software infrastructure mentioned above, in a closed source SoS context, all software would have been bought, or the conscious decision made that no modification of existing systems would take place. As we stated above, the modification of the whole SoS rarely will make sense in such a case. An exception might exist in connecting the SoS as a whole to other, external systems. Web-service-based integration comes to mind in this case. More practical might be the case for integrating a new account-

---

[3] We take the point of view here that, since in large organizations the individual systems might be under the control of different sub-organizations (different disciplines or countries), this system might qualify as a SoS.

[4] Of course, in practice, a company hardly will create the whole systems from scratch, but acquire many systems, making a pure Greenfield approach rather unlikely.

[5] Enterprise Service Bus, an integration infrastructure that provides message- and event-based interactions in a heterogeneous environment, often is related to service-oriented architectures.

ing system or perhaps an accounting system that is not yet deeply integrated with the remainder of the enterprise systems. In the case of a third-party development, this would be a typical closed source system context.

Table 2 summarizes this range of possibilities.

Table 2: Matrix for Development Context

	Greenfield	Brownfield	Closed Source
System of Systems	Create a new system of system without the need to take legacy into account (e.g., if a new organization is set up, or a new platform, such as web services, is introduced).	Establish a new SoS by creating new APIs and deprecating existing APIs. The integration of new APIs requires deep modifications (e.g., the move to web services, or SOA conversions in companies).	Wrap existing system in a way that creates interface compliance with the existing environment (e.g., SOA conversions where existing systems are wrapped instead of altered, .net in the Windows environment, and sometimes web services).
System (in SoS)	Create a new system to be integrated into an existing (at least defined) SoS.	Adapt an existing system such that it can be integrated into a SoS. This could be done, for example, by integrating necessary web service interfaces into a legacy system.	An existing system must be integrated in a SoS, but due to a lack of access to its implementation (or for other reasons disallowing alteration) it cannot be adapted. So, it must be wrapped in some way.

As the discussion above illustrates, the different situations provide significantly different architectural constraints that determine the acceptable solutions to the integration problem. However, the border between these various scenarios is often vague in practice. Moreover, existing integration capabilities of the systems must be taken into account. For example, an existing system might already provide a web service interface, so no alteration is needed for moving towards web service-based integration.

3.1.3 Integration Purpose

Another issue, which requires early clarification, is the purpose of integration among the systems. This issue will often require definition on two levels: the purpose of the integration of the system versus the purpose of an individual interface. We identified the following range of potential purposes.

- **One-directional information exchange (inform)**: One system must provide information to one or more systems.
- **Bi-directional information exchange (sync)**: Two (or more) systems must exchange information to keep each other in sync. There is no clear provider-consumer relation among the systems (summarized over all information exchanges).

- **Control**: One system controls the other (the direction may change over time or different aspects might be controlled in different directions). Unlike in information exchange, where the receiving system determines how to act based on the information, in this case the sending system already determines how the receiving system should react.
- **Negotiation**: Multiple systems must negotiate to achieve their particular purpose. This typically involves particular patterns, such as auctions, and may include the previous purposes as special cases (e.g., systems negotiate to determine which instance may exert controlling power).

This list is, of course, not necessarily exhaustive. However, identifying the particular purpose of the integration is important to determining an adequate pattern, as individual (low-level) patterns typically support only a specific purpose.

4 Categorization of Technical Integration Characteristics

The categorizations we presented in Sections 2 and 3 focused mainly on describing the development context and the overall goals of the integration problem. While these are important to understanding the overall context, they are still rather abstract. In this section, we emphasize the more specific technical characteristics that may relate to individual interactions. Of course, this categorization must be interpreted "on top of" the previous ones.

4.1 Using the Technical Categorization

The goal of this technology-oriented classification is to provide a detailed categorization that can be useful for determining relevant patterns for solving integration problems. Note that the categorization we present here is of a more fine-grained nature than the general ones provided in the previous sections. It should not be applied on a system-wide level to determine the integration of a system as a whole in a SoS context; rather, it should be applied to an individual aspect of the integration.

We will illustrate the application with examples. In Section 4.2, we discuss different purposes for integration, such as information exchange or user-interface sharing. Of course, within the integration of a single system into a SoS, multiple goals may apply for different parts of the system. In this case, it is important first to identify the different aspects of the system and how strongly they should be integrated. This information is then used to determine not a single pattern, but a whole set of patterns that support the integration for the different integration levels. Of course, each of the patterns must be compatible with the system-wide context description that we presented in the previous sections. In the case of data abstraction level, we also see that the different levels build on top of each other; thus, multiple layers of patterns might apply simultaneously to solve the whole set of problems, even for a single aspect of integration.

4.2 Integration Level

The *integration level* aims to describe how deeply the different applications are integrated with each other. As there is no generally accepted set of categories for integration, we devised the categories below.

- **Information Exchange (Data Level)**: one system provides information that is used in another system as part of its normal processing. The technical problem here is simply the data exchange, or the common data access. The categorizations in Section 4.3 may offer further refinement of this category.
- **Basic Behavior Interaction (Service Level)**: one system makes use of the capabilities of another. This may be simple service requests or requests with reply. Interactions with a greater complexity are deferred to the next level. Basic behavior interaction also needs to address information exchange as part of the interaction for communicating needed information to the peers.

- **Complex Behavior Interaction (Logic/Business Process Level):** in this case there is a complex interaction among the different systems. In the context of service-oriented architecture, this is often described as choreography and orchestration. Complex behavior interaction differs from basic behavior interaction in that it typically spans across a number of individual interactions and later ones may depend on earlier ones. Complex behavior interaction may include basic behavior interaction and information exchange as sub parts.
- **User Interface Sharing (UI Level):** in this case multiple systems may need to share the user interface. These might be different portals in a web-based interface or it may be that even within a single user interface some regions belong to different systems. A classical way to do this is OLE integration in and with the Office suite. In this case the different systems might not even know about each other.

4.3 Data Abstraction Level

For communication and integration among systems to occur, the mutual understanding of any communicated data is important and requires a common basis on multiple levels. Typically the following levels are differentiated [COE 2010, p.18]:

- **Structural**: defined as adherence to relevant standards that describe data exchange. This might be a low-level standard that does not necessarily determine the details of the data format.
- **Syntactic**: defined as data exchange occurring with the appropriate formats. Often standards will define both structural and syntactic aspects. In particular we will use *syntactic* to denote that higher level data types are appropriately mapped.
- **Semantic**: defined as providing the correct data as part of the data exchange. For example, the customer address mentioned above is really a known customer address and the correct one that should be used here as part of the processing (the latter is also called *relevant* [COE 2010]).

Most patterns that we could identify do not by themselves enforce a given level. Thus, additional measures must be taken to ensure a given abstraction level.

4.4 Data Level Integration

Data level integration addresses how integration of data is performed; it has a significant impact on the strength of coupling of the individual systems and hence of the appropriate integration patterns. However, data level integration requires that the decision on how integration of data is performed can already be made.

- **File Transfer**: This is the mechanism that induces the lowest amount of coupling. Data is written to some file in a defined format at a specified place by one application and then read under the same assumptions by a different system. However, this mechanism restricts the range of applicability of the data level integration approach, as no events are generated that can be used to communicate the availability of data among the different systems.
- **Message-Exchange**: Integration is achieved by using messages to communicate among the individual systems. This is the approach emphasized by Hohpe and Woolf as well as other

researchers [Hohpe 2003]. An advantage is the loose coupling that such a pattern brings about.

- **Streams**: Often the systems must communicate through continuous streams of data. Typical examples are video streams or streams of price data at stock exchanges. While, in principle, streaming can be seen as a consecutive sequence of messages, it brings about some special criteria; these include order preservation and minimizing delays and interruptions, often on a millisecond or microsecond level, combined with the limitation of being able to access only a very limited window of a potentially infinite data stream at any point in time.
- **Common Data**: This method is fundamentally different from the previous two. As data is shared and accessed in an interleaving way by different entities, the guarantee of SoS-level consistency becomes a difficult issue, as it must be ensured that different applications use and interpret the data in the same way. However, this method also allows for additional integration applications, such as cross-application data mining.

When applying this categorization, one should again be aware that a single system might well use multiple of these methods to share data concurrently, typically for different types of data. Thus, an individual pattern will address only one of these data level integration modes, and only for the relevant part of the information.

4.5 Interaction Style

Interaction style describes the form of interaction implied by the pattern, and is strongly related to the Integration Purpose (see Section 3.1.3). However, here we focus on the style of integration of a single system. We discuss the different interaction styles roughly in the order of the increasing coupling that they induce.

Send: uni-directional sending of information from one system to one or multiple others in a SoS context.

Call: not only transfers information, but also triggers certain actions. By itself it is asynchronous, but often it will either implement a call-return or call/call-back approach.

Call-Return: refines the call by synchronously transferring control to a different system (typically this happens only on a single thread) and waiting until the other system answers and returns a result.

Call/Call-Back: an asynchronous variant of the above, where the reaction of the called system is not transferred as a return to the original call, but indirectly by performing a separate call back to the originating system at a later point in time.

Time-Based: a form of interaction that uses timing to synchronize behavior among the participating systems. In one form of this style, one system provides a file at certain points in time that is read at specific points in time by a different system. Here the time frame is typically every 24 hours or similarly large. Other variants of this form include polling or time-slice-based bus sharing, where the time frame is typically in the millisecond range. The drawback of this approach is that the various systems must be synchronized by implicit design decisions, such as the position of time slots.

Multi-Call Protocols: an extension of the call-based interaction. While the above styles focus on individual interactions, multi-call protocols describe more complex interaction scenarios, for example 1-N messaging. For example, we will typically see more complex interactions in negotiations (see Section 3.1.3). This subsumes a large number of different protocols and could in principle be further subdivided.

The different interaction styles relate to the integration purposes described in Section 3.1.3. For example, one-directional information exchange will usually be realized by send or time-based interaction. Bi-directional information exchange is typically implemented by send, call-return, and call/call-back. Control is typically realized through use of patterns that support some form of call. Negotiation typically relies on multi-call protocols.

4.6 Quality of Integration

Finally, the quality that must be ensured as part of the integration is an important criterion. While integration and interoperability are typically qualities in their own right [Bass 2012], and these are the main foci of integration patterns, often *additional* qualities must be ensured with respect to the integration.[6] Below we identify the additional qualities that we have observed as being the most important and common in integration patterns.

Reliability: requires that the integration works reliably, meaning that at least the system initiating the communication is informed if the integration breaks down (e.g., if the communication fails).

Performance: requires that the integration perform adequately; in particular, that the integration activities do not require too many intermediate steps or excessive executions and data transfers.

Security: requires that the integration is secure; for example, no alteration of the exchanged information may occur, and it must be possible to assure the source of the data.

Availability: requires that the integration source / destination remain available.

Interoperability: assures the connectivity and information interchange among systems. Interoperability concerns technology and engineering challenges related to communication, data management, semantics, architectural mismatches, and similar aspects.

Scalability: requires that the integration is scalable across large numbers of systems. Thus, the integration will work correctly if many different systems are integrated, many instances of individual systems are integrated, or much information is exchanged (a performance consideration).

Manageability: requires that it is easy to manage the integration. For example, the ability to easily change the peers of the integration might support this.

Consistency: ensures the validity and integrity of the data shared between systems when integrated.

[6] This should not be confused with the qualities of the application themselves.

The above is a non-exhaustive list of qualities that stakeholders might demand in combination with the integration. An individual pattern might support a quality in combination with ensuring the integration, or additional patterns and tactics might support these qualities, and the integration pattern can be combined with those.

5 Pattern Overview

Based on the discussions and descriptions of categories in Sections 3 and 4 and Appendix A, we describe in this section a template for characterizing different integration patterns and provide a collection of different patterns. This set of patterns is far from complete; we intend that they merely exemplify the broad scope of integration patterns. As evidence that this is a reasonable representative set of patterns, we note that the patterns presented in this report cover the space of possible pattern categories, as we will discuss in Section 5.10.

Before we present our patterns and our template, we first give an example of how patterns are described in the research literature and in the patterns community.

5.1 Pattern Example

This example of the Broker pattern is adapted from the book by Bass and colleagues [Bass 2012]. While there is no universally agreed-upon format for presenting patterns, this example contains the major components found in all patterns catalogs.

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request. The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client. In this way, the client remains completely ignorant of the identity, location, and characteristics of the server. Because of this separation, if a server becomes unavailable, a replacement can be dynamically chosen by the broker. If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change, and so the client is unaffected. Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshaling and unmarshaling messages.

The down sides of brokers are that they add complexity (brokers and possibly proxies must be designed and implemented, along with messaging protocols) and add a level of indirection between a client and a server, which will add latency to their communication. Debugging brokers can be difficult because they are involved in highly dynamic environments where the conditions leading to a failure may be difficult to replicate. The broker would be an obvious point of attack, from a security perspective, and so it needs to be hardened appropriately. In addition, a broker, if

it is not designed carefully, can be a single point of failure for a large and complex system. Brokers also can potentially be bottlenecks for communication.

Table 3 summarizes the solution of the broker pattern.

Table 3: Broker Pattern Solution

Overview	The broker pattern defines a runtime component, called a broker, which mediates the communication between a number of clients and servers.
Elements	*Client*, a requester of services *Server*, a provider of services *Broker*, an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client *Client-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages *Server-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
Relations	The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
Weaknesses	Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. The broker can be a single point of failure. A broker adds up-front complexity. A broker may be a target for security attacks. A broker may be difficult to test.

5.2 Pattern Template

Table 4 presents our template for listing a SoS pattern's attributes and values. In the column "Value Range," the template shows the possible values that might be relevant for various patterns. If an attribute value is only partially relevant it is shown in parentheses. For each attribute, a particular pattern would select one or more of the values listed here.

The various attributes shown here have been discussed in the previous sections, with the exception of the LISI and PAID attributes, which have been discussed in previous literature [Morris 2004] and are summarized in the appendix.

The possible LISI (Levels of Information System Interoperability) attribute values are the following: Isolated, Connected, Functional, Domain, or Enterprise. "Enterprise" means that all applications share data and collaborate across the enterprise. At the "Domain" level of information exchange, there are shared databases and sophisticated collaboration among separate applications. At the "Functional" level there is heterogeneous exchange of products and basic collaboration, with few common functions. The "Connected" level describes the state where data and applications are separate, but can communicate by, for example, messages, exchange of text files, and email. Finally, at the "Isolated" level, the various systems are not connected.

The PAID attributes are part of the LISI model and describe what aspects of integration are supported. The possible values are the following: "Procedures"—the degree to which procedures and governance are integrated; "Applications"—the degree to which applications are integrated from single processes to applications suites; "Infrastructure"—the infrastructure components that support the integration; and "data"—the range of data formats and standards that support interoperability.

Table 4: Template for Integration Pattern Matrix

Attribute	Value Range
SoS-Scope	System of Systems, System
Development Context	Greenfield, Brownfield, Closed Source
Integration Purpose	One-Directional Information Exchange, Bi-Directional Information Exchange, Control, Negotiation
LISI	Isolated, Connected, Functional, Domain, Enterprise
PAID Attributes	Procedures, Applications, Infrastructure, and Data
Integration Level	Information Exchange, Basic Behavior Interaction, Complex Behavior Interaction, User Interface Sharing
Data Abstraction Level	Structural, Syntactic, Semantic
Data Level Integration	File-Transfer, Message-Exchange, Streams, Common Data
Interaction Style	Send, Call, Call-Return, Call/Call-Back, Time-Based, Multi-Call Protocols
Quality Attributes of Integration	Reliability, Performance, Security, Availability, Interoperability, Scalability, Manageability, Consistency

Attribute	Value Range
Pattern is defined by	Reference to pattern documentation

5.3 Salesforce Integration Patterns

In an attempt to include patterns that are used in industry as best practice, we turned to the set of documented Salesforce integration patterns [Sal 2012]. The documentation makes clear that the patterns are particularly adapted to the situation of Salesforce.com. However, the underlying patterns are generic and useful in a broad range of SoS integration scenarios.

5.3.1 User Interface Update Based on Data Changes

This addresses the problem of a deep integration among multiple systems. The pattern describes how a UI may subscribe to information from a data stream to update the view, if a change of the underlying data occurs. In particular, this implies that the user interface must be aware of different potential data sources. However, the fact that the data may originate from different systems in the SoS environment will be hidden when a client requests data in a generic way (e.g., by interacting with a broker).

The actual data integration occurs through messaging. The pattern uses an event-based approach in which systems subscribe to change notifications. The originating system includes data in the notification, and leaves it to the subscribers to handle it. Thus, there is no connotation of control in this case, even though a visible reaction will occur. The pattern can be used in the Salesforce context for creating new systems, but it could also be applied in general for the creation of new systems of systems. The pattern requires deep data integration and semantic interoperation. Therefore, the receiver must be able to understand the data and judge whether it is adequate for display in the current view.

Table 5: Matrix for User Interface Based on Data Changes

Attribute	Pattern Value
SoS Context	
SoS Scope	System, (System of Systems)
Development Context	Greenfield (Brownfield)
Integration Purpose	One-Directional Information Exchange
LISI	Functional
PAID Attributes	Infrastructure, Data
Technical Categorization	
Integration Level	Information Exchange
Data Abstraction Level	Syntactic, Semantic

Attribute	Pattern Value
Data Level Integration	Streaming
Interaction Style	Send
Quality Attributes of Integration	Adheres to Salesforce organization-level security
Relation to Patterns	
Pattern is defined by	Salesforce Integration Patterns: UI Update based on Data Changes [Sal 2012, p.8]

5.3.2 Remote Process Invocation – Request and Reply

This section addresses the need to make explicit changes in a different system as a result of some current activity in the local system. The Salesforce documentation discusses four different approaches; we will use only the first three as relevant examples [Sal 2012]. In this specific case, either through a button or a data change, a SOAP or a HTTP request is initiated. The corresponding response is handed back to the caller.

This interaction style can easily be introduced in any system where adaptation is possible, as it provides a simple call-return style of integration. It supports bi-directional information exchange and is bi-directional by nature. When a result is given back to the caller, the information transfer is reliable (in the sense that at least a notification of a problem is provided). The pattern also integrates well with other patterns, such as a broker or a load balancer, to achieve further qualities of integration.

The connection might occur on many different levels, ranging all the way from the user interface to the data layer; it can thus serve a number of integration purposes [Sal 2012].

Table 6: Matrix for Remote Process Invocation

Attribute	Pattern Value
SoS Context	
SoS Scope	System (System of Systems)
Development Context	Greenfield, Brownfield, (Closed Source)
Integration Purpose	Bi-Directional, Information Exchange
LISI	Domain
PAID Attributes	Applications / Procedures
Technical Categorization	
Integration Level	Basic Behavior Interaction

Attribute	Pattern Value
Data Abstraction Level	Syntactic (Web Services Definition Language (WSDL), but semantics of WSDL must be developed)
Data Level Integration	Message-Exchange
Interaction Style	Call-Return, Call/Call-Back
Quality Attributes of Integration	Reliable (semi-batch variants for high performance)
Relation to Patterns	
Pattern is defined by	Salesforce Integration Patterns: Remote Process Invocation – Request and Reply, [Sal 2012, p.11]

5.3.3 Batch Data Synchronization

Batch Data Synchronization is often used to integrate data among different systems in a high-performance (but not necessarily timely) way. In this case, a large number of different data entries are written into a single data structure that is then handed over to the communicating system. The hand-over process may—or may not—be combined with an event. If no event is created, the synchronization is typically triggered on a time basis. In the specific case of the Salesforce infrastructure, this pattern may be used in two ways: for importing data from foreign systems into Salesforce or by exporting data from Salesforce for consumption by other systems.

This approach can be applied in virtually any situation, as it only requires the integration of a reader (or writer) into the system. Even in Brownfield situations, this approach can be often applied; most systems possess such an interface, or a translator can be conceived to integrate an existing system with such an interface. The focus is purely on data exchange in this case. Typically, such an interface style does not provide specific constraints on the data format. However, in the Salesforce case, this pattern is used together with Extract, Transform, Load (ETL) tools that typically provide some constraints on the data format.

This pattern is quite common, especially when used in connecting to legacy systems.

Table 7: Matrix for Batch Data Synchronization

Attribute	Pattern Value
SoS Context	
SoS Scope	System
Development Context	Greenfield, Brownfield, (Closed Source)
Integration Purpose	(Batch) Information Exchange
LISI	Connected

Attribute	Pattern Value
PAID Attributes	Data
Technical Categorization	
Integration Level	Information Exchange
Data Abstraction Level	Structural (Syntactic)
Data Level Integration	File-Transfer
Interaction Style	Time-Based
Quality Attributes of Integration	Performance
Relation to Patterns	
Pattern is defined by	Salesforce Integration Patterns: Batch Data Synchronization [Sal 2012, p.28]

5.4 Data Warehouse Integration Patterns

Data warehouses are widely used in industry as a means of integrating data that will be shared across multiple systems. As such, the warehouse serves as an integration vehicle for those systems (as noted in Table 1).

5.4.1 History Pattern

The history pattern is one of a set of data warehouse integration patterns identified by Köppen and colleagues [Köppen 2011]. The problem focus is that master data,[7] while it is fundamental to system operation and overall rather stable, will sometimes change. In such a case, the time windows of validity of the data must be captured so that the systems can adequately operate with this data. The history pattern addresses only how some aspect of data is communicated across a range of systems and in this process makes important assumptions about the overall form of integration (data warehouse). Thus, it has a clearly described range of applicability; for example, it can be applied in a system of systems, particularly if a data warehouse is planned anyway. It cannot be applied on a single system level, as it relies on shared assumptions regarding the data warehouse and its data structure.

The history pattern requires enterprise-level integration and addresses only the data and infrastructure levels. It addresses only data integration and deals directly with the semantic interpretation of data (validity time stamps).

[7] Master data typically refers to rather stable "configuration" data in an information system that is vital to its operation.

Table 8: Matrix for History Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System of Systems
Development Context	Greenfield, Brownfield
Integration Purpose	Track Data Evolution
LISI	Enterprise
PAID Attributes	Data/Infrastructure
Technical Categorization	
Integration Level	Data Integration
Data Abstraction Level	Semantic
Data Level Integration	Common Data
Interaction Style	Call-Return
Quality Attributes of Integration	Manageability
Relation to Patterns	
Pattern is defined by	[Köppen 2011, p. 53]

5.5 SAP3 Integration Patterns

The following integration patterns are all found in the book *Software Architecture in Practice, 3rd edition* [Bass 2012].

5.5.1 SOA (Service-Oriented Architecture)

The SOA pattern is a natural and popular choice for many modern systems of systems. It is organized around the concept of loosely coupled, distributed *services*, which are offered, described, and implemented by service providers. The services may be implemented in different languages, on different computing platforms, and by different organizations. Service consumers and providers may be entirely unaware of each other's existence, which makes this pattern well suited to support interoperability and system evolution. Service interactions are typically mediated by an *enterprise service bus* that routes messages between service providers and consumers. An SOA implementation typically provides a *service registry* so that services can learn of each other's existence. In addition, many implementations include an *orchestration engine* to support the creation and execution of complex workflows.

Table 9: Matrix for SOA Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System of Systems, System
Development Context	Greenfield, Brownfield, Closed Source
Integration Purpose	Information Exchange, Collaboration
LISI	Domain, Enterprise
PAID Attributes	Applications, Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction, Complex Behavioral Interaction
Data Abstraction Level	Syntactic, Semantic
Data Level Integration	Message-Exchange
Interaction Style	Call-Return, Send
Quality Attributes of Integration	Availability, Performance, Security, Interoperability
Relation to Patterns	
Pattern is defined by	[Bass 2012, pp. 222-226]

5.5.2 Peer-to-Peer (P2P)

The peer-to-peer (P2P) pattern underlies some of the largest systems known. The communication mechanism underlying the internet, for example, is a P2P system. In the P2P pattern, components directly interact as *peers*. All peers are "equal" and no peer or group of peers can be critical for the health of the system. For this reason, this pattern is attractive in a SoS context. Peer-to-Peer communication is typically a request/reply interaction. That is, any component can, in principle, interact with any other component by requesting its services. Note that, while request/reply appears to be superficially similar to call-return, it is fundamentally different in terms of how *control* is managed, since peers operate asynchronously and do not suspend themselves until their requests receive a reply.

Table 10: Matrix for Peer-to-Peer Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System of Systems, System
Development Context	Greenfield, Brownfield
Integration Purpose	Information Exchange, Collaboration, Negotiation
LISI	Domain, Enterprise
PAID Attributes	Applications, Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction, Complex Behavioral Interaction
Data Abstraction Level	Syntactic, Semantic
Data Level Integration	Message-Exchange
Interaction Style	Send, Call
Quality Attributes of Integration	Performance, Availability
Relation to Patterns	
Pattern is defined by	[Bass 2012, pp. 220-222]

5.5.3 Broker

The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a *broker*. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request. The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client. In this way, the client remains ignorant of the identity, location, and characteristics of the server. Because of this separation, if a server becomes unavailable, a replacement can be chosen dynamically by the broker. If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change, and so the client is unaffected. Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshaling and unmarshaling messages.

Table 11: Matrix for Broker Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	SoS
Development Context	Greenfield, Brownfield
Integration Purpose	Information Exchange, Collaboration, Negotiation
LISI	Domain, Enterprise
PAID Attributes	Applications, Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction, Complex Behavioral Interaction
Data Abstraction Level	Syntactic, Semantic
Data Level Integration	Message-Exchange
Interaction Style	Call-Return, Call/Call-Back
Quality Attributes of Integration	Interoperability
Relation to Patterns	
Pattern is defined by	[Bass 2012, pp. 210-212]

5.5.4 Publish-Subscribe

In the Publish-Subscribe pattern components interact via messages (sometimes called *events*). Components may subscribe to a set of events. It is the job of the Publish-Subscribe runtime infra-structure to make sure that each published event is delivered to all subscribers of that event type. Thus, the main form of connector in these patterns is an *event bus*. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events. In addition, any component may be both a publisher and a subscriber.

Table 12: Matrix for Publish-Subscribe Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	SoS
Development Context	Greenfield, Brownfield
Integration Purpose	Information Exchange, Collaboration, Negotiation
LISI	Domain, Enterprise
PAID Attributes	Applications, Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction, Complex Behavioral Interaction
Data Abstraction Level	Syntactic, Semantic
Data Level Integration	Message-Exchange
Interaction Style	Send
Quality Attributes of Integration	Interoperability, Manageability
Relation to Patterns	
Pattern is defined by	[Bass 2012, pp. 226-229]

5.6 Pattern-Oriented Software Architecture

The following integration patterns are all found in the book *Pattern-Oriented Software Architecture, Volume 4* [Buschmann 2007].

5.6.1 Blackboard

By its very nature, a SoS involves some degree of cooperation among the constituents that make up the SoS. The systems (sometime) interact and collaborate to reach a solution that fulfills a common goal.

The Blackboard pattern offers a way for distributed systems to work together in a common solution space. Using the Blackboard, multiple systems can give their contributions to solving the problem and react to each other's input to make decisions. In addition, the Blackboard makes it easy to scale such systems, adding new loosely coupled collaborators.

It is easier to adopt this pattern in a Greenfield implementation, while it can be more difficult to implement in a system involving legacy systems that have a closed source. The pattern requires a central storage repository for the shared solution space, and often the pattern includes some kind of prioritization of goals and inputs. Therefore, it can be a challenge to implement the pattern in a SoS with no overall centralized authority. It can, however, be done by adapting systems through the use of some intermediates to perform the coordination, such that the legacy systems need not realize that they are participating in a Blackboard pattern.

The pattern has the benefit of having a clear semantic understanding of shared data by making use of global information. This makes the pattern work at the highest level of integration, where each system is utilizing the capabilities of other systems.

Table 13: Matrix for Blackboard Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System
Development Context	Greenfield, Closed Source
Integration Purpose	Information Exchange, Decision Making
LISI	Domain, Enterprise
PAID Attributes	Applications, Data
Technical Categorization	
Integration Level	Complex Behavioral Interaction
Data Abstraction Level	Semantic
Data Level Integration	Message-Exchange
Interaction Style	Multi-Call Protocols
Quality Attributes of Integration	Interoperability, Scalability
Relation to Patterns	
Pattern is defined by	POSA 4 [Buschmann 2007, p. 204]

5.7 Messaging

5.7.1 Pipes and Filters

Doing complex operations on data is a challenge in a SoS, as each sequence of the processing steps often will occur on distributed independent systems that use a broad range of data formats.

The Pipes and Filters pattern enables a sequence of transformations and data processing to be combined in a flexible way. The pattern uses a simple interface to ensure that the constituent systems are compatible and can be connected in a flexible sequence to establish a pipeline. Creating an architecture where a pipeline forms the interconnection, and the constituent systems are considered filters is a means of structuring the SoS in a flexible way, which ensures that the constituents are independent from each other.

A challenge of using the pattern is that the interface connecting the filters must be agreed upon. The required interface may entail that the legacy system needs an adaptor to participate in the pipeline. However, the pattern should enable existing systems to be composed without necessarily altering them. Another challenge is constructing the pipeline itself, which may require some type of centralized entity. Regarding the level of integration, the pattern can be used both for systems that have separate data as well as for more integrated systems that have shared data models.

Table 14: Matrix for Pipes and Filters Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System
Development Context	Greenfield, Brownfield
Integration Purpose	Information Exchange
LISI	Functional, Domain
PAID Attributes	Infrastructure /Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction
Data Abstraction Level	Structural
Data Level Integration	Streams
Interaction Style	Send, Call
Quality Attributes of Integration	Interoperability, Manageability
Relation to Patterns	
Pattern is defined by	[Hohpe 2003, p. 70]

5.7.2 Dynamic Router

A SoS is always evolving, which means that the system topology is constantly changing over time. This evolution presents a challenge for the exchange of data between the distributed systems; these changes will affect the routing of data.

The Dynamic Router pattern aims to maintain the exchange of data between the distributed systems without losing efficiency, because the Dynamic Router is not responsible for tracking each individual distributed system. Instead, the individual constituent systems are responsible for announcing changes by sending special control messages to the Dynamic Router. Having a Dynamic Router in a SoS context means that you get a system structure that can evolve, is largely self-regulating, and is scalable over time.

As the pattern requires the participating systems to announce themselves initially, as well as to announce any changes in conditions, it is most suitable for a SoS in which a common agreement can be made among the individual constituents. Such a scenario may be more easily accomplished in a Greenfield development context. Agreeing on supplying these announcements of changed conditions is simply the price for participating in a SoS using the Dynamic Router pattern. The individual systems must do some maintenance of their own, but they also gain from the services delivered by the router. The requirement for participation also means that the pattern is sensitive to autonomous systems that unexpectedly choose not to follow the common agreement, or become disconnected from the SoS due to failures.

Table 15: Matrix for Dynamic Router Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System
Development Context	Greenfield
Integration Purpose	Information Exchange
LISI	Functional, Domain
PAID Attributes	Infrastructure, Data
Technical Categorization	
Integration Level	Information Exchange, Basic Behavior Interaction
Data Abstraction Level	Structural
Data Level Integration	Message-Exchange
Interaction Style	Send
Quality Attributes of Integration	Scalability, Manageability
Relation to Patterns	
Pattern is defined by	[Hohpe 2003, p. 243]

5.7.3 Canonical Data Model

Building a SoS is all about the interaction between distributed constituent systems. This interaction often involves a large degree of information exchange between heterogeneous systems that may not share structures, notations, and ways of data interpretation.

The Canonical Data Model aims to minimize the dependencies that can arise from the need for conversion between data formats by providing a level of indirection. The benefit of having a precise and uniform understanding and interpretation of data formats throughout the systems is obvious. It can, however, be challenging to implement the pattern in a SoS context, as this often involves both legacy systems and dispersed ownership of the individual constituents. Therefore, the type of SoS and its current development context must be considered. In a SoS with no collaborative agreement between the owners of the constituents systems, it would be difficult to introduce a Canonical Data Model, while it might be considerably easier in a SoS where the ownership of the constituent systems is more centralized. Likewise, it would be easier to utilize in a Greenfield development context, as the development of a highly established SoS might require the addition of Message Translators between the Canonical Data Model and legacy formats.

As introducing the Canonical Data Model can be complicated, it may not be the initial best fit for a SoS with a small number of constituents. However, one must consider the extent to which the SoS may evolve over time. With the increasing size and scope of an evolving SoS, the cost of utilizing the Canonical Data Model may prove valuable over the long term. The pattern fits into the Domain level of the LISI model, with a clear focus on Data attributes supporting large integrated systems with shared domain data.

Table 16: Matrix for Canonical Data Model Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System of Systems
Development Context	Greenfield
Integration Purpose	Information Exchange
LISI	Domain
PAID Attributes	Data
Technical Categorization	
Integration Level	Information Exchange
Data Abstraction Level	Semantic
Data Level Integration	Message-Exchange
Interaction Style	Send

Attribute	Pattern Value
Quality Attributes of Integration	Scalability
Relation to Patterns	
Pattern is defined by	[Hohpe 2003, p. 355]

5.8 Patterns from Enterprise Application Architecture

5.8.1 Remote Façade

A SoS can consist of large and complex systems that offer a great deal of advanced functionality. These systems may not be designed for use in this particular SoS, or they may be meant to interact with various other constituent systems that each want to access different functionality. Interacting with such systems of systems may be challenging; their advanced functionality may necessitate complex and detailed interfaces.

The Remote Façade pattern delivers a coarse-grained access to these complex and detailed systems. The pattern can be seen as a type of single-address gateway that wraps complexity into a simpler, unified interaction. Accessing a system through a Remote Façade will provide a strong focus on certain capabilities as seen from specific systems. However, some system-wide knowledge may be needed to shape the Remote Façade towards the needs of the specific systems. In certain cases, the Remote Façade also can minimize the number of interactions needed between systems, because of the coarse-grained approach.

The pattern is a way of affecting the structure of the overall system by using a fairly simple concept working at the lower levels of integration. Still, the pattern's focus on individual capabilities makes both access and use of the system considerably easier and faster under certain conditions.

Table 17: Matrix for Remote Façade Pattern

Attribute	Pattern Value
SoS Context	
SoS Scope	System
Development Context	Brownfield, Closed Source
Integration Purpose	Information Exchange
LISI	Functional, Domain
PAID Attributes	Applications
Technical Categorization	
Integration Level	Basic Behavior Interaction

Attribute	Pattern Value
Data Abstraction Level	Structural
Data Level Integration	Message-Exchange
Interaction Style	Call, Call-Return
Quality Attributes of Integration	Performance
Relation to Patterns	
Pattern is defined by	[Fowler 2003, p. 388]

5.9 Cooperative Platforms

5.9.1 Collaborative Virtual Environments

The integration effort between the constituents in a SoS occurs on all levels, from the hardware level to the user interfaces. Enabling constituent systems to work cooperatively on the user-interface level presents a major integration challenge. The individual constituents will have different ways of portraying data and of interacting with the users of the system. Some may have an extensive human computer interface, others may be headless systems, and some may have no interface directed towards human users. The challenge is to integrate these different degrees of user interface into a coherent user environment that can reflect the capabilities arising from the joining of constituent systems.

Depending on the purpose of the SoS, it may require a user interface that allows multiple stakeholders and users to interact and monitor the SoS as a whole.

Collaborative Virtual Environments focus on establishing collaboration among dispersed users by creating a shared environment that allows them to interact. The core element of Collaborative Virtual Environments is to establish a shared context through User Interface Sharing, which creates a collaborative space that can be accessed and altered synchronously. Having a shared context enables users to view or alter the data and processes in the overall system that is composed of many distributed constituent systems. Being able to see the behavior of other users and systems helps create a common understanding of underlying actions and awareness of the current goals in the system.

Creating a Collaborative Virtual Environment in a SoS context entails a system architecture that allows for advanced forms of collaboration at a user level, with multiple users simultaneously accessing a shared information space. A shared user interface may be the technology needed to satisfy the demands for insight and joint responsibility of the overall SoS, which are required by the many stakeholders and users of a SoS. However, achieving integration at the user-interface level has many challenges. As the system may require bi-directional information sharing and potentially performs a large number of data modifications, it must have fast access to shared resources. Creating a Collaborative Virtual Environment requires advanced mechanisms in the SoS architecture and user interface framework for handling both legacy systems and the evolution of

the SoS over time. Finally, despite the overview and insight afforded by a Collaborative Virtual Environment, the fact remains that decision making in a SoS, involving no real authority, is complex and easily becomes unmanageable.

Table 18: Matrix for Collaborative Virtual Environment

Attribute	Pattern Value
SoS Context	
SoS Scope	System of Systems
Development Context	Greenfield, Brownfield
Integration Purpose	Bi-Directional Information, Control
LISI	Enterprise
PAID Attributes	Applications
Technical Categorization	
Integration Level	User Interface Sharing
Data Abstraction Level	Semantic
Data Level Integration	Message-Exchange, Streams
Interaction Style	Multi-Call Protocols
Quality Attributes of Integration	Security, Availability
Relation to Patterns	
Pattern is defined by	[Churchill 2001]

5.10 Summary of Patterns

Looking at our classification of the various patterns, we might ask whether our examples cover the space of the available categorizations. To determine the answer, we summarized all values for all possible categories and listed them in the following table. We see that the patterns are not evenly distributed across the various categories. However, it is difficult to determine the source of this effect. Different possible interpretations are below:

- The range of all possible patterns is not evenly distributed according to our categorizations.
- We had some bias when performing our categorizations. This hypothesis is supported by the fact that the corresponding categorization was not always clear.
- Some categories of patterns are more common than others.

We tend towards the last interpretation and assume that our survey might be roughly in line with the overall trend. We presume, as a foundation for this interpretation, that some problems can be more readily solved (and in more ways), thus giving rise to more patterns.

However, more important than the overall distribution of the different patterns is whether patterns for all different situations could be identified. If we use a combination of characteristics as a basis for such identification, it immediately becomes impossible, as the number of resulting combinations would be significantly larger than our set of patterns. If we look only at the individual values, we see that we could identify patterns that instantiate all attribute values except for PAID Attributes: Procedures. We presume that the reason for missing this is that procedures are actually more on an organizational level than on a technical level and thus cannot be guaranteed by technical means.

Table 19: Summary of All Patterns

Attribute	Value Range	Pattern
SoS Context		
SoS Scope	System-of-Systems	History Pattern, SOA (Service-Oriented Architecture), P2P (Peer-to-Peer), Broker, Blackboard, Collaborative Virtual Environments
	System	UI Update based on Data Changes, Remote Process Invocation – Request and Reply, Batch Data Synchronization, History Pattern, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Pipes and Filters, Dynamic Router, Canonical Data Model, Remote Façade
Development Context	Greenfield	UI Update based on Data Changes, Remote Process Invocation – Request and Reply, Batch Data Synchronization, History Pattern, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Pipes and Filters, Dynamic Router, Canonical Data Model, Collaborative Virtual Environments
	Brownfield	Remote Façade, Remote Process Invocation – Request and Reply, Batch Data Synchronization, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), History Pattern, Broker, Blackboard, Pipes and Filters, Remote Façade, Collaborative Virtual Environments
	Closed Source	Batch Data Synchronization, Remote Façade, Remote Process Invocation – Request and Reply, SOA

Attribute	Value Range	Pattern
Integration Purpose	Information Exchange	Remote Process Invocation – Request and Reply, Batch Data Synchronization, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Pipes and Filters, Dynamic Router, Canonical Data Model, Remote Façade, Collaborative Virtual Environments
	Negotiation	Publish-Subscribe, P2P (Peer-to-Peer), Broker
	Updating	UI Update based on Data Changes
	Track Data Evolution	History Pattern
	Collaboration	SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker
	Decision Making	Blackboard
LISI	Connected	Batch Data Synchronization
	Functional	UI Update based on Data Changes, Pipes and Filters, Dynamic Router, Remote Façade
	Domain	Remote Process Invocation – Request and Reply, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Pipes and Filters, Dynamic Router, Canonical Data Model, Remote Façade
	Enterprise	History Pattern, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Collaborative Virtual Environments
PAID Attributes	Procedures	
	Applications	Remote Process Invocation – Request and Reply, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Remote Façade, Collaborative Virtual Environments
	Infrastructure	UI Update based on Data Changes?, History Pattern, SOA (Service-Oriented Architecture), P2P (Peer-to-Peer), Pipes and Filters, Dynamic Router

Attribute	Value Range	Pattern
	Data	UI Update based on Data Changes, Batch Data Synchronization, Publish-Subscribe, History Pattern, Broker, Blackboard, Pipes and Filters, Dynamic Router, Canonical Data Model
Technical Categorization		
Integration Level	Information Exchange	UI Update based on Data Changes, Batch Data Synchronization, History Pattern, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Pipes and Filters, Dynamic Router, Canonical Data Model
	Basic Behavior interaction	Remote Process Invocation – Request and Reply, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Pipes and Filters, Dynamic Router, Remote Façade
	Complex Behavioral Interaction	SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard
	User Interface Sharing	Collaborative Virtual Environments
Data Abstraction Level	Structural	Batch Data Synchronization, Pipes and Filters, Dynamic Router, Remote Façade
	Syntactic	UI Update based on Data Changes?, Remote Process Invocation – Request and Reply, History Pattern, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker
	Semantic	(History Pattern), SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Canonical Data Model, Collaborative Virtual Environments
Data Level Integration	Message-Exchange	Remote Process Invocation – Request and Reply, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Broker, Blackboard, Dynamic Router, Collaborative Virtual Environments, Canonical Data Model, Remote Façade
	Streaming	Pipes and Filters, UI Update based on Data Changes, Collaborative Virtual Environments

Attribute	Value Range	Pattern
	Common Data	History Pattern
	File-Transfer	Batch Data Synchronization
Interaction Style	Send	UI Update based on Data Changes, SOA (Service-Oriented Architecture), Publish-Subscribe, P2P (Peer-to-Peer), Pipes and Filters, Dynamic Router, Canonical Data Model
	Call	P2P (Peer-to-Peer), Remote Façade
	Call-Return	Remote Process Invocation – Request and Reply, History Pattern, SOA (Service-Oriented Architecture), Broker, Remote Façade
	Time-Based	Batch Data Synchronization
	Call/Call-Back	Broker, Remote Process Invocation – Request and Reply
	Multi-Call Protocols	Blackboard
Quality Attributes of Integration	*Reliability*	Remote Process Invocation – Request and Reply
	Performance	SOA (Service-Oriented Architecture), P2P (Peer-to-Peer)
	Security	SOA (Service-Oriented Architecture), Collaborative Virtual Environments
	Availability	SOA (Service-Oriented Architecture), P2P (Peer-to-Peer), Collaborative Virtual Environments
	Scalability	Blackboard, Pipes and Filters, Dynamic Router
	Manageability	Dynamic Router, Pipes and Filters, Publish-Subscribe
	Interoperability	Blackboard, Broker, Pipes and Filters, Publish-Subscribe, SOA (Service-Oriented Architecture)

6 Conclusions/Future Work

Any repository of patterns will be, of necessity, incomplete. Patterns reflect the experience of the broad community of software engineers over decades of building complex systems. This process of accumulating new challenges, new experience, and new solutions never stops. Therefore, any catalog of patterns will be subject to revision and reinterpretation in light of new knowledge.

Our goal in this report was to provide a classification that could, at a later time, be used to develop such a catalog. In this sense, our goal here is to present a way of thinking about and evaluating SoS patterns for integration, and to present a sample set of patterns from industrial and academic sources. Assuming you agree with this premise, how can you, as an architect, make use of the work presented here? We began this report by stating that our objective was to make it easier for an architect to approach the integration problem.

Clearly, there are many "start states" for a SoS. Just as clearly, it is also never the case that an architect has a clean slate to work with; legacy will always burden a SoS and its evolution. So clearly understanding the characteristics of the legacy systems, infrastructure, and data, and understanding the intended characteristics of the goal state (the SoS that we are trying to build and evolve towards) are critical.

We see a process emerging for how to create a SoS based on patterns as a central, organizing, architectural concept. As a start state, we assume that the architect has one or more independent systems that must be integrated, or that must be integrated into an existing SoS. To achieve this, the architect can make use of patterns. But which patterns to use? We suggest the following set of steps:

1. Choose/prioritize architecturally significant requirements, focusing on the driving quality attributes [Bass 2012].
2. Catalog the technological, budget, schedule, and organizational constraints that will limit the set of possible architectural options.
3. Choose data and control strategies that reflect the requirements of the interoperating systems.
4. Based on the requirements, constraints, and data/control strategies, choose one or more patterns.
5. Tailor and combine any chosen patterns.
6. Finally, choose technologies to implement and realize these patterns [Cervantes 2013]. Enterprise frameworks, for example, will realize many of the patterns described in this report at relatively low cost to the project.

Steps 4 and 5 are, of course, the most fundamental and most challenging of these steps. And these steps are the most harmful if the architect makes poor choices. So let us say a few more words about how to achieve these crucial steps. Buschmann et al. [Buschmann 2007], in their discussion of patterns and pattern languages for distributed systems, recommend starting with patterns that take one from the "Mud" of requirements and constraints to a set of primary structures that partition and organize the architecture. These could be patterns such as Domain Model, a set of Layers, Pipes and Filters, a Shared Repository, and so forth.

Typically SoS systems engineers begin architecture design with a functional viewpoint (sometimes called a “behavioral” or “horizontal” view). This viewpoint describes how SoS constituent systems and subsystems should interact at runtime to achieve SoS capabilities. The SoS capabilities are described by “kill chains”, business or mission threads, or similar scenario-based approaches. This functional viewpoint is the most common starting point for SoS architecture (e.g. see [Dahmann 2011]). Once the “Mud to Structure” patterns have been chosen, the next choice for the architect is how the individual systems within the SoS interact. These patterns—called “Distribution Infrastructure” by Buschmann et al.—are the main concerns of this report. These describe the primary means of interaction among the constituent systems and include patterns such as Client-Server, Peer-to-Peer, Broker, Messaging, and Publish-Subscribe.

At this point other patterns are typically chosen to augment these fundamental ones—for database access, synchronization, concurrency, event handling, extension, and so forth. In this way, the SoS architect can move from a largely functional description of the SoS to one that is instantiated with patterns, and hence analyzable and implementable.

Creating a successful SoS—that meets the needs of its stakeholders today and can evolve and scale to sustain those stakeholders into the future—is one of the most complex engineering challenges facing the software world. The patterns we have presented here are, of course, not the whole solution. But they do present the distilled learning from many thousands of projects that have gone before, and so they offer a solid foundation on which to build.

Appendix: Levels of Information System Interoperability

The Levels of Information System Interoperability (LISI) is described by Morris and colleagues as a widely recognized model for system-of-systems interoperability [Morris 2004, p. 5]. It comprises the levels shown in Figure 1. The description of the levels is also based on work by Morris and colleagues [Morris 2004, p.5].

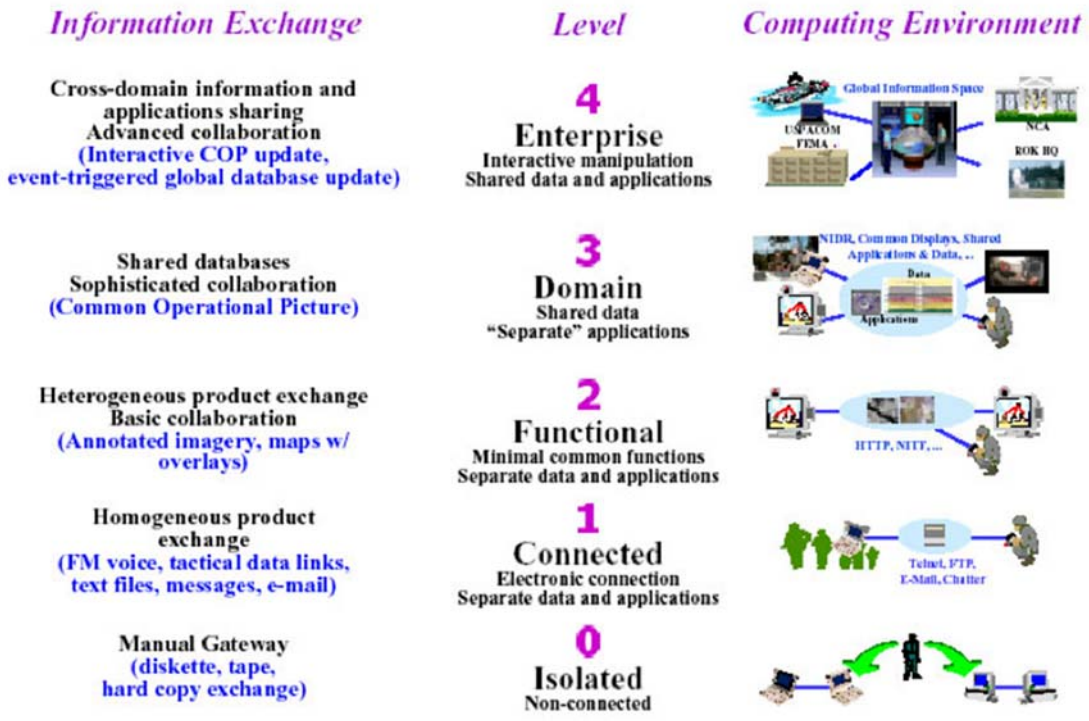

Figure 1: LISI Levels

Level 0 – Isolated *interoperability in a manual environment between stand-alone systems: Interoperability at this level consists of the manual extraction and integration of data from multiple systems. This is sometimes called "sneaker-net."*

Level 1 – Connected *interoperability in a peer-to-peer environment: This level relies on electronic links with some form of simple electronic exchange of data. Simple, homogeneous data types, such as voice, text email, and graphics (e.g., Graphic Interface Format files) are shared. There is little capacity to fuse information.*

Level 2 – Functional *interoperability in a distributed environment: Systems reside on local area networks that allow data to be passed from system to system. This level provides for increasingly complex media exchanges. Logical data models are shared across systems. Data is generally heterogeneous—containing information from many simple formats fused together (e.g., images with annotations).*

Level 3 – Domain-based *interoperability in an integrated environment: Systems are connected via wide area networks. Information is exchanged between independent applications using shared*

domain-based data models. This level enables common business rules and processes as well as direct database-to-database interactions. It also supports group collaboration on fused information.

Level 4 – Enterprise-based *interoperability in a universal environment: Systems are capable of using a global information space across multiple domains. Multiple users can access complex data simultaneously. Data and applications are fully shared and distributed. Advanced forms of collaboration are possible. Data has a common interpretation regardless of format.*

Understanding in advance, before building the actual systems, what level of integration must be achieved is crucial to identifying appropriate patterns, as different patterns will address only certain levels of integration.

The *PAID attributes* are part of the LISI model [Morris 2004, p.6]. They further refine the level of integration by describing which aspect of integration is supported on the corresponding level [BMP 2012]. The following attributes are distinguished:

Procedures: describes the degree to which overall procedures and governance are integrated

Applications: describes the degree to which the actual applications are integrated from single processes to applications suites

Infrastructure: defines the infrastructure components that support the integration

Data: describes the range of data formats and standards that support interoperability. Again, the degree of data integration varies across the different LISI levels.

While applications, infrastructure, and data are of technical nature and hence supported by technical patterns, procedures belong more closely to the organizational realm and must be supported on this level.

References/Bibliography

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

[Bass 2012]
Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice*. Addison-Wesley, 2012.

[BMP 2012]
Best Manufacturing Practices, Center of Excellence. *LISI Model: Levels of Information Systems Interoperability (LISI) Reference Model*. "The PAID Paradigm." http://www.bmpcoe.org/library/books/lisi%20model/32.html (Accessed March 14, 2013).

[Buschmann 2007]
Buschmann, Frank; Henney, Kevlin; & Schmidt, Douglas C. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*. Wiley & Sons, 2007.

[Cervantes 2013]
Cervantes, Humberto; Velasco-Elizondo, Perla; & Kazman, Rick. "A Principled Way to Use Frameworks in Architecture Design." *IEEE Software*, Mar/Apr. 2013.

[COE 2010]
Chief Information Office, United States Army. *Common Operating Environment Architecture, Appendix C to Guidance for 'End State' Army Enterprise Network Architecture.* Technical Report, U.S. Army CIO/G-6. October 2010.

[Churchill 2001]
Churchill, Elizabeth F.; Snowdon, David N.; & Munro, Alan J. *Collaborative Virtual Environments: Digital Places and Spaces for Interaction*, Springer-Verlag, 2001.

[Dahmann 2011]
Dahmann, Judith; Rebovich, George; Lowry, Ralph; Lane, JoAnn; & Baldwin, Kristen "An implementers' view of systems engineering for systems of systems", *IEEE International Systems Conference (SysCon '11)*, pp. 212 -217, 2011.

[DoD 2012]
Department of Defense. *Systems Engineering Guide for Systems of Systems, version 1.0*. 2008. www.acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf

[Fowler 2003]
Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003

[Hohpe 2003]
Hohpe, Gregor & Woolf, Bobby, United States Army. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003.

[Köppen 2011]
Köppen, Veit; Brüggemann, Björn; & Berendt, Bettina. “Designing Data Integration: The ETL Pattern Approach.” *Business Intelligence 12*, 3, pp. 39–55, 2011.

[Maier 1996]
Maier, Mark. “Architecting Principles for Systems-of-Systems” 567–574. *INCOSE 1996 Sixth annual International Symposium of the International Council on Systems Engineering*, Boston, MA, June 1996.

[Morris 2004]
Morris, Edwin; Levine, Linda; Meyers, Craig; Place, Pat; & Plakosh, Dan. *System of Systems Interoperability (SOSI): Final Report* (CMU/SEI-2004-TR-004). Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/library/abstracts/reports/04tr004.cfm

[Sal 2012]
Salesforce.com. “Integration Patterns and Practices,” version 26.0. 2013. http://www.salesforce.com/us/developer/docs/integration_patterns/integration_patterns_and_practices.pdf

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2013	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Understanding Patterns for System-of-Systems Integration	FA8721-05-C-0003

6. AUTHOR(S)

Rick Kazman, Claus Nielsen, Klaus Schmid

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2013-TR-017

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

Creating a successful system of systems—one that meets the needs of its stakeholders today and can evolve and scale to sustain those stakeholders into the future—is a very complex engineering challenge. In a system of systems (SoS), one of the biggest challenges is in achieving cooperation among systems through some form of system integration. Previous work has approached the integration challenge in a generic way, not specific to a SoS context, or has provided only a limited range of solutions. This technical report discusses how an architect can address the SoS integration challenge from an architectural perspective; it also illustrates the breadth of potential solutions to the challenge through a categorization of SoS patterns. To demonstrate the practical relevance of this work, the authors instantiate this categorization with a set of patterns described in both the research literature and by companies that support SoS platforms.

14. SUBJECT TERMS	15. NUMBER OF PAGES
pattern, integration, system of systems, software architecture	55

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