Some Current Approaches to Interoperability

David Carney
David Fisher
Ed Morris
Pat Place

August 2005

Integration of Software-Intensive Systems Initiative

Technical Note
CMU/SEI-2005-TN-033

Contents

Abstract

This technical note examines some of the complexities of interoperability and some recent research approaches to achieving it. There are many reasons why achieving interoperability between complex, heterogeneous systems is difficult. These include the problem of semantics; the differences between hardware and software; the difference between bounded and unbounded software systems; the need for trust, trustworthiness, and security in software systems; and the difficulty of quantifying interoperability. Many research efforts currently underway are aimed at finding improvements in both technologies and procedures to achieving interoperability more easily. These efforts include work in ontologies, service-oriented architectures, emergent methods, and new approaches to security. While these efforts show many signs of promise, a considerable amount of work will be needed to bring these to a mature state.

1 Introduction

In this paper, we consider relationships between multiple software systems, specifically, those relationships that produce cooperation between these systems. This cooperation is generally called interoperability. We examine some of the complexities of interoperability, and some recent research approaches to achieving it. As a preface, we first set out our initial understanding of what interoperability is, together with some of its necessary characteristics.

1.1 Interoperation as a Relationship

The term *interoperability* has many definitions; a reasonable one is

> The ability of a collection of communicating entities to (a) share specified information and (b) operate on that information according to a shared operational semantics in order to achieve a specified purpose in a given context.[1]

The essence of interoperation is that it is a **relationship** between **systems**, where systems are the entities in the above definition. While our focus will be on computer-based systems, the definition extends beyond the world of mechanical systems to organizational and other contexts. To interoperate one system must provide a service[2] that is used by another. This cannot be achieved without, at a minimum, communication from the provider to the consumer of the service.

Interoperability relationships **necessarily** involve communication. Just as in the physical world a relationship of proximity may not involve interoperability (e.g., the table is **close to** the chair), a proximity relationship in the software domain may not involve interoperation. For instance, the mere fact that two software systems are both installed on a single machine does not imply that they are interoperable (though they might, of course, be interoperable by some other relationship).

1.2 Changing Demands on Interoperability

While there are many ways that multiple, heterogeneous systems can interoperate, we posit two of the most important, which we term *design-time* interoperability and *run-time* interoperability.

[1] Carney, D.; Smith, J.; & Place, P. *Topics in Interoperability: Infrastructure Replacement in a System of Systems* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. To be published.

[2] While it seems obvious, it must be stated that provision of service includes the provision of data.

In the former case, it is often possible for all of the parties responsible for all components of a system of systems to agree a priori on the particulars necessary to achieve whatever level of interoperability is needed. These systems of systems tend to be closed (e.g., the various systems that make up an automobile) and amenable to control by some individual or group with full responsibility for the overall system and its interoperation. When no individual is in total control, agreement among the various system managers can be achieved before the systems are developed. We term this kind of interoperation *design-time*. (This is, for example, how the Army's software blocking policy produces such agreement.) Design-time interoperability is relatively well understood and, while not necessarily easily achieved, is well within current technological capability for many classes of systems.

A very different notion of interoperability, which we term run-time interoperability, is less well understood. In this kind of system of systems, now imagined for "net-centric operations" (NCO), it is assumed that the constituent systems will be able to support ever-changing demands for service; to meet those demands, the components will continually adapt to new operational contexts. Because the operational context is changing continuously, the developers of those systems cannot know a priori the systems with which they will interoperate. The result is that the difficulty of reaching agreement between developers has been magnified, since agreement can only be reached **after** the systems have been developed. In essence, interoperability becomes a **run-time** and not a design-time problem. The most significant implication of this is that, since interoperability becomes a run-time issue, it follows that no overall set of agreements can be reached, but that each system must negotiate on a pair-wise basis on the meaning of a particular communication, and do this dynamically, at run-time.

1.3 The Need for Flexibility

Design-time and run-time interoperability exhibit many differences. For instance, design-time interoperable systems achieve their necessary degree of interoperability only by means of tight programmatic control of engineering choices. This approach typically comes at high cost, and involves inflexible agreements about specific requirements (e.g., standards, data semantics, and QoS), very close interaction between the organizations responsible for the systems, and very extensive testing to verify the specific interoperable pathways. The resulting integrations are commonly too inflexible to permit introduction of any new elements into the systems. Also, maintaining such interoperability has its own level of difficulty as system versions change and evolve. More significant to end users, these inflexible integrations limit the users' ability to form ad hoc, creative solutions when necessary.

By contrast, we can consider several tactics employed by U.S. Armed Forces in Afghanistan in 2001 that manifest run-time interoperability: B-52's were used to provide on-call close air support; F-18's were used to support cavalry charges; and Predators provided real-time video to gunships. In Iraq, soldiers without military issue radios maintained contact within convoys by using commercial walkie-talkies; they also used commercial Global Positioning System positioners to create their own maps of Iraqi desert roads [Davis 03]. None of these solutions were the result of tightly integrated systems of systems that had been predefined by Pentagon planners. Rather, these solutions resulted from creative solutions developed to address situational needs.

What these examples suggest is that, to meet changing business or battlefield demands, users want integrated **solutions,** not integrated **systems**. Users will cobble together any combination of doctrine, organization, computing and other material capability to provide an integrated solution to their problem. The type of tightly coupled systems of systems described above, which is what too many users currently have available, often frustrate their efforts.

Thus, the goal for planners and developers of future computing capabilities is to find ways to support interoperability between components while maintaining the flexibility to construct new, creative solutions.

This technical note discusses several conceptual issues that affect our understanding of the task of achieving interoperability, and provides short overviews of some new software approaches that are potential solutions. In Section 2 we posit some of the factors that complicate the problem of system interoperation. In Section 3 we examine some of the more promising research efforts currently underway. Section 4 is a brief summary of the paper.

2 Some Complicating Factors

In this section, we consider a number of factors that make achieving interoperability difficult in today's software-intensive systems of systems. Of the many complicating factors that exist, we posit the following as exemplars:

- the problem of semantics
- the differences between hardware and software
- the difference between bounded and unbounded software systems
- the need for trust, trustworthiness, and security in software systems

While we shall discuss these factors separately, significant interrelationships will become apparent. The first factor, that of semantics, permeates every other aspect of interoperability. Then, after noting that a key difference between software and hardware systems is that software boundaries are far more fluid than hardware, we shall see that this concept leads directly to considering the differences between bounded and unbounded systems. Finally, even conjecturing about unbounded systems necessitates considering how security and trust can operate in such a context.

2.1 The Problem of Semantics

Interoperability depends to a large extent on common understanding. For two systems to interoperate, hardware pins must align, communication protocols must be consistent, data formats and structure must be understandable, system invocation mechanisms must be shared, and so forth. Yet even with all of the things in place to assure connectivity, there is still no guarantee that either system will be able to the convert signals, bits, and bytes into the information necessary to perform its requisite tasks. Both systems must also make consistent interpretations on the **meaning** of the data communicated between them; they must exhibit semantic interoperability.

As a trivial example, suppose one system sends the number "5" to another system. What does that communication mean? The answer is that its meaning depends on both systems having agreed that "5" represents a high-priority risk, or that it represents the fifth day of the week, or some other such meaning. In other words, we need to relate the communicated data itself to the **meaning** of that data. We may, therefore, informally define semantics as the implied meaning of data, providing a way to establish what entities mean with respect to their roles in a system.

There is a limited number of ways that agreements on meaning can be achieved. In the context of design-time interoperability, semantic agreements are reached in the same manner as interface agreements between the constituent systems. If the system of systems is a closed system, then the context of those agreements is only that of the system of systems; there is no need for any other entity to share in the agreements, or to understand the implied meaning of the data. However, in the context of run-time interoperability, the situation is more complex, since there is need for some manner of universal agreement, so that a new system can join, ad hoc, some other group of systems. The new system must be able to usefully share data and meaning with those other systems, and those other systems must be able to share data and meaning from an unfamiliar newcomer.

One mechanism often mentioned to solve this problem is the use of standards, to which all systems adhere, and which govern all interactions, whether planned or otherwise. If this goal could be achieved, then the standards would provide a third-party locus for agreements: system A follows standard Y, and system B also follows standard Y, hence interconnections between system A and B are guaranteed to succeed, even if they have been designed and built in complete isolation from each other.

This is, in fact, the situation that occurs at the lower levels of software interconnection. Standards define the parameters of the physical components, making it possible to connect one hardware device to another. Similarly, standards define protocols for communication so that data can be successfully passed from one system to another.

But at the higher levels of meaning, gaining consensus on such “universal” standards has proven remarkably difficult to achieve. For one thing, as systems deal more and more with meanings, namely, with the complexities of “information” as opposed to raw data, the ambiguity of human semantics enters in. Thus, even in human communication, misunderstandings arise. It is not uncommon for two people to believe that they fully understand each other’s words, yet their understandings are different. This ambiguity does not disappear when the communication takes place through software systems. Another problem with standards is that, even when efforts have been made to gain universal and standardized agreements on some useful subset of information, the rapid march of technological change has been much faster than the pace at which these standardized agreements can be implemented in systems. Hence, many such agreements have been obsolete even before they were achieved.

One way proposed for sharing semantic agreements has been the use of mathematics. But even in the purely mathematical representations of semantics, the problem is quite difficult. For instance, if one system represents its meaning using set theory and another uses a process algebra, it is unlikely that those two systems can communicate their semantics to each other dynamically. (Indeed, mathematics-based semantic agreements are difficult to resolve even at design time.)

2.2 Differences between Hardware and Software

There has been considerable argument over the degree to which software is genuinely different from hardware; this argument typically finds expression in a parallel argument about software engineering vs. system engineering. The long-standing view is that software engineering is less disciplined than the system engineering commonly found in hardware projects. Critics of software engineering point to this supposed lack of discipline as a leading cause of failure for many software-intensive systems. This perspective is supported by the degree of success that has been achieved by applying more disciplined approaches from system engineering to software engineering activities (most notably the Capability Maturity Model®(CMM®) and its later instantiation, the Capability Maturity Model Integration (CMMI®).

We will not argue whether this impression is fair or not. However, we do assert that at least part of the solution lies in the realization that software genuinely **is** different from hardware:

- The potential rate of change for software components vastly exceeds that for hardware components. This flexibility is a direct result of software's malleability; software is easier and cheaper to change, and it requires no retooling of production machinery.
- Hardware interfaces, being observable, are easier to identify; they are also apt to be less complex than software interfaces.
- The boundaries between software components are not as easy to define and are more fluid than those between hardware components.
- Hardware components tend to be better isolated from other components. Hence, changes to software components tend to have more widely cascading effects on other components, due to their greater interdependence.
- Quality of service (QoS) for hardware components is better understood, in terms of what is required, which component can deliver it, and how it is measured. Hardware engineers also have a better understanding of how to increase performance for a specific quality of service.

Most of the attempts to improve software engineering assume the superiority of the system engineering common for hardware projects. These attempts usually consist of strategies that target areas such as requirements management, design, standards, and management processes, and seek to apply techniques successful in hardware to the software engineering domain.

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office.

However, the gains achieved in this manner have been uneven, and require tremendous and disciplined effort to sustain, particularly as the proportion of software increases in most systems. And the problem is greatly intensified when several independently developed software systems must cooperate—the normative condition when several software systems interoperate. When no one controls the whole, the best efforts at hardware-derived discipline very often fail.

The typical perceived solution is to apply still more hardware and system engineering discipline to gain better control of the changes to the various components. But there is an alternate perspective. It speculates that ultimately **all** traditional approaches to managing software complexity will fail in large-scale systems of systems. The failure will occur precisely because the techniques try to counteract the features of software that give it its power, for instance, its malleability and flexibility. And as these techniques apply ever-increasing discipline and coordination, the engineering problems will become ever more resistant to solution, as software, true to its inherent nature, becomes used in ever more complex and unanticipated ways.

Proponents of this alternate perspective maintain that the way to achieve and sustain the interconnected system of systems that are in such constant demand is to use the inherent characteristics of software (e.g., its flexibility) as a key to the solution, rather than something to be tamed. They suggest that research efforts must begin to determine just how the traditional hardware-derived approach breaks down, and then to modulate traditional engineering techniques with new techniques better fitted to the real challenges at hand.

2.3 Bounded vs. Unbounded Systems

Engineers often make the assumption that the requirements for a system are completely knowable. We refer to such systems, developed with complete knowledge of the expectations and actions of all participating components, as "bounded." Bounded systems typically rely on effective mechanisms involving centralized control, centralized data, or hierarchical structures both in the development and execution of the system in order to provide the required degree of trust.

The degree to which software systems have ever been truly bounded is debatable. Software systems (and software engineers) have, as described in the previous section, been plagued by a very large number of spectacular failures, many of which relate to unclear requirements. But whatever the truth for the past, it is a virtual certainty for the future that software engineers will probably never have complete knowledge of the expectations and actions for the software systems of systems they will build.

Consider, for instance, one growing phenomenon: many systems of systems now employ their components in ways that were neither intended nor anticipated. Such is clearly the case regarding the Internet, where the numbers of participants and the quality of the information they provide is often unknown. It is also the case in complex systems of systems such as command and control, air traffic control, electric power grid, individual aircraft, and modern PC operating systems. For example, in recent battlefield encounters, agility and rapid progress were achieved by direct and unplanned interactions among ground troops, helicopters, artillery, and bombers, all using equipment whose designs did not anticipate the ad hoc manner of use.

The massively interconnected systems now imagined, such as the semantic Web, or the DoD's net-centric operations are likely to contain even more component parts and be even more dynamic, with participants and components almost constantly changing [W3C 01, Cebrowski 88]. These systems of systems are *unbounded*, because they involve an unknown number of participating systems. They require individual systems to act and interact in unanticipated ways, often in the absence of complete information. It is not possible a priori to understand all of the ways in which the elements in unbounded systems will behave. Unlike bounded automated systems, where neither correct nor useful results can be computed in the absence of complete and correct data, unbounded systems must function effectively with incomplete data and with data that cannot be fully trusted.

It is in this context that the need for deep and rich semantic underpinnings is so vital. Making an assumption that a new component or system can join a network of existing systems is also making the assumption that there will be some significant level of understanding between the newcomer and the existing systems; that understanding must be more than just ASCII or HTML.

Given that few of today's systems (and systems of systems) are based on a rich and shared semantics, and few exhibit the kind of flexibility required, making that assumption and achieving that scenario can seldom succeed. There are many reasons for this. For instance, the primary mechanisms for reducing error, compromise, and failure, and for achieving data integrity and trust in closed, tightly coupled, and fully understood systems are far less effective for unbounded systems of systems. Another reason is that few owners of such bounded systems are willing to open their systems to any other system whose provenance is unknown. This is because today's systems are highly vulnerable. Centralized data and control create a single-point target for attacks, accidents, and other failures. They also create communications vulnerabilities by increasing communication delay, transaction time, and ultimately user response times. If the success of the system of systems depends on the success of each of its components and subsystems, then an error, compromise, or failure in a critical central component propagates to the system as a whole and undermines enterprise success. The unknown provenance of the unbounded system presents a high risk for such undermining activity.

2.4 Trust, Trustworthiness and Security

The issue of vulnerability described above is significant, and we consider it from the standpoint of three related notions: trust, trustworthiness, and security. All of these notions play a part in achieving interoperability, and all of them depend on deep, rich, and shared semantic understanding. Discussion of these notions is also dependent on our initial definition, where we stress that interoperability can be considered only in a given context.

In the hoped-for context of unbounded systems of systems, trust in the actions and capabilities provided by interoperating parties is essential. Each party to an interaction must have, develop, or **perceive** a sense of whether the actions of interoperating parties can be trusted. This sense of trust is not Boolean (e.g., parties can be trusted to varying degrees), is context dependent (Party A can be trusted in a particular context but not in another), and is time sensitive (Party A can be trusted for a certain period). Further, the absence of trust—distrust—is less dangerous than misplaced trust: it is better to know that you **cannot** trust a particular party than to misplace trust in a party.

Trustworthiness relates to the actual state of the end-to-end service provided (e.g., by a system of systems). Thus, misplaced trust is essentially a condition where one party perceives that a second party can be trusted, but the latter party is not trustworthy. Like trust, trustworthiness is context and time dependent, but it is not continuous (e.g., for a specific context, at a specific instance, the end-to-end capability is either trustworthy or not.).

Security addresses issues of confidentiality (information available only to those authorized), integrity (information not corrupted due to unauthorized—by error or intent—change), and availability (information not erased or inaccessible) [CERT 97]. Security concerns are commonly addressed through either policies or technologies:

- security policies identifying risks and threats, guidelines and security practices for system management and for legitimate use, and guidelines for reacting to compromises in security
- security technologies to minimize or detect intrusion or to limit the damage, such as one-use password technologies, firewalls, monitoring tools, security analysis tools, and encryption

Whereas security is concerned with preventing unauthorized and accidental use, corruption, and blocking of access, trust and trustworthiness are concerned with other factors and the overall system behaving as expected. Thus, it is entirely possible for a highly secure and error-free actor providing information over an equally secure network to be untrustworthy for a particular need. For example, the data provided by a secure radar device with a slow sweep rate may be untrustworthy for fire control of an anti-missile missile system. It would be a mistake for the commander of the anti-missile system to trust the information from the radar for this purpose. The key is that interoperating systems rely not only on secure interactions, but on interactions that provide appropriate information for a given context and point in time.

Thus, the mechanisms that are useful for providing security are useful **but insufficient** for constructing trustworthy capabilities and establishing trust between components. In traditional system development, we circumvent this problem and establish trustworthiness of components by working closely with component providers and modeling and testing the end-to-end capability that is expected.

However, we expect to require greater degrees of dynamism and on-the-fly composition for future systems such as the Semantic Web and systems capable of net-centric warfare. For these sorts of systems, unanticipated uses, rapidly evolving and uneven technology, capabilities coming online and departing rapidly, changing mission needs, and potentially untrustworthy and even adversarial users are the norm. Establishing the kind of complete trustworthiness found in tightly coupled or bounded systems is highly unlikely and perhaps even impossible in these environments.

3 Some Current Proposed Solutions

3.1 Ontologies

An ontology defines the terms and relationships among terms that represent an area of knowledge. In software engineering, computer-readable ontologies are growing in importance for defining basic concepts within a domain. If multiple-domain applications are developed utilizing a shared ontology, or if their distinct ontologies can be related, then the applications can have a common understanding of data, and semantic interoperability is enhanced. In addition, ontologies can be developed that relate information across domains, opening up new possibilities for interoperability.

While offering promise for enhanced semantic interoperability by helping developers to locate relevant descriptions and allowing computers to infer relationships and properties, ontologies are hard to define well because

- Few people are expert in the representation of knowledge, and these experts are rarely the domain experts building ontologies. As a result, ontologies are often poorly constructed and hard to maintain.
- Consensus building is a hard task that can be made more difficult by the scope and diversity of the organization and domain(s), by the existence of legacy applications that encourage advocates to fight for their solutions, and by widely different intended uses of the ontology.

In addition, the long-term evolution of ontologies is a complex task. Since applications are strongly coupled to specific ontology versions, evolution will be constrained to maintain upward compatibility, unless mappings between versions are provided.

Several ontology languages have been developed, but interest is now focused on the Web Ontology Language (OWL) [W3C 04b]. OWL is a core capability leading to the semantic Web, which supports interoperation across system and organizational boundaries by providing well-defined and shared meaning to data. OWL builds on other Web standards to define ontologies that can be distributed across the Web. It is supported by a growing number of tools (see http://www.w3.org/2004/OWL/#specs) and hundreds of domain ontologies representing commercial, government, military, and academic interests have been developed (see http://www.daml.org/ontologies/).

Merging ontologies or mapping between them is also a current research topic. While a variety of approaches and tools are under investigation, almost all require significant human intervention, normally by an individual or group of individuals familiar with the ontologies to be joined.

Two additional and somewhat contradictory problems with ontology evolution have been noted: diffusion (or mission creep) and enforced orthodoxy.

Diffusion refers to a phenomenon whereby an ontology originally intended to serve one purpose is adopted and extended to serve other purposes [Musen 05]. As a result, the complexity and number of ontologies grow, becoming difficult to use for all and less useful for the original intent. The International Classification of Diseases (ICD) that forms the basis of all medical claims and reimbursements in the U.S. represents a case in point. The original classification was created in the 19th century to compare causes of death. In 1948, the World Health Organization took responsibility and added non-fatal diseases to the classification. In 1977, the ICD was expanded further to address statistics for the planning, monitoring and evaluation of health services. It is now difficult to find and add terms amid codes such as "W65.40: Drowning and submersion while in bath-tub, street and highway, while engaged in sports activity."

Enforced orthodoxy refers to avoiding change to an ontology, even when change is needed. Since the ontology represents a form of community orthodoxy, bias can develop against change. Such orthodoxy is in part practical, since large volumes of data may be encoded based on an obsolete model. Enforced orthodoxy may also hinder new ways of thinking, particularly regarding revolutionary paradigm shifts, because the new thoughts that can be constructed are limited by the language that is used.

In summary, ontologies provide a useful mechanism for sharing semantic content of data across applications or system components. They may also help to increase the flexibility with which components interoperate, but only if sufficiently broad ontologies are developed and shared, and actively managed to prevent the loss of focus or constraining of new ideas.

3.2 Service Oriented Architectures

A *service-oriented architecture* (SOA) is a software architectural paradigm that is defined by a collection of independent, self-contained services that can be accessed in a standard way. Capabilities provided by individual services can be connected to perform required processing.

A *service* is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model [Brown 02]. A service differs from an object with associated methods in that the service is normally coarser grained and tends to have a relatively small set of interfaces employing messages with a standard format, structure, and semantics. A common example of a capability that could be provided by a service is credit card validation [Lewis 04].

Proponents of SOA suggest the following advantages:

- simple standards that define the available interfaces and structure of data that is conveyed across those interfaces
- platform- and language-independent interfaces based on these standards, which allow applications to invoke services operating on any device supporting the SOA regardless of the hardware platform, operating system, or implementation language
- clear separation of service interface from implementation, allowing many service upgrades to occur without impact on service users
- message oriented communication allowing distribution across a wide area
- loose coupling between services, thus minimizing interdependencies and facilitating reuse
- mechanisms for discovery of services available and for establishing connections with services, facilitating service use[3]

The most common form of SOA is represented by Web services, which define programmatic interfaces for application to application communication across the World Wide Web. Web services use the Simple Object Access Protocol (SOAP) and Web Services Design Language (WSDL) standards to define an Extensible Markup Language (XML)-based protocol for exchanging structured information. They also define a language for describing a Web service in terms of the messages it sends and receives, along with bindings to underlying transport and network protocols [W3C 04, W3C 01]. However, it is possible to implement SOA using other protocols, languages, and technologies.

The hallmark of SOA is flexibility. Computing platforms and languages can vary; services can be accessed across a network via simple, well-defined interfaces, and without concern for side effects resulting from dependencies between services. These factors allow applications to use (or be composed of) services efficiently and effectively.[4]

However, SOA in isolation does little to guarantee interoperability. For interoperability to be achieved by SOA, additional capabilities needed include

- mechanisms for conveying additional semantic information about services such as behavior, QoS and expected preconditions and post conditions. Currently, we do not have good ways of representing this information such that a user of the service could efficiently and reliably determine whether the service provides appropriate capability for a given context.

---

3 Some experts do not include discovery mechanisms in core SOA capabilities. In fact, it is possible to create an SOA that does not have an online discovery capability (e.g., no searchable database of available services).

4 Our experiments at the SEI have not convinced us that SOA is very quick and relatively simple for engineers to use when building applications. See Lewis [Lewis 04].

- mechanisms for conveying semantics of data required by and shared by a service. Ontologies provide a good starting point, but new techniques are needed to map between the different ontologies that are likely with independently developed services.
- ways of achieving optimal and predictable performance and other QoS expectations for the end to end capability provided by sequences of services and other application components
- ways of constructing services that have wide application to avoid proliferation of similar, but slightly different, services

3.3 Emergent Properties

Emergent properties are those properties of a whole that are different from, and not predictable from, the cumulative properties of the entities that make up the whole. The concept of emergent properties becomes increasingly important as the number and type of "actors" in a system of systems increase. Thus, large-scale networks such as the Internet (and in the future, networks that support net-centric warfare) are likely to experience emergent properties. Such networks are composed of large numbers of widely varied components (hosts, routers, links, users, etc.) that interact in complex ways.

Of necessity, each participant in such real-world systems (both the actor in the network and the engineer who constructed it) acts primarily in his or her own best interest. As a result, perceptions of system-wide requirements are interpreted and implemented differently by various participants, and local needs often conflict with overall system goals. Although collective behavior is governed by control structures (e.g., in the case of the networks, network protocols), central control can never be fully effective in managing complex, large-scale, distributed, or networked systems.

The net effect is that the global properties, capabilities, and services of the system as a whole emerge from the cumulative effects of the actions and interactions of the individual participants propagated throughout the system. The resulting collective behavior of the complex network shows emergent properties that arise out of the interactions among the participants.

The effect of emergent properties can be profound. In the best cases, the properties can provide unanticipated benefits to users. In the worst cases, emergent properties can detract from overall capability. In all cases, emergent properties make predictions about behavior such as reliability, performance, and security suspect. This is potentially the greatest risk to wide-scale networked systems of systems. Any long-term solution must involve better understanding and managing of emergent properties.

Recent research in the area of emergent algorithms has begun to identify, develop, and refine the methods first developed for other sorts of systems to solve problems of interoperability

[Davis 03]. These methods and techniques are derived by analogy from approaches that have been effective in social, biological, and economic systems, but are applicable to the design, implementation, and evolution of software in a systems-of-systems context.

At their most fundamental level, emergent algorithms provide an alternative to those approaches that achieve interoperability through ever tighter control of engineering processes and technology choices. Emergent algorithms exploit cascading effects of loosely coupled, dynamically changing, and partially trusted neighbors to achieve a common purpose shared by a subset of the participants.

Only a limited repertoire of emergent algorithms has been identified, and they are only partially understood. The methods of emergent algorithms as they apply to interoperability include cooperation without coordination, dynamic adaptation, continuous trust validation, dynamic capability assessment, opportunistic actions, anticipatory neighbor assistance, encouragement and influence, perturbation, and survivable architectures.

3.4 Potential New Approaches to Security

For environments such as the semantic Web or net-centric computing, entirely new ways of establishing adequate trustworthiness and developing trust must be created. Potential mechanisms that can establish trustworthiness within this essentially untrustworthy environment can be placed in three very broad categories:

1. approaches that establish trust through a trusted third party, such as a certificate authority for public-key certificates. This approach is proven in security application, but it is questionable whether the approach could “keep up” with changing components and expectations in a highly dynamic environment. The approach is also subject to calamity resulting from failure of the central trust authorities.
2. approaches based on networks of members who incorporate trust information into modeling of relationships for a small number of other members. These trust webs can then be composed into trust relationships for all members. This approach, like the previous approach, requires research in defining the semantics of trust and trusting relationships, models for computing and manipulation of trust, and algorithms for quickly building and updating trust webs.
3. approaches that essentially mimic the swarming behavior of ants. For example, software “ants” deposit a cue on a search trip for a capability and modify that cue on the return trip if the correct capability is found. Other ants can then “swarm”—follow that cue to the goal. This allows rapid dissemination of information about trustworthy and untrustworthy actors by employing very simple, locally implemented rules. Such techniques are promising, particularly for highly volatile network environments, but are relatively new and unproven outside of network routing application.

There is a significant volume of research aimed at developing and maturing each of these approaches. It is entirely possible that all these approaches will find their way into use in varying circumstances, or even as complementary ways of varying trust and supporting interoperability.

4 Summary

Ontologies, SOA, emergent algorithms, and novel approaches to security are providing significant opportunities to improve the degree and flexibility of interoperability that can be achieved. Alone, each technology is limited. In combination, these technologies have the potential to address many problems of data sharing, application construction, and managing the cumulative effects of the actions and interactions of diverse and varying system components. Before they reach that potential, several questions must be answered:

- Can accepted ontologies be established and maintained by communities of interest, and can techniques and tools be built to map between ontologies?
- Can the community quantify the characteristics of applications and services that affect semantic interoperability such that engineers will trust SOA services in demanding applications?
- Will emergent algorithms be developed that establish control of our increasingly unbounded systems?
- Can sufficient security and trust be found in massively connected systems of systems?

Changes in technology alone will not be sufficient to drive the shift from integrated systems to supporting integrated solutions. Individual program offices that are building individual systems will continue to be a barrier to integrated solutions to the extent that each considers its program distinct from others. In addition, even if program offices begin to build capabilities that are flexible and can be integrated to the extent that system boundaries are no longer evident, end users will continue to be thwarted in their desire for integrated solutions unless the organizations they represent and the doctrine they employ become equally flexible.

Bibliography

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

[Brown 02]	Brown, A., Johnston, S; & Kelly, K. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications*. White Plains, NY: Rational Software Corporation, IBM, 2002.
[Cebrowski 88]	Cebrowski, A. & Garstka, J. *Network-Centric Warfare: Its Origins and Future.* U.S. Naval Institute, Proceedings, 1988. http://www.usni.org/Proceedings/Articles98 /PROcebrowski.htm.
[CERT 97]	CERT Coordination Center, http://www.cert.org /encyc_article/tocencyc.html#Overview "Security of the Internet,"231-255. *The Froehlich/Kent Encyclopedia of Telecommunications vol. 15.* New York, NY: Marcel Dekker, 1997.
[Davis 03]	Davis, Joshua. "If We Run Out of Batteries This War Is Screwed." *Wired*, Issue 11.06, June 2003. http://www.wired.com/wired/archive/11.06 /battlefield_pr.html.
[Lewis 04]	Lewis, Grace A., Wrage, Lutz. *Approaches to Constructive Interoperability* (CMU/SEI-2004-TR-020). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents /04.reports/04tr020.html.
[Musen 05]	Musen, Mark A. "Building Ontologies from the Ground Up" http://lsdis.cs.uga.edu/courses/GlobalInfoSys_2005 /index.php?page=2.
[W3C 01]	World Wide Web Consortium *Semantic Web* http://www.w3.org/2001/sw/ (2001).

[W3C 01]	World Wide web Consortium *Web Services Description Language (WSDL)1.1.* http://www.w3.org/TR/wsdl (2001).
[W3C 04a]	World Wide Web Consortium *Latest SOAP Versions* http://www.w3.org/TR/soap/ (2004).
[W3C 04b]	World Wide Web Consortium. *OWL Web Ontology Language Overview* http://www.w3.org/TR/owl-features/ (2004).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE August 2005	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Some Current Approaches to Interoperability		5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) David Carney, David Fisher, Ed Morris, Pat Place		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2005-TN-033
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This technical note examines some of the complexities of interoperability and some recent research approaches to achieving it. There are many reasons why achieving interoperability between complex, heterogeneous systems is difficult. These include the problem of semantics; the differences between hardware and software; the difference between bounded and unbounded software systems; the need for trust, trustworthiness, and security in software systems; and the difficulty of quantifying interoperability. Many research efforts currently underway are aimed at finding improvements in both technologies and procedures to achieving interoperability more easily. These efforts include work in ontologies, service-oriented architectures, emergent methods, and new approaches to security. While these efforts show many signs of promise, a considerable amount of work will be needed to bring these to a mature state.

14. SUBJECT TERMS Interoperability, interoperation, integration			15. NUMBER OF PAGES 27
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL