

# Migrating Legacy Assets through SOA to Realize Network Enabled Capability

David Webster, Lu Liu, Duncan Russell, Colin Venters, Zongyang Luo, and Jie Xu

Distributed Systems and Services Group,  
School of Computing,  
University of Leeds,  
LS2 9JT,  
United Kingdom,  
D.E.Webster@leeds.ac.uk,

WWW home page: <http://www.comp.leeds.ac.uk/distsys/>

**Abstract.** Network Enabled Capability (NEC) is the UK Ministry of Defence's aspiration to enhance the achievement of military effect through the networking of future and existing military capabilities. The NEC-TISE (NEC Through Innovative Systems Engineering) program responded to this need by investigating the question '*are you ready for NEC?*' on behalf of equipment and service providers. Research work on this project proposed Service Oriented Architectures (SOA) as an architectural approach to delivering dependable and sustainable military capability. Specifically the work looked at how loosely coupled services could be used to expose and reuse functions and databases and how to describe the quality of service for heterogeneous systems and networks.

Whilst presented in a military scenario, the challenges here are applicable more generically in the web services and cloud computing domains due to the increasing trend to expose existing systems and databases as services using web technologies.

The System of Systems that NEC will be realized from will not be implemented from scratch, but rather will be migrated from legacy assets over time. These assets will provide both functionality and data/information services, for example a weather sensor. The focus of this chapter is to layout an understanding of the challenges faced and lessons learned in realizing NEC when migrating legacy assets to an SOA (Service Oriented Architecture) based System of Systems over time in order to reuse their functionality and databases. This work was based around a Software Demonstrator to illustrate a situational awareness capability realized by dynamically discovering and aggregating sensor data. The focus of this work is not specifically on sensors, however, the sensor example provides a good example of data integration to realize military capability. An abstract decision process model for wrapping legacy components was proposed to guide how existing system components can be selected for integration into the system of systems that NEC will be realized from. This model can be used to assist in the integration process of system

components when migrating to or between execution architectures. The process model provides decision support to trade-offs between the costs of reimplementation, system wrapping and those costs incurred as a consequence of System of Systems complexity and ongoing maintenance.

## 1 Introduction

Network Enabled Capability (NEC) is the UK Ministry of Defence's aspiration to enhance the achievement of military effect through the networking of future and existing military capabilities [39]. Military capability in the context of NEC has been described as "*the ability to achieve a specified 'wartime' objective*" [43]. It is important to state at this point that NEC is not a *system*, but rather is the integration of assets to fulfil a mission objective. The MOD defines NEC as "*the coherent integration of sensors, decision-makers, weapon systems and support capabilities to achieve the desired effect*" [39], therefore, the networking of communications is only part of the problem as human activities are required within this asset integration. Military capability is the whole integration of resources including people and equipment; this includes the systems that provide dependable inter-operation to support human activity. Examples of capability include: surveillance; capture and defend hilltop; and deliver and administer medical treatment in the field.

To realize NEC the Armed Forces need to be flexible, ready and rapidly deployable, with the application of controlled and precise force, to achieve realisable effects and should have the ability to support and co-operate with each other to deliver a real-time capability [35]. To be successful in achieving this goal, NEC requires system integration of independent components (both functional and data oriented) that can evolve, operate in a dependable manner, managing system and component changes, cost effectively and connecting industrial, defence and pan-defence environments. In NEC one of the challenges (amongst many) involves the through-life provision of military capability; acquisition, service and support.

The NECTISE (NEC Through Innovative Systems Engineering) program [37] responded to this need by investigating how loosely coupled services can be used to expose and reuse the functions and databases and describe the quality of service for heterogeneous systems and networks. The EPSRC and BAE Systems jointly funded NECTISE throughout its three year duration, which involved ten UK universities and aimed to address the question of how BAE Systems delivers systems to NEC to the UK MOD, taking account of the aims summarised in the 2005 Defence Industrial Strategy [38].

Systems developed by NEC oriented programs will be delivered over time and these will need to be updated rapidly as technology and ongoing capability needs develop. Systems that come out of NEC programs will possess varying timescales for their life-cycles, which, in order to provide integrated operation, need to be coordinated in some manner, be it implicit or explicit. It is generally acknowledged that some military systems will have a long service life; for example

a ship may have a service life of forty years. During this time the operational environment will evolve and the operational concepts will evolve. As noted by [32] in the military context “*many currently fielded embedded information systems face readiness challenges imposed by evolving missions and extended service life spans*”. Likewise, assets will be used within multiple capabilities throughout their active service lives.

The focus of this chapter is to layout an understanding of the challenges faced and lessons learned in realizing NEC when migrating legacy assets to an SOA (Service Oriented Architecture) based System of Systems over time. This has come from an understanding that the System of Systems that NEC will be realized from will not be implemented from scratch for the NEC initiative, but rather migrated from legacy assets over time. SOA technologies (for instance Web Services) allow data provided by legacy applications to be integrated in a manageable manner to support the realization of NEC. Whilst legacy systems have been discussed in research literature in the context of system life-cycles [15, 14, 53, 10]; the drive for this chapter is that organizations operating legacy systems face ongoing maintenance risks of ‘*brittle systems*’ [12, 32] and that can be viewed as a false economy in the long run - this is particularly the case when the characteristics of NEC are taken into consideration.

This chapter further presents our experience in examining and testing the benefits and consequences of SOA applied to the NEC environment through conceptual understanding of existing practice and a proof-of-concept SOA Software Demonstrator to simulate a situational awareness capability where data is aggregated from multiple dynamically discovered sensors in a geographical region. It should be noted at this point that the focus of this work is not specifically on sensors, however, the sensor example provides a good example of data integration to realize military capability.

The structure of the remainder of this chapter is as follows. Section 2 recapitulates current research work from NECTISE conducted towards delivering NEC through the SOA architectural style and helps to justify the decisions for why SOA is a suitable architectural approach for realizing NEC. In Section 3 the incremental delivery within the NEC System of Systems is discussed. This leads to an examination of the problems faced when implementing NEC from existing assets and discusses techniques to reuse existing assets. A decision support model for wrapping legacy system components within an SOA environment is proposed. Section 4 discusses a scenario for NEC based on sensor data integration which was used for our developed Software Demonstrator system. In Section 5 we discuss the lessons learned our research work (partially based on the Software Demonstrator) when providing a sustainable and maintainable approach to NEC provision through wrapped legacy assets. Finally in Section 6 conclusions are drawn and the potential for future work is outlined.

## 2 Service Oriented Architectures in NEC

Within NECTISE, SOA has been proposed as an architectural style for the realization of military capability to aid the engineering process for scalable System of Systems [43–45].

Engineering at the capability and product system engineering levels is defined by the MOD in [18]. *Military Capability Engineering* is concerned with the management of system of systems coherence and is defined by a high level capability architecture, critical scenarios and mission threads. On the other hand, *Product Systems Engineering* involves mortal projects that deliver capability. The key linkage between the two is described that Product Systems’ “*interoperability requirements should be derived from the system-of-system portfolio architecture to which they contribute.*” NEC will be enabled by a networked system of systems and will be provided through multiple services from multiple providers. The engineering process for systems that contribute towards capability, therefore, will be affected by requirements and changes from the capability engineering level.

### 2.1 Service Oriented Architectures

‘*Service Oriented Architecture*’ (SOA) is an architectural style that has been proposed as a way for businesses to redefine their processes to take advantage of ‘*formerly isolated component activities*’ [8] for building distributed systems that are cross-organisational. This marks the move away from building traditional ‘*stove-pipe*’ systems with fixed requirements [5] to building distributed systems from reusable and discoverable components. This proposal reflects the Defence Industrial Strategy’s statement that “*we are seeing a shift away from platform oriented programmes towards a capability-based approach*” [38].

The premise here from the above definitions is that a deployed software application component can be reused as a ‘*network-available service*’ with published and discoverable interfaces. Part of this reusability comes from the ability to clearly define the software interface in a manner and the data that passes through it abstracted away from the underlying system technology to promote loose coupling.

**Service Contracts** - A service contract in the context of SOA is a technical service contract describing the service’s interface, according to [21]. SOA contracts provide a description of technical constraints of using the service and any associated requirements. The motivation of providing a service contract is to ensure a consistent expression of service capabilities [20]. This can be viewed as the documentation of the benefits and obligations for each party in the service interaction, or as a promise from a service provider in order for a service client to be built around that promise [4].

Service contracts define the data required for service providers and clients to communicate and build upon what in the past has been known as an Application Programming Interface (API) [46]. Typically this information is described using an Interface Definition Language (IDL).

Service contracts can be described at different granularities. To give an example of this concept, describing a service with a fine granularity requires less logic processing of the data that passes through the interface than describing a service with a coarse granularity as this will map closer to internal business documents and handling logic. Describing the data granularity of services typically involves to how the input, output and fault messages of a service endpoint are described [21].

**Loose Coupling** - “*Loose Coupling*” is the degree to which software components depend on each other, according to [25] and similarly defined by [2]. as:

*“a feature of software systems that allows those systems to be linked without having knowledge of the technologies used by one another”.*

The loose coupling that SOA provides (both through the standardised communication protocols and the logical separation of systems into services) delivers an architectural approach to limit the impact of changes to services’ internal systems on other services that will each have their own internal life-cycle. SOAs provide a style of loose coupling by negating the need for providers and consumers to require access to each other’s underlying implementation details and system components (for instance, databases) of these services.

With this in mind, one of the major attributes of SOA is in the interfaces for services. The interface can be described as a logical layer abstraction of the components that provide functionality within the service [2]. This can be further clarified as a provider who offers a service to a consumer/client through the use of an interface - stating obligations of the provider and the responsibilities that the consumer must agree to. A key to the SOA paradigm in software is that services should ideally share schemas and contracts, not classes as is the case with object-oriented languages [36]. As discussed, one benefit of achieving loose coupling is to ensure that when modifications are made to a software system, the effects on local systems are reduced. One tactic to achieve this is to use ‘*deferred binding*’ by binding modules as late to run-time as possible [7].

## 2.2 Web Services

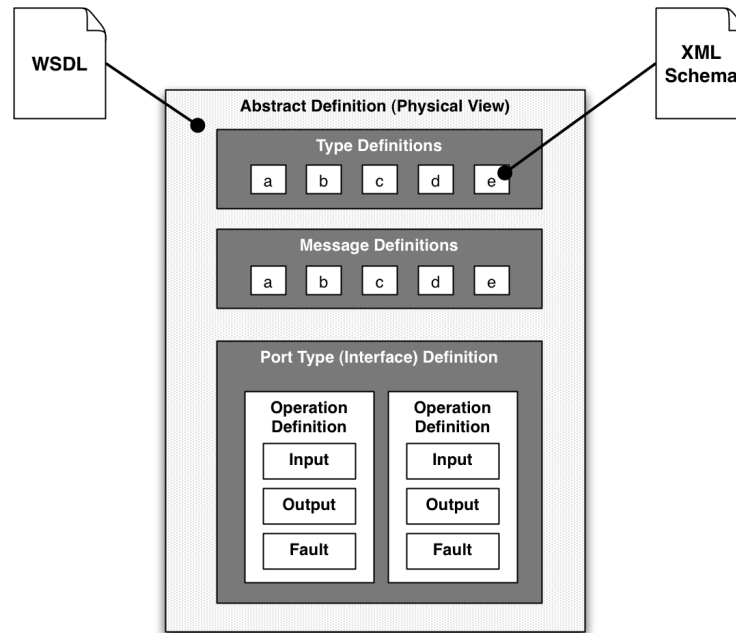
Web Services are a *de facto* set of industry standards for implementing an SOA and are used to define contracts that publicly describe a service’s operations or capabilities. Web Services can be described as:

*“self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces”*; according to [49].

To give a more technical description, Web Services are composed of a group of specifications: WSDL, XML Schema, SOAP and WS-Profile. Web Services Definition Language (WSDL) is an XML-based machine-processable interface

definition language for describing messages that are exchanged between a client and service provider [11] and is considered to be the most important of these standards according to [21]. XML Schema is a formal language used to describe and validate the structure of XML documents, for instance the WSDL document. The ‘*SOAP*’ (initially standing for Simple Object Access Protocol) protocol specification defines an XML-based stateless protocol for exchanging structured and typed information across Web peers, typically in a Web Services environment [50]. WS-Policy is a specification used for describing behaviour characteristics of a service in a machine processable manner.

**WSDL** - WSDL is composed of abstract and concrete parts [51]. The abstract part of the WSDL description describes the messages that are exchanged. This can be broken down into types and service definitions (see **Figure 1**). The concrete part of the WSDL document defines the network protocol and message format of messages sent and received and defines an endpoint which associates a network address with a binding [11, 51]. The advantage of separating the abstract from concrete part of the WSDL is that multiple service providers can provide similar services that implement the same abstract service definition.



**Fig. 1.** WSDL abstract document part and technology mapping. Adapted from [21].

### 2.3 NEC Requirements and Realizing NEC through an SOA

In the delivery of military capability enabled by networks, dynamic integration based on SOA has the following characteristics [45, 43]:

- *Service Integration* - Services are defined as composable functions, similar to component architecture [48], and can be combined to form higher levels of functionality and deliver capability.
- *Service Discovery* - Service providers offer services in a loosely coupled architecture to consumers for dynamic composition. The consumer requires discovery mechanisms to locate and bind before utilising services. For NEC, discovery is the means to identify service types for integration before forces are operational, and to enable dynamic binding in service integration during operations.
- *Service Reconfiguration* - Services can be adapted to meet consumer requirements at binding time. During service discovery, the consumer and provider may negotiate terms of service delivery involving QoS parameters. For example, using redundancy, such as replication of resources, may improve service dependability.
- *Service Evolution* - By abstracting the interface from the service implementation, a service can adapt to changes in its environment and the demands of the service consumer. Selecting appropriate resources at the time of service execution allows the resources to be updated and adapted without interrupting service availability. This supports continuous service delivery and therefore, sustainable delivery of capability.

The NEC initiative recognises that offering functionality is the main requirement in supporting military capability, and that functionality can be delivered without ownership of the delivery mechanism.

In NECTISE, design for change is being addressed by exploring architectures that can both adapt to changes and themselves can evolve within carefully defined limits. Traditional design methodologies (e.g. waterfall) address the issues of design for change by planning future changes in detail. However, modelling future changes is an extremely difficult task for complex system development. Agility in NEC represents an ability to adapt to changes occurring in through-life provision of capability not only at the design time but also at runtime. The frequent changes made in the short life-cycles could cause significant modifications of the interfaces of these military services, which lead to serious versioning and compatibility problems. Services need to be frequently re-integrated and re-configured to provide a reliable and sustainable capability. When factoring the agility of the systems that need to be evolved in order to satisfy new requirements of services then if some of these systems are legacy systems potential impedance to this redevelopment agility should be considered properly. By comparison to product delivery, service delivery is a continuous process, assuring reliability by maintaining the service provision and evolving the service implementation to respond to changes in environment, situation, supply, information and ongoing development.

SOA enables systems to operate across organizational boundaries and, therefore, contributes towards the ability for services to be discovered from different organizations and bound just before runtime [3]. This is supported by mechanisms of discovery to match services during composition. Architectural characteristics required by NEC, such as flexible interoperability and future proof evolvability, can be for the most part provided by SOA, where organisations, systems and computing each have defined service interfaces.

It should be noted, however, that services may be information services and are dedicated to providing data as opposed to purely computational services or those that control a physical resource - an example of an information service is exposing weather data or sensor data through a Web Service. The rapid growth of information services and resources in military systems makes it difficult and challenging to manage dynamic information and resources efficiently. In military systems, information is usually obtained from multiple data sources (e.g. database systems, file systems) to be integrated into a usable format for decision-making by battlefield commanders. For example, to analyse the implications of an aerial attack, background information could include data on meteorology, topography, settlements, population, infrastructure, sociology, and even hydrology. The premise of SOA, therefore, works towards standardising the mechanism by which data and information are described; for instance through XML Schema.

## 2.4 SOA and Workflows to Realize NEC

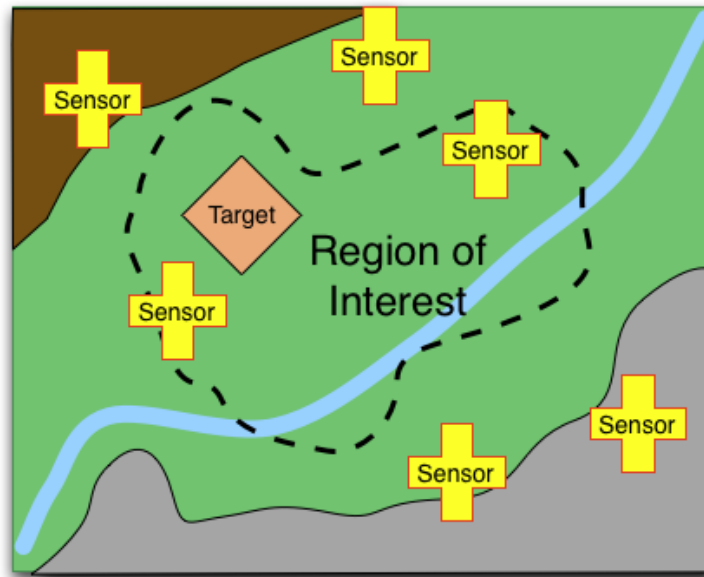
One advantage of SOA is that the functionality provided by system components can be exposed as services and reused in different contexts. A way to achieve this integration of services to realize business processes is through the use of a workflow. In the case of Web Service based SOA implementations, a standard service integration workflow technology is the Business Process Execution Language (BPEL)<sup>1</sup>.

Workflow is defined by Hollingsworth [28] as “*the computerised facilitation or automation of a business process, in whole or part.*” Business processes are descriptions of an organisation’s activities designed to fulfil a customer’s need, whereas similarly information processes are tasks performed by systems (people or computers) that process or provide information [27].

In the NEC context, as described by the UK Defence Industrial Strategy, a real-world example for capability enabled by networks is to achieve sensor data fusion through “*rationalising the masses of incoming sensor data*”. Our research work was based around a Software Demonstrator to illustrate a situational awareness capability realized by dynamically discovering and aggregating sensor data. An example SOA workflow would be the integration of data from sensors, which can be discovered at run-time using ultra-late binding. This example forms the basis of the practical Software Demonstrator described in Section 4 and was introduced briefly in our previous work [43]. **Figure 2** shows a simple

<sup>1</sup> <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>

illustrative example of a target tracking capability. In this example, the capability is to detect and track a moving target within an area of interest. This is realized by integrating sensors that have their functionality shared on the network as services. The benefit of this approach is that these sensor services can be discovered and configured both at mission planning time and at run-time. Using a workflow approach, the sensors can be discovered and then configured based on functionality and quality of service attributes. The resultant data returned from the sensors is then aggregated through a data fusion algorithm. In the illustrated example the target (which for the case of our chapter's example is an automobile) can be seen travelling through grassland valley terrain between two hills.



**Fig. 2.** High-level illustration of sensor integration scenario for target tracking.

## 2.5 An SOA Integration Model for Realizing NEC

The abstract SOA integration model for NEC has been developed through the NECTISE architectures research work. It has been presented both within the NEC and academic communities [33–35]. It helps understanding of the reuse of system/platform functionality and data flow through services to realize a capability. **Figure 3** visually represents our model for the abstraction of SOA in the provision of network-enabled military capability within NEC.

In the capability layer, new capability requirements are determined through long to medium term capability planning. In the integration layer, configurations and specifications are defined based on these requirements. Configuration defines the actual combination of services used to implement the capability. This allows the abstract concept of the capability to be defined in terms of a set of abstract specifications.

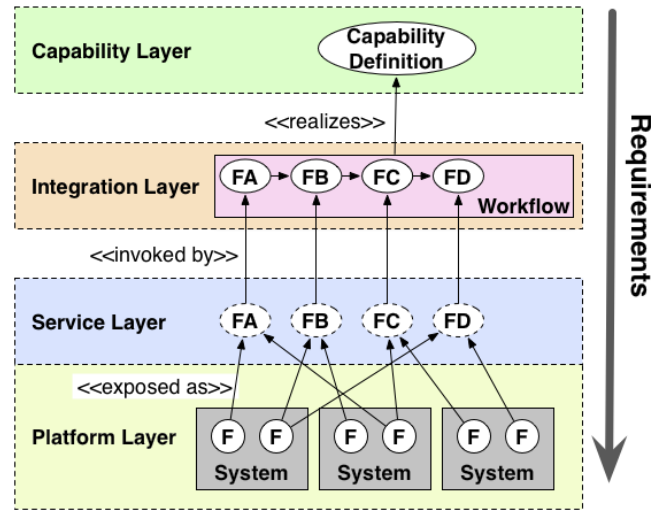
In this model, functions (*'F'*) provided by concrete platform hosted systems are exposed and abstracted as services (*'FA ... FD'*) in order to be reused. In the integration layer these services are integrated close to (or at) run-time to realize a capability; in the diagram this is shown through the use of a workflow to invoke the individual services. As discussed earlier in this chapter, workflows are one way to integrate information services and can be implemented at the integration layer. The effects from the integration within the workflow are then synchronised to implement military capability.

To give a concrete example of this integration in action (based on the scenario presented in *Section 2.4*), a capability definition is created whereby a moving target needs to be tracked across a terrain. In order to realize this capability an integration workflow is used based on an SOA architectural style. Surveillance functionality is provided by an [open] network of sensors of different types.

The integration workflow submits real-time requests to a sensor service registry that dynamically discovers sensors, retrieving attributes such as position and range. A selection algorithm determines which sensors can 'see' the region of interest. The relevant sensors are contacted, which return information about the detected points of interest. This sensor information is processed to eliminate duplicates and points outside of region of interest and the detected feature positions are displayed on a map. This sensor information along with human interpretation combines to realize a target tracking capability.

In order to realize military capability, the following life-cycle description can be applied:

1. At the Capability Management level, there is a life-cycle (for example, owned by the MOD), which defines the need for capabilities, defines them, evolves them, uses them strategically and eventually ends their life if needed.
2. Within each capability, there will be a life-cycle which takes the definition of the capability from (1) and creates that capability. In the case of SOA, this would be by composing a capability from services. This process will involve defining the individual services needed to fulfil the capability, constructing the capability, maintaining/evolving the capability, and ending the life of



**Fig. 3.** An SOA integration model for realizing NEC.

the capability. This is where new services would be defined, if necessary, or selected for the workflow from existing services.

3. There is also a life-cycle for the individual services themselves. This will go along the line of create, maintain/evolve, end of life, as will happen with the capability itself similarly.

Summarising, the System of Systems to realize NEC will in itself contain a number of asynchronous life-cycles for individual systems. SOA has been proposed as an architectural approach to implementing this System of Systems. The rationale behind this is that the impact of changes to system internals should not affect other systems due their encapsulation through well defined interface specifications.

To recapulate, in this chapter the realization of applying SOA to an NEC environment in light of real-world constraints of military systems is investigated. Whilst SOA based integration is fairly well established in the research literature at a high-level, this chapter aims to add to this knowledge by focussing on how to implement an SOA system of systems in an NEC context. Specifically the challenge investigated is based on a fundamental characteristic of NEC in that its underlying systems will not be built from scratch as SOA-enabled, but rather will evolve from a whole host of existing legacy assets. Examples of concrete assets are a sensor or a database. This investigation is conducted by looking at SOA-based integration from both the capability integrator and service provider's perspectives and is further broken down in terms of the SOA-based service and system platform delivery later on.

### 3 Incremental Service Delivery within the NEC System of Systems

#### 3.1 Introduction to Constraints of developing Service Systems for NEC

The delivery of NEC will not involve a clean sheet implementation, but will come largely from a migration of existing assets. As the UK MOD Defence Equipment and Support (de&s) Handbook on Systems Engineering [19] states, in the context of MOD’s working with disparate commercial approaches to systems engineering:

*“MODs open architecture approach needs to acknowledge existing legacy system boundaries, while over time and where practicable seeking to migrate these systems closer to the target architecture.”*

Incremental delivery is an important design feature when building complex systems. As has been acknowledged for single software systems, in order to deliver capability, full capability can be delivered through a single step, or alternatively through a number of evolutionary steps [9]. Brooks [23] describes this pattern of building a system (as opposed to specifying a satisfactory system in advance) as growth. The UK Defence Industrial Strategy [38] recommends incremental acquisition life-cycles to reduce project management risk and to enable the flexible insertion of new technology to respond to evolving requirements.

Whilst dynamic architectural styles (eg., SOA) have been proposed for the realization on NEC by supporting ongoing development without dependence on previous implementations [43], the ‘*growing*’ nature of military Systems of Systems to realize NEC will involve legacy assets (systems) that will need to be factored in to the development process of services and their integration. The large volume of existing equipment and often the data bound to these provides a major constraint in the migration constraint of military systems to the NEC environment . A legacy system can be described as an old system that still provides essential business services [47]. Most of this equipment was not designed as SOA-enabled; additionally most legacy equipment is no longer actively being further developed. The clean-sheet ideal of developing SOA services that can be integrated and reused is not as simple as presented at the outset when legacy systems are factored in.

#### 3.2 Existing Approaches to Reuse Legacy Systems as (Web) Services

In this section existing literature on recovering and reusing existing legacy systems as services is discussed. These techniques can be broadly categories into ‘*white box*’, ‘*black box*’ and ‘*grey box*’ techniques, generally described as follows:

- **Black box** - Legacy system has no source code available and cannot be decomposed.

- **White box** - Legacy system has source code available. This allows for refactoring and re-architecting of the system.
- **Grey box** - As with black box, but the system can be decomposed, for example the system uses a component architecture.

A black-box approach to reusing legacy software systems across hosted processor execution environments is proposed by Corman [16]. Here the description of a software wrapper given here is of a “*software adapter or shell that isolates a software component from other components and its processing environment (its context)*”. The options he gives for achieving this wrapping include:

- **Re-host** - Translate/recompile source code for new target host processor - this assumes either access to source code or the ability to transcode existing executable code.
- **Hybrid** - Spread system functions between old and new processors. Migrate over time.
- **Emulate** - Emulate the original target processor architecture. This type of approach has been described elsewhere by Booch [10] as putting it ‘*on life support*’.

The use of software emulation of an execution environment allows for a growth path for legacy and useful system executables that previously depended on now obsolete hardware. The approach presented in the paper describes a graphical tool to express data interfaces and constraints of the legacy system to determine whether requirements of the system on a new execution environment can be met. This approach based on the premise that legacy development tools can be used to maintain a computing system executable. The third-party maintainers of software systems, however may be forced (due to contracting) into using prescribed tools for generating artifacts that only useful within a specific suite of development tools, according to Kennedy [30]. These tools will also be subject to a servicing life-cycle and will become legacy systems themselves.

In Corman’s paper the emulated approach was tested on an F-15 tactical fighter to allow the reuse of an Ada 83 Operational Flight Program component with a COTS PowerPC microprocessor. Through a demonstration activity, the emulated component was able to operate within given execution timeliness requirements despite the emulation overhead. Whilst this approach has demonstrated that the legacy system can be reused to extend its service life in a new processing environment, new requirements (functional and non-functional) over time acting on the legacy system as a result of operating an a new context are not yet considered.

The creation of wrappers in a workflow integration context has been used to mediate between legacy C code and Object-Oriented Java code for use within Triana workflows for scientific communities, as documented by Huang [29]. In this situation, tools have been developed to aid in the semi-automatic conversion

between C and Java interfaces and then for the wrapping of Java components within the Triana workflow environment. This technique, however, requires access to the original C code and assumes that the C code will recompile on the target execution environment.

The approach proposed by Zhang [53] can be classified as a grey box approach. Here the assumption is that a particular legacy system can be broken up into smaller components. Following this process these components are wrapped as reusable information services of various granularities based on a domain model analysis to bridge the difference between the domain model and the legacy system component model. A decision tree based on specific criteria is used to determine if a legacy system is suitable for decomposition within this approach. These criteria involve:

- Is reusable and reliable functionality embedded within the system with valuable business logic available?
- Are reuse components from the legacy system reasonably maintainable compared to maintenance effort on the whole legacy system?
- Can the functionality be exposed as independent services and are they useful from the requirements point of view?

If these are satisfied then this approach takes legacy systems with source (or at least individual binary components/classes) available and partitions them up to expose certain parts as services. Here the legacy code components are modelled as UML models to aid in understanding of the legacy system and the linkage between components. Once particular components have been decided as mapping closely to the business defined service components (in this case Web Services) then the glue code is developed - in the case of Zhang's paper, the approach is using Java to mediate between existing C++ code and an Axis SOAP processor.

In summary, whilst this approach attempts to reduce the system complexity compared with wrapping black-box legacy systems, a shortcoming of the approach is that the maintenance associated with the 'glue code' of the mediator components is not given due consideration. Furthermore, this approach has limited applicability to black box '*monolithic*' systems that cannot be decomposed unless they provide available source code. For NEC there are a number of challenges that complicate this situation. Firstly as described, NEC initiative recognises that for service providers and contractors offering functionality, this service functionality can be delivered without ownership of the delivery mechanism. This means that it is not possible to make general assumptions for the state of legacy systems used to provide this service - for instance, whether they are black box monolithic systems or whether they are fully documented systems with source code available in order to be modified and customised by any party within the supply chain.

The approach described by Kotonya [31] aims to modernise legacy systems and provide a means for requirements mapping. To achieve this, domain entities/actors associated with the system are identified and secondly their needs are

mapped to the reusable legacy systems. By performing this the impact of change from the requirements domain can be assessed on the legacy systems. Part of this analysis involves assessing the impact on system life-cycle planning as a result of new enhancements, for example the extensions to legacy systems required to allow operation on new platforms. To achieve this requirement, sources are modelled as viewpoints from various stakeholders within a UML model - this allows further mapping onto modelled legacy systems through a service intermediary.

This approach offers an advantage over Zhang [53] as systems can be modelled in a way that allows for the expression of component cost, certification and dependability requirements. Similarly as with Zhang [53] this paper acknowledges and suffers from the problem that system degradation will occur manifested as patches and *“increasingly complex glue code”*.

### 3.3 Abstract Decision Process Model for Wrapping Legacy Components

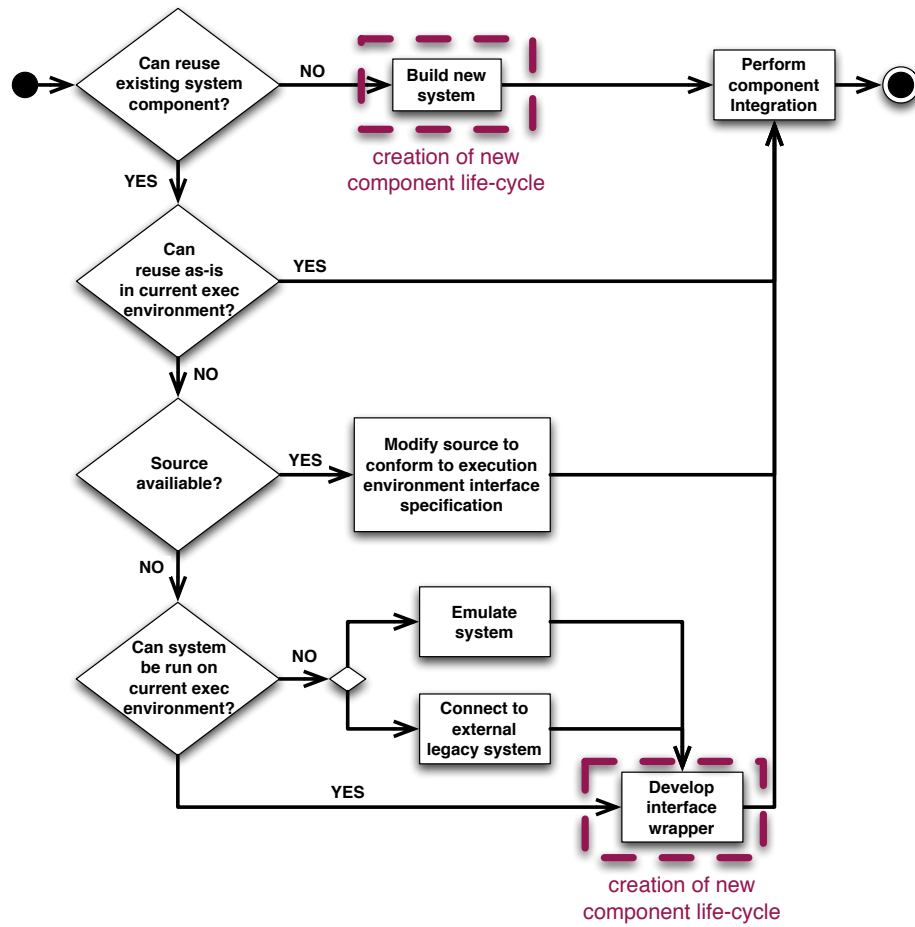
An abstract decision process model for wrapping legacy components is proposed here and illustrated in **Figure 4**. This provides a guide to how existing system components can be selected for integration into the System of Systems that will be used to realize NEC. For instance, this model is to be used to assist in the integration process of system components when migrating to or between execution architectures or in new operating contexts. Wrappers are a standard technique for reusing a legacy software component in a new originally unintended context and can be used to reuse legacy systems in new contexts. The concept of wrappers will be explored further in Section 5.1.

The decisions that the model will guide involves whether to wrap existing components based on the characteristics of modifiability for these components. Such guidance will inform the developer/integrator when new system life-cycles are created as a side-effect of implementing a wrapper; these are shown in **Figure 4**.

As discussed later in Section 5.1 the characteristics of the legacy system that will guide this process model’s paths involve whether existing system components are modifiable. In the case of software components this can be through the inability to gain access to source code or that the component may not be modifiable (or at least difficult or costly) due to its phase in the servicing life-cycle. In the case of a wrapper being used, there may be no need to modify the underlying component.

The main decision points of the process shown in **Figure 4** can be broken down as follows:

- The ideal case is where a system component does not need to be wrapped and can be integrated as is. In the scope of this chapter we do not focus on the evaluation and certification process of integrating systems in NEC for space, but do point the reader to our previous work conducted within the NEC context [52].



**Fig. 4.** UML activity diagram for wrapping options when integrating a component into a system or workflow.

- If the source code or development tools for the system component are available (and ideally documented) then the option is available to modify the component to operate with the interface standard required for integration in the System of Systems. This maintenance may be costly due to a lack of developer expertise or documentation within the organization to perform this modification of legacy code.
- In the case where the system component is non-modifiable, then the decision to make is whether the (software) system can be executed on the target execution environment or not.
  - Where the component can be executed on the target execution environment, but does not conform to the correct interface, then an interface-mediating wrapper can be developed.
  - Where the component cannot be executed on the target execution environment - for example due to relying upon a different microprocessor architecture for execution or incompatible service container - then there are two possible options:
    - 1) emulate the microprocessor architecture;
    - 2) reuse the legacy execution environment and provide an interface (possibly physically) between the two execution environments.

An interface-mediating wrapper can then be developed to mediate between the differing system interfaces.

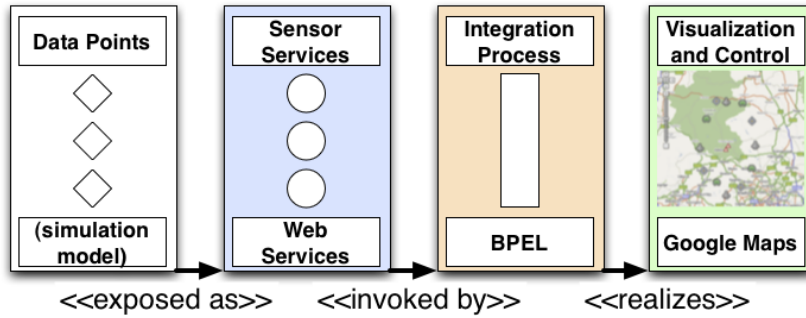
A key hypothesis to state here is that the creation of wrapping interfaces will increase the overall complexity of the wrapped systems as a whole and, therefore, the System of Systems. The wrapped interfaces will need to evolve in accordance to the interface requirements of integrating with the System of Systems environment over time. The process model, therefore, provides decision support to trade-off reimplementation cost against complexity and maintenance responsibility.

## 4 NECTISE SOA Demonstration and Critical Evaluation

In this section we discuss our experience with the NEC Software Demonstrator and discuss the legacy problems it can help to expose both above and below the Service Layer. The NEC Software Demonstrator was used to illustrate aspects of the research into systems architecture and through-life systems management (TLSM) within the NECTISE project and was implemented using Web based tools and technologies; for instance WSDL, SOAP, BPEL, XML Schema, HTML+AJAX.

The scenario aim of the Software Demonstrator was to model a Region Surveillance capability using dynamic service integration of sensor networks in the NEC battlefield - this allows a comprehensive 'picture' to be formed of the geographical region based on data communicated to a controller from deployed mobile sensors. Due to the high cost and confidential nature of testing

our research work on real military systems a small but informed simulation environment has been developed for use within this research work. The surveillance capability was based on Points of Interest (PoIs), physical and military features within a geographical area that are detected by a group of simulated sensors. The integration of sensors was achieved by using a workflow to contact a sensor service registry and to dynamically discover sensors for a given region. The workflow uses simulated sensors that access a simulation of PoIs and then exposes this data as services. The sensor data is then processed to eliminate duplicates and points outside the region of interest and the detected feature positions are displayed on a map. The workflow can be illustrated at a high level in **Figure 5**. This diagram can be directly mapped onto the SOA integration model shown in **Figure 3** and illustrates the implementation technology used in the lower boxes. The workflow integrates Web Services which were the chosen demonstration implementation technology for SOA. The implementation of region surveillance used Google Maps to display the results from the feature detection workflow. A screenshot from this interface can be shown in **Figure 6**. This screenshot shows the output of the sensor integration workflow plotted onto a Google Maps interface within a web browser. The blue rectangle is the region of interest. The lists on the right of the image show the detected features and the sensors that have been accessed in the workflow.



**Fig. 5.** A high-level overview of the sensor integration workflow.

The Software Demonstrator was developed in NetBeans 6.1 IDE on a GlassFish application server, which enabled our research group members to write, deploy, test and debug SOA applications using the Extensible Markup Language (XML), Business Process Execution Language (BPEL) and Java Web Services.

#### 4.1 Makeup of the Demonstrator Implementation

The Software Demonstrator consists of three main modules:

1. Web Services to simulate capabilities of different systems;

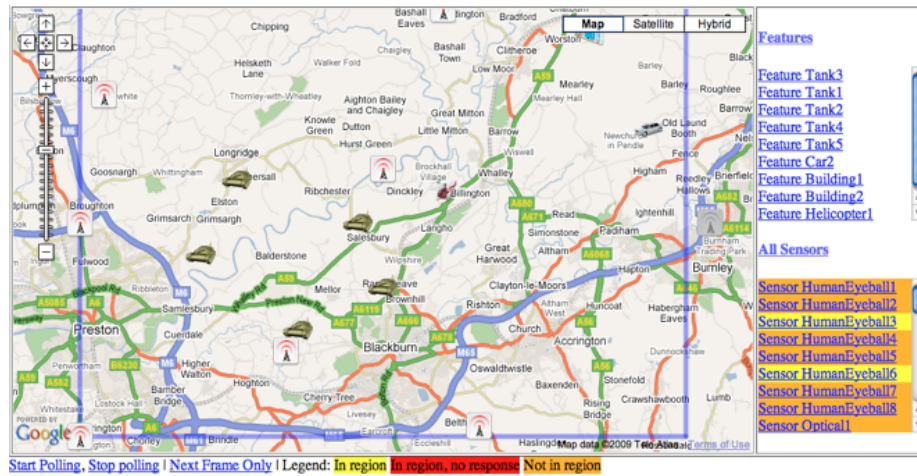


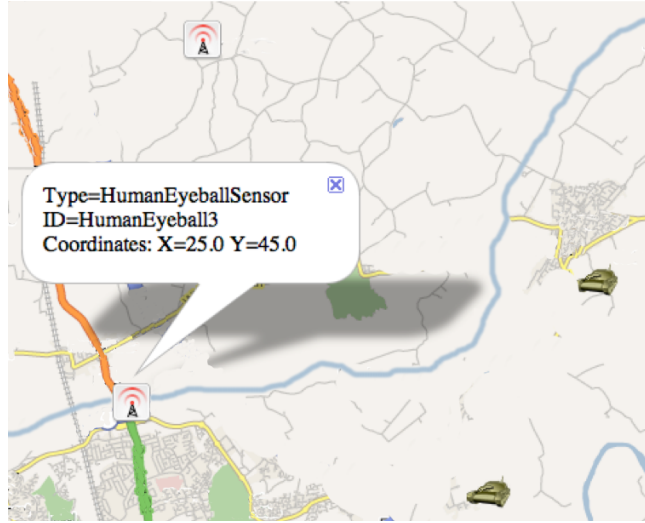
Fig. 6. Sensor surveillance Software Demonstrator screenshot.

2. A workflow module for dynamic web services selection and composition;
3. A client interface with sensor information display and user input.

The workflow dynamically discovers and integrates data from the sensor services and is implemented in a Java written Web Service itself. An Apache Derby SQL database server contains a model of the sensor and feature positions and their other attributes. There are five types of sensors maintained in the database including human observer, optical, infrared, and long-range and short-range radar. The detectable features include bridges, buildings, vehicles, helicopters, and humans. Each feature has a coordinate position  $x_n$ ,  $y_n$  and movement is simulated through a path of coordinates. One or more of the sensors can detect each type of feature. Sensors are assumed to be static. **Figure 7** illustrates a user querying information about a sensor in the field.

Three types of web services have been created: Sensor Registry, Sensor, and Data Filter. The Sensor Registry was implemented as an Enterprise Java Bean (EJB). The registry's function is to return a list of sensors that can 'see' the region of interest. This square region is defined by the request with two pairs of coordinates  $x_1$ ,  $y_1$  and  $x_2$ ,  $y_2$  and Sensor services are selected from their attributes about position and range. A Sensor service is called for each entry in the list returned by the registry. The Sensor returns a list of all features that sensor can detect. The Data Filter service reduces the detected features from all the sensors by eliminating duplicate feature points and those outside the region of interest.

The Map Client allows the user to define the coordinate pairs for the ROI along with required response time. The user interface, shown in **Figure 6**, displays a map of the ROI utilizing Google Maps and POI overlay. In order to integrate the Map Client application with the rest of the components of the



**Fig. 7.** Example of a user querying a sensor node within the Map Client.

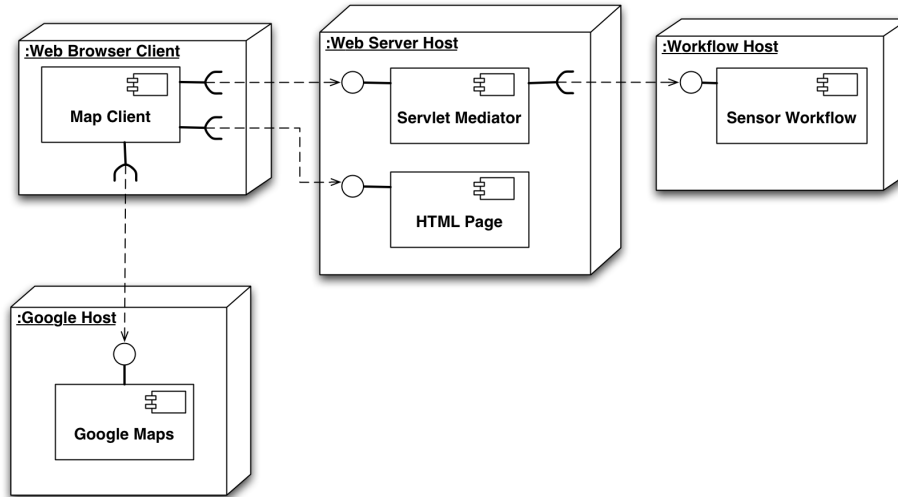
demonstrator system as a whole, we needed the application to act as a client component of the Sensor Workflow component.

Due to the ‘Same Origin Policy’ [42] built into most web browsers, this technical limitation required a mediating component to connect the web browser client-side application with the Sensor Workflow. This policy prevents the properties of a document loaded from one origin being modified from another origin. Whilst we could have implemented a Web Services client within the web browser application, the allocation of this responsibility to the Servlet meant that the implementation architecture was simplified and allowed us to build upon existing and well-proven technologies from the Java Framework for Web Services.

In order to achieve this connectivity, an intermediate component was created to mediate requests from the user interface to the Sensor Workflow. To implement this mediator, a Java Servlet was created to mediate requests from the HTML+JavaScript part of the Map Client to the Web Service interface of the Sensor Workflow. Whilst we could have implemented a web services client within the JavaScript part of the user interface, the allocation of this responsibility to the Servlet meant that the implementation architecture was simplified and allowed us to build upon existing and well proven technologies from the Java Framework for Web Services in addition to easing the debugging process.

The implementation of the user interface of the Map Client consisted of an HTML+Javascript application that makes asynchronous requests to the mediator servlet commonly known as the AJAX technique (Asynchronous JavaScript And XML). In order to connect the client application to the mediator Servlet a simple XML schema was created to pass region and QoS requests to the Sensor Workflow through the mediator and consequently, return sensor and feature

information from the Sensor Workflow. The features were then overlaid into a Google Maps display of the ROI. A UML deployment diagram is illustrated in **Figure 8**.



**Fig. 8.** UML Deployment Diagram for Map Client.

Whilst the primary aim of the Software Demonstrator was to demonstrate architectural and dependability aspects of NEC (based on our previous experience [34]), the life-cycle aspects are focussed on in this chapter and demonstrate the challenges of migrating legacy assets to an SOA NEC environment as introduced previously in the chapter.

## 4.2 Life-cycle Aspects of the Demonstrator

The following life-cycle related use-cases were illustrated through the Software Demonstrator:

1. Show Platform Upgrade
2. Show Service Life-cycles (Evolution)

Each of these use-cases is described in the context of the life-cycle work and illustrate the SOA implementation challenges when reusing legacy systems in SOA. These use-cases help expose evolution challenges above and below the Service Layer.

**Show Platform Upgrade** This use-case illustrates a service upgrade. The upgrade to a service may be by changing the underlying system implementation

without changing the service definition. The upgrade may improve quality of service attribute values, or it may improve efficiency of service delivery, thereby providing an internal benefit without affecting service delivery. This is found at the Platform Layer in **Figure 3**. The process can be described as having foundation in the concept of Refactoring [1].

**Show Service Life-cycles (Evolution)** This use-case illustrates the life-cycle of a service and its underlying system(s). The service life-cycle would include:

- Conceptual definition using service discovery to identify capability gaps;
- Development using evaluation to identify and trade-off possible solutions;
- In-service, by adding and removing the service from deployment;
- Disposal, where a service implementation is retired, causing:
  1. A change in service execution from one system to another by being replaced by another system;
  2. A change to the integration due to upgrading to new version, requiring all users of the existing service to migrate to the new one;
  3. End of life for service, requiring users of existing service to find alternative (newer) service or cease use of that function entirely.

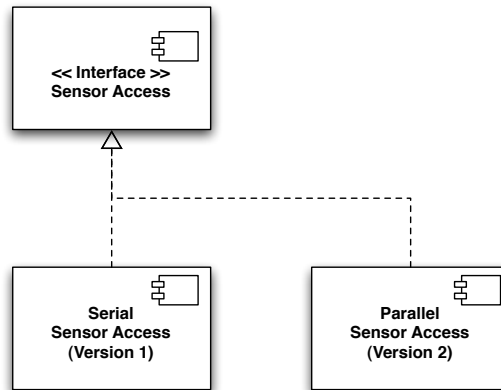
The lesson learned here was that whilst it is possible to view the wrapping of a legacy system to provide an SOA interface as a solution for controlling the description of and access to its functionality, there are problems regarding the evolution and versioning of this interface description. Here is a research challenge whereby there is a dependancy between the service clients and service providers based on the agreed functionality of a particular version of a service. For example, if a service provider updates the interface of a service in a non backwards compatible manner and does not maintain the old version of the service interface, then the old client will not be able to access the service without reengineering. This shows that the problem of ‘legacy’ can occur at the Integration Layer.

By supporting the existing interface, backwards compatibility can be ensured. An example of this is to add an additional versioned interface to a service or by extending the existing interface with new parameters or functions in a backwards-compatible manner. A second option is for the service provider to provide multiple interfaces [4, 13]. Whilst it is possible to still maintain interfaces for older versions of clients, this creates a maintenance responsibility.

As stated, in the case where backwards compatibility cannot be maintained and a new interface is defined, then interface mismatch will occur and will require upgrading of clients. In this case, older clients need to be migrated to operate with the new service interface. Here a wrapping solution may resolve the interface mismatch between the old client and new service, with the associated issues of increasing the complexity of the System of Systems as a whole and the creation of maintenance responsibility for the wrapping solution at the SOA Interface Layer level. There are many situations in the real-world where a client not being able to upgrade to a new service interface, for instance:

1. client systems no longer being maintained (legacy systems);
2. timeliness of upgrade cycles.

Within the development of the Software Demonstrator the concept of service evolution was demonstrated during its development. Two versions of a sensor access component were developed with both versions corresponding to the same client interface. This meant that the client was able to access the second version of the component with the same interface despite it being evolved to provide a different underlying behavior. To realize this, two versions of sensor workflow were implemented to represent serial and parallel processing of sensor aggregation as shown in **Figure 9**.



**Fig. 9.** UML diagram of serial and parallel sensor access components.

**Version 1 - Serial Sensor Access** - Version 1 of the Sensor Access component implements a serial process of sensor access, thus the sensor accessing is sequentially processed. In implementation, there is only one thread in the program. Each sensor name returned from the aggregated sensors list is handled sequentially.

**Version 2 - Parallel Sensor Access** - Version 2 of the Sensor Access component used parallel sensor access. For each sensor name returned from Sensor List, a thread was launched to access the sensor Web Service. This provided the following advantages:

- Improved workflow performance by launching network requests in parallel, rather than waiting for one to finish before launching the next. In related

research work in the GRID context, Quan [17] successfully demonstrated that parallel matching of GRID processing resources and their subsequent configuration allowed that approach to meet SLA deadlines and achieve cost optimization.

- Improved resilience to failure of any Web Service requests. Web Service technology usually has a timeout for lost responses, typically 30 seconds. By issuing requests in parallel, the overall timeout would be a maximum of one failed response.
- The additional enhancement was to model variable availability in sensor accesses. Each sensor has a random ability to respond to a request. This is a basic model of two aspects:
  - network traffic delays in either request or response
  - request or response message loss - in particular using Web Services there is no guaranteed messaging, unless WS-ReliableMessaging is used.

The implementation of both the Version 1 (serial) and Version 2 (parallel) versions of the sensor access services that correspond to the same interface demonstrated the following advantages:

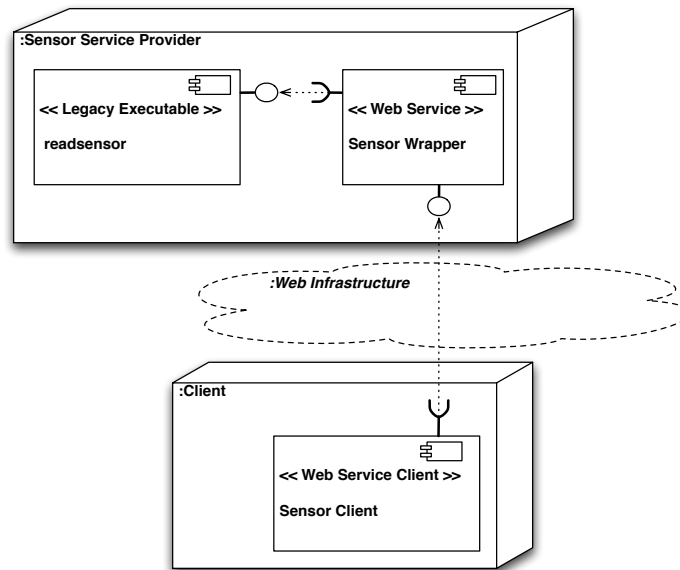
1. The two versions of the software services could be developed independently and orthogonally.
2. The versions of the software services could be swapped at run-time, meaning that they could be used for demonstration purposes to demonstrate the effects of serial and parallel access to sensors from a workflow. However, this meant that the interface could not be changed, leaving the only option of extending the interface and require clients to possess knowledge to utilize this (advanced) interface.

### 4.3 Exposing a Legacy Sensor Application to an SOA Network

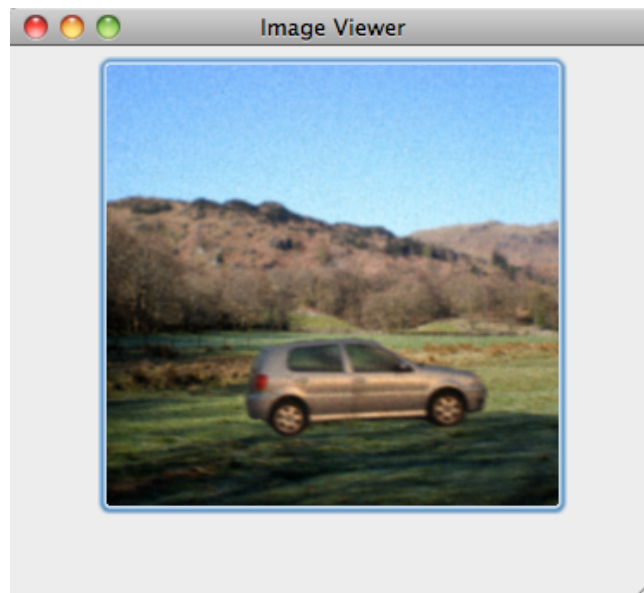
The example presented here demonstrates wrapping legacy systems to operate within an SOA environment. In this example an unmodifiable legacy sensor application is exposed as a Web Service in order to allow it be integrated into an SOA enabled business process. This system model can be summarized with UML as in **Figure 10**.

In this model a Service provider (**‘Sensor Service Provider’**) and client **‘Sensor Client’** are defined. The Web Service (**‘Sensor Wrapper’**) server is hosted within the **‘Sensor Service Provider’** along with the legacy application (**‘readsensor’**) that communicates with the physical sensor hardware. **‘Sensor Wrapper’** acts as both an external facing Web Service and a wrapper to the **‘readsensor’** application.

In this example **‘readsensor’** is a legacy executable that reads in an image from an optical sensor, performs some image processing and then directs a textual encoding of the image in base64 to the standard output stream. We acknowledge that this is a contrived example as the Demonstrator was simulation based and did not use any physical equipment; however, it demonstrates our argument. A mockup screenshot of an example client of the service can be seen in **Figure 11**.



**Fig. 10.** UML diagram of wrapped sensor System of Systems.



**Fig. 11.** Mockup screenshot showing a typical client decoded optical sensor image of the tracked target.

The wrapping of the ‘**readsensor**’ application to a GlassFish hosted Web Service was a relatively simple task to implement and test. The reader is directed towards the code listing in **Program 1** to see the implementation of this wrapper. However, this example exposes the fragile nature of this kind of wrapping solution as there are limitations on the value the wrapping component can add to the legacy application. To give an example; the business process utilizing this exposed sensor application has a requirement to not only retrieve the sensor image data, but also to provide a timestamp and geo-location coordinates for the sensor. Given that the original ‘**readsensor**’ application cannot be modified; this may be achieved by recording the time that the sensor was accessed and attaching a GPS to the service provider with the assumption that the sensor reading is taken immediately following the call to the ‘**readsensor**’ application and the server is located at the same location of the sensor. This implementation can be summarized as the ‘*decorator design pattern*’ [24].

Whilst the above example demonstrates that it is possible to bridge the mismatch between capability requirements on an unmodifiable legacy system and the functionality that the system exposes, there will be a limitation to how far this approach can be taken. To give an example, if a business requirement is that the sensor imagery needs to be exposed along with directional data (for example, as a compass would provide), then this functionality cannot be provided from ‘**readsensor**’ as-is. Likewise, if the sensor is a remote sensor and not physically attached to the Web Service mediator, then it may not be possible to determine the geo-location of the sensor imagery.

---

**Program 1** Simple Sensor Wrapper program in Java.

---

```
/**
 * Simple Sensor Wrapper
 */
@WebMethod(operationName = "getSensorImage")
public String getSensorImage()
{
    /**
     * Launch 'readsensor' program and read the output
     * represented by plain text with base64 encoding.
     */
    Runtime runtime = Runtime.getRuntime();
    Process proc;
    StringBuffer programOutput
        = new StringBuffer();
    try
    {
        proc = runtime.exec("readsensor");

        InputStream inputStream = proc.getInputStream();
        InputStreamReader inputstreamreader
            = new InputStreamReader(inputStream);
        BufferedReader bufferedreader
            = new BufferedReader(inputstreamreader);

        // read the program output
        String programOutputLine;
        while (
            (programOutputLine = bufferedreader.readLine())
            != null )
        {
            programOutput.append(programOutputLine);
        }

    } catch (Exception ex) { return null; }

    /**
     * Return the wrapped program output to the
     * Web Service client.
     */
    return programOutput.toString();
}
```

---

## 5 Conclusions of Legacy System Migration Towards SOA

Wrappers are a standard technique for reusing a legacy software component in a new originally unintended context and have been discussed in this chapter as a solution for this purpose. Whilst in this chapter software wrappers have been discussed as the primary technique for reusing legacy components there are, however, other techniques that are covered here. Aside from reimplementing the component from scratch, an alternative technique is to ‘recover’ legacy software through reverse engineering of its business logic [53]; this also helps to document its organisation and functionality [47]. The drawback of this approach is that it is invasive to the legacy system and that the accurate recovery of the business logic is difficult. Furthermore, reverse engineering of software systems may be impractical if source code is not available [26].

We identified and discussed the NEC need to investigate the wrapping of legacy equipment into an SOA as a response to the requirements for NEC discussed in **Section 2.3**. In this chapter, the realization of this process has been explored in more detail to expose hidden factors (for instance, System of Systems complexity, coupling, and maintenance costs) when migrating to an SOA enabled NEC environment from existing legacy assets.

The point to focus on here is that suppliers and system providers transferring to SOA based delivery of functionality will undertake the process of wrapping existing legacy assets into an SOA in order to conform to an SOA implementation’s communication infrastructure, for instance a Web Services middleware. We can characterize this wrapping of legacy systems to an SOA middleware interface as the Adapter design pattern [53, 25], which is sometimes referred to as a software wrapper. The software wrapper is a form of encapsulation whereby a software component is exposed with an alternative abstraction manifested as a software interface [6]. The purpose of the wrapper, therefore, is to allow system components that would be otherwise incompatible to be able to communicate with each other. Whilst the wrapper can be a complex piece of software, the cost of wrapping is often less than reimplementing the legacy system/component from scratch [47].

### 5.1 Challenges in Legacy System Migration Towards SOA for NEC

The frequent changes made in the short life-cycles for new requirements on services to realize ongoing capability requirements for NEC could cause significant modifications of the interfaces of these military services, which lead to serious versioning and compatibility problems. Services need to be frequently re-integrated and re-configured to provide a reliable and sustainable capability. When legacy systems are factored into the System of Systems the problems that occur here, therefore, are twofold:

- 1) When new requirements are required of services, the systems that provide them need to be re-engineered for them to evolve to meet their new requirements. If the underlying systems are ‘*healthy*’ systems that are currently in

a evolutionary or servicing phase of their life-cycle (as can be illustrated in a Staged Life-cycle [40]) then this is just a reengineering process, where current approaches to agile development can be applied where appropriate. If there is a legacy system that is being wrapped to the service interface, then a danger with the practice of wrapping legacy assets is that the software system underneath the SOA interface will be difficult or costly to modify once in the late servicing and phase-out phases of a staged life-cycle model. As stated one attribute of this problem is the *'health'* of the development of that underlying system. For example if the system is in a phaseout phase of its life-cycle then it will be costly to modify, particularly if it relies upon diminished manufacturing sources (DMS) [32] both for hardware and software development.

- 2) A wrapping process could be used to satisfy an initial requirement to use a particular legacy system through an SOA integration middleware ‘as-is’, for example taking a black box approach. However, there will be challenges in evolving that system in order to adapt to ongoing capability requirements as a result of exposing the asset in a new environment through the service infrastructure. Examples of such challenges are having to deal with the increased information processing throughput over time; and using a system (or system component) in a domain or context that it was not intended or designed for.

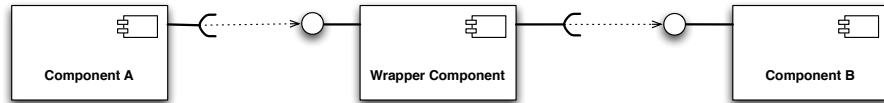
Reflecting on the discussion of existing approaches to wrap legacy systems to service interfaces in **Section 3.2** - (with the exception of legacy systems which are either highly composable or have full source code available) - these systems suffer from the specific problems that they involve glue code to bridge the gap between the service requirements and the functionality and throughput ability of the legacy system. Challenges that still remain from existing work is the management of complexity over time with new requirements demanded from services - this has the potential to become unmanageable in a long running System of Systems over time. In the NEC context, the major problems here are the large number of stakeholders with differing capability requirements that will demand a high tempo for agile changes to be made services.

Using agile development methods for NEC, it is very important to check whether the current change will affect the interface of the service or QoS defined in the Service Layer Agreement. Good advice is that a comprehensive analysis and evaluation of the impact on dependencies must be done before any change is put into effect. However, where there are demands from multiple stakeholders in NEC that demand rapid response to ongoing capability requirements, this is not an easy process to go through when factoring in the ‘legacy’ challenge.

## 5.2 Maintenance Life-cycle of Wrappers

Just as software components/systems need to be maintained, wrappers will possess their own servicing life-cycle since they will wrap between one or more

evolving interfaces. Wrappers will need to maintain interoperability despite ongoing interface evolution. **Figure 12** provides a simple model to illustrate the relationship of the wrapper component against the two component interfaces that it is mediating between.



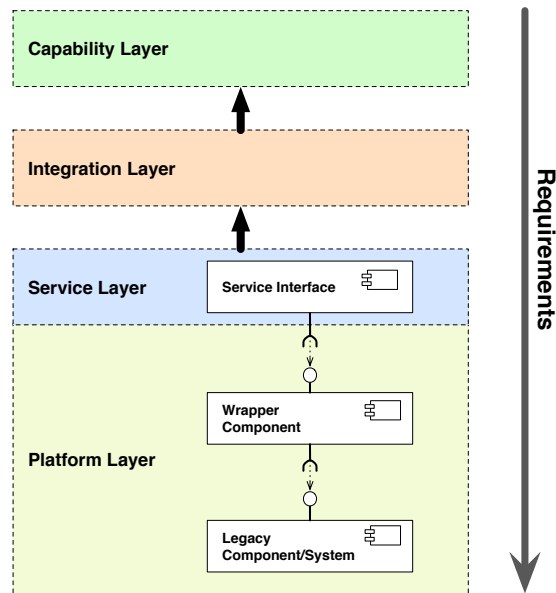
**Fig. 12.** UML component diagram representing an abstract model of component interface wrapping.

In the UNIX programming domain, programmers are advised to avoid using wrapping ‘*glue*’ layers to hide cracks and unevenness in software layers [41]. The problem here is that this process is responsible creating an additional layer of glue with the caveat of too many layers being the loss of transparency within a system. Whilst the technology in the SOA domain is different, the concept still applies, as the network integration of services will include wrapping components that will have to be managed and maintained. Therefore, the life-cycle complexity of the System of Systems is increased through the use of wrappers. We can logically conject that since a wrapper of a system mediates between two moving standards (which may be proprietary) the wrapper will also need to be maintained and evolved in accordance with its wrapped interfaces.

**Figure 13** demonstrates the placement of the legacy system wrapper in the context of NEC service system integration. As with the capability life-cycle description in Section 2.5, requirements can be viewed as affecting the underlying system through the interface. These changes, as discussed earlier, can be functionality changes or performance requirement changes for instance an increase in processing throughput.

## 6 Conclusion and Opportunities for Future Work

This chapter reports our understanding of how to implement and realize NEC Systems of Systems when developing from existing assets rather than a purely clean-sheet position. The NECTISE program included the investigation of sustainable system maintainability with respects to the NEC problem and this work has explored a number of challenges to be faced in realizing NEC when migrating legacy assets to an SOA based System of Systems over time.



**Fig. 13.** Wrapping a legacy system in the context of capability integration.

The challenges and constraints that have guided this research in the NEC domain include:

- System of Systems will be grown over time and not comprehensively designed upfront.
- Factors such as cost will drive the migration of existing/legacy assets to an SOA infrastructure rather than developing solutions from scratch.
- Systems that realize NEC possess varying timescales for their life-cycles and need to provide integrated operation at their current stage. In military systems, long life systems are common; for example a ship may have a service life of forty years. During this time the operational environment and operational concepts will both evolve.

This work shows that whilst it is possible to produce an abstract system model for SOA integration by abstracting the functionality of platform hosted systems into services which can be recombined (as in **Figure 3**), there are still *‘devils in the detail’*. Application of SOA to an NEC environment has been investigated in light of real-world constraints of military systems. Specifically the reuse of legacy systems in an SOA-enable NEC environment has been considered. SOA-based integration has been viewed from both the capability integrator and service provider’s perspectives and is further broken down in terms of the SOA-based service and system platform. We have explored problems from both legacy systems (changing underneath and extra requirements due to a context change)

and evolution and mismatch at the Interface Layer. Problems of ‘legacy’ can occur at both the Platform Layer and at the SOA Integration Layer. These will occur, not just at the stage where non-SOA legacy assets are integrated, but when SOA-enabled services are evolved over time.

A summary of the understanding for the use of legacy systems is as follows:

1. The wrapping of legacy systems has been illustrated through the diagram shown in **Figure 13** and the worked sensor example in Section 4.3. Ongoing capability requirements affect wrapped legacy systems through the Service Interface layer. Limitations to how far this approach can be taken have been discussed, in particular for performance limitations and the inability to modify the system. The mediator/wrapper will need to bridge the gap of requirements put upon that system in its new context and the functionality implemented by the system in its original context. As discussed in Section 5.2 a hypothesis is presented that maintenance life-cycles need to be associated with wrappers created to overcoming interface mismatch. A decision support model has been proposed (**Figure 4**) to inform the developer of the creation of these life-cycles and the maintenance responsibility. A more structured documentation and analysis process is planned for future work. Part of this documentation includes the tracking of the creation of and life-cycles associated with wrapping components.
2. Whilst it is possible to view the wrapping of a legacy system to an SOA interface as a solution for controlling the description of and access to its functionality, there are still problems regarding the evolution and versioning of this interface description. Whilst it will be possible for service providers to support older versions or at least provide backwards-compatible versions of interfaces, semantic changes or the maintenance of a single interface version will prompt a reengineering task for existing clients. The evolution of the dependancy between service clients and service providers based on the agreed functionality of a particular service interface version is a challenge for future research.
3. Use of wrapper/adaptor technologies is a potential solution to the problem of architectural and interface mismatch. This occurs below the Interface Layer for replacing replacing traditional systems whilst still maintaining the SOA interface definition and also above the Service Layer when providing functionally equivalent services from different service providers. The Software Demonstrator has illustrated that changing a serial sensor aggregation component with a parallel sensor aggregation component to improve performance is possible whilst encapsulating this change behind the same service interface.

Future work has been considered which would include formally analyzing the complexity, maintainability and interoperability for wrapping legacy systems into SOA infrastructures used in a System of Systems to realize NEC. Modeling of this work would be investigated in the context of documenting and reverse

engineering the wrapping and legacy architectures through the use of MODAF (MOD Architecture Framework).

A structured approach to evaluating the replacement of components is required for future development of the Software Demonstrator. An approach for the design of a component behaviour test suite has been proposed by Flores [22] for equivalent components. This approach can be used as a starting point and will be key to evaluating the through-life upgrading, refactoring and replacement of components whilst still maintaining a System of Systems to realize capability.

Finally, whilst only wrappers have been suggested here, there is scope for other techniques for mitigating architectural mismatch to be investigated and evaluated.

## 7 Acknowledgements

The work reported in this paper has been supported by the NECTISE programme, jointly funded by BAE Systems and the UK Engineering and Physical Sciences Research Council Grant EP/D505461/1. The authors would like to thank John Davies from BAE Systems (INSYTE) for his input into this research work.

## References

1. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
2. Paul Allen. *Service Orientation: Winning Strategies and Best Practices*. Cambridge University Press, May 2006.
3. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.
4. Vasilios Andrikopoulos, Salima Benbernou, and Mike P. Papazoglou. Evolving services from a contractual perspective. In *CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, pages 290–304, Berlin, Heidelberg, 2009. Springer-Verlag.
5. Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.
6. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2 edition, April 2003.
7. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.
8. Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.
9. Barry Boehm and Wilfred Hansen. The spiral model as a tool for evolutionary acquisition. In Pam Bowers, editor, *Journal of Defense Software Engineering*, volume 14.5, pages 4–11. Crosstalk, May 2001.
10. Grady Booch. Nine things you can do with old software. *IEEE Software*, 25(5):93–94, 2008.
11. David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. W3C Note NOTE-ws-arch-20040211, World Wide Web Consortium, February 2004.

12. Fred P. Brooks, Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.
13. Cisco Systems. Service virtualization: Managing change in a service-oriented architecture. Technical report, Online: <http://www.cisco.com/en/US/prod/collateral/contnetw/ps5719/ps7314/prod%5fwhite%5fpaper0900aecd806693c2.html> - Last accessed 13th October 2009, 2007.
14. David Cohen, Gary Larson, Doug McDougal, and Bill Ware. Extending life cycle of legacy systems. *Computer Networks and Mobile Computing, International Conference on*, 0:291, 2003.
15. Santiago Comella-Dorda, Kurt C. Wallnau, Robert C. Seacord, and John E. Robert. A survey of legacy system modernization approaches. Technical report, Report CMU/SEI-2000-TN-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 2000.
16. Dave Corman. The iuls approach to software wrapper technology for upgrading legacy systems. In Pam Bowers, editor, *Journal of Defense Software Engineering*, volume 14.12, pages 9–13. Crosstalk, December 2001.
17. Jorn Altmann Dang Ming Quan and Laurence TYang. Improving the capability of the sla workflow broker with parallel processing technology. In *International Journal of Computer Systems Science and Engineering*, volume 24, September 2009.
18. de&s. The systems engineering handbook: Principles, practices and techniques. In *Draft D*. UK Ministry of Defence, 2007.
19. Director General Safety & Engineering. Implementing systems engineering in defence. Technical report, UK Ministry of Defence. de&s, 2008.
20. Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.
21. Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, L. Umit Yalcinalp, Kevin Liu, David Orchard, Andre Tost, and James Pasley. *Web Service Contract Design and Versioning for SOA*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.
22. Andres Flores and Macario Polo Usaola. Testing-based assessment process for upgrading component systems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 327–336, 28 2008-Oct. 4 2008.
23. Jr. Brooks F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.
25. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
26. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12:17–26, 1995.
27. Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.
28. D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, January 1995.

29. Yan Huang, Ian Taylor, David W. Walker, and Robert Davies. Wrapping legacy codes for grid-based applications. *Parallel and Distributed Processing Symposium, International*, 0:139b, 2003.
30. Damian Kennedy and Sergey Nesterov. Issues with third party maintenance of software intensive legacy systems: A case study – avionic mission systems. In *18th Annual International Symposium of INCOSE Utrecht, The Netherlands*, 2008.
31. Gerald Kotonya and John Hutchinson. A component-based process for modelling and evolving legacy systems. *Softw. Process*, 13(2):113–125, 2008.
32. Kenneth Littlejohn, Michael V. DelPrincipe, Jonathan D. Preston, and Dr. Ben A. Calloni. Reengineering: An affordable approach for embedded software upgrade. In Pam Bowers, editor, *Journal of Defense Software Engineering*, volume 14.12, pages 4–8. Crosstalk, December 2001.
33. Liu Liu, Duncan Russell, Nik Looker, David Webster, Jie Xu, John Davies, and Ken Irvin. Evolutionary service-oriented architecture for network enabled capability. In *International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS), Leeds, UK*. Published by the eWiC series of the British Computer Society (BCS), 2008.
34. Lu Liu, Duncan Russell, David Webster, Zongyang Luo, Colin Venters, Jie Xu, and John K. Davies. Delivering sustainable capability on evolutionary service-oriented architecture. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:12–19, 2009.
35. Lu Liu, Duncan Russell, Jie Xu, John Davies, and Ken Irvin. Agile properties of service oriented architectures for network enabled capability. In *Realising Network Enabled Capability (RNEC 08), Leeds, UK*, 2008.
36. Microsoft. Soa in the real world. Online Book. Retrieved September 21, 2009, from <http://www.microsoft.com/DOWNLOADS/details.aspx?FamilyID=cb2a8e49-bb3b-49b6-b296-a2dfbbe042d8&displaylang=en>, August 2007.
37. NECTISE. Network enabled capability through innovative systems engineering (nectise). Online: <http://nectise.com/>, 2005.
38. UK Ministry of Defence. Defence industrial strategy: Defence white paper (cm6697). Technical report, UK Ministry of Defence, 2005.
39. UK Ministry of Defence. *Joint Services Publication 777*. [http://www.mod.uk/NR/rdonlyres/E1403E7F-96FA-4550-AE14-4C7FF610FE3E/0/nec\\_jsp777.pdf](http://www.mod.uk/NR/rdonlyres/E1403E7F-96FA-4550-AE14-4C7FF610FE3E/0/nec_jsp777.pdf), first edition, 2005.
40. Václav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.
41. Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.
42. J. Ruderman. Same origin policy for javascript. Technical report, Online: [https://developer.mozilla.org/En/Same\\_origin\\_policy\\_for\\_JavaScript](https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript), 2009.
43. Duncan Russell, Nik Looker, Lu Liu, and Jie Xu. Service-oriented integration of systems for military capability. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:33–41, 2008.
44. Duncan Russell and Jie Xu. Service oriented architectures in the delivery of capability. In *Proc. of Systems Engineering for Future Capability*, 2007.
45. Duncan Russell and Jie Xu. Service oriented architectures in the provision of military capability. In *UK e-Science All Hands Meeting 2007, Nottingham, UK*, 2007.
46. Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
47. Ian Sommerville. *Software Engineering: (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

48. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
49. uddi.org. UDDI Executive White Paper, 2001.
50. W3C. Soap version 1.2 part 0: Primer (second edition). online, April 2007. W3C Recommendation.
51. Web Services Description Working Group. Web services description language (wsdl) version 2.0 part 1: Core language. Technical report, W3C, 2007.
52. David Webster, Nik Looker, Duncan Russell, Lu Liu, and Jie Xu. An ontology for evaluation of network enabled capability architectures. In *Realising Network Enabled Capability (RNEC 08)*, Leeds, UK, 2008.
53. Zhuopeng Zhang and Hongji Yang. Incubating services in legacy systems for architectural migration. *Asia-Pacific Software Engineering Conference*, 2004.