Evolving Existing Systems to Service-Oriented Architectures: Perspectives and Challenges

John Hutchinson, Gerald Kotonya, James Walkerdine, Peter Sawyer, Glen Dobson, Victor Onditi
Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{hutchinj, gerald, walkerdi, sawyer, dobsong, onditi}@comp.lancs.ac.uk

Abstract

The advent of and growing interest in Service-Oriented Architectures (SOA) present business leaders with a number of problems. They promise to deliver hitherto unseen business process agility, but at the risk of making investment in existing systems obsolete. The established orthodoxy is that the maintenance problem presented by installed systems is about finding an acceptable balance between risk involved in evolving the system and benefits offered by the update. SOAs represent a "paradigm-shift" and, as such, present a more complicated problem: how to minimise the risk to their investment (existing software systems) and exploit the benefits of migrating to SOA. We provide a review of a number of approaches that may contribute to a pragmatic strategy for addressing the problem and outline the significant challenges that remain.

1. Introduction

Businesses use software systems in every corner of their operations. This suggests a strong relationship between a business' success and the "fitness for purpose" of the software systems that it relies on. As businesses grow and adapt to the prevailing market conditions, so too must their software systems and there is a well-established understanding that the usefulness of a given software system is dependent on its continued maintenance and evolution to reflect the needs of its changing environment [1]. The advent of service-oriented architectures (SOA) presents an altogether more challenging dilemma. The growing interest in SOA is driven by the promise that it will allow businesses to achieve broad-scale interoperability of their software systems (through service reuse and process agility), while maintaining the flexibility required to continually adapt these systems to changing business needs. In 2004, Leavitt cited a report predicting that worldwide spending on web service-based software projects would increase ten-fold in the five years to 2008, to around $11 billion [2]. But SOA represents a paradigm shift in the way business functionality is implemented and delivered. Thus, the potential benefits of services and SOA threaten to make the tremendous investment in existing systems obsolete. However, we believe that SOA provides a viable means for industry to support changes in business while leveraging past IT investments, through a process of progressive evolution rather than wholesale replacement. The challenge is to understand the issues that must be addressed if the migration of existing systems to SOA is to be successful.

The remainder of this paper is organized as follows: Section 2 outlines the main advantages of service, which are prompting so much attention in business, and considers how the motivating factors are likely to result in "hybrid" systems; Section 3 gives our summary of the key challenges that threaten the migration process; Section 4 described some process-oriented approaches to evolving existing systems and outlines a combined strategy. Section 5 provides some concluding remarks.

2. Adopting SOAs

2.1. The Vision

SOAs present a compelling vision for businesses. Conceptually, services bring together a layer of business functionality and a layer of technological implementation. Technologically, Brown et al [3] provide an excellent summary of what services *are*, whilst suggesting that it is not the individual features that matter, but the aggregation of them. So, we expect services to be "coarse grained", "discoverable", "loosely coupled", etc. From a software engineering perspective, services are the embodiment of interface-based design – and thus can be seen as progression of a trend that has brought modular design, object-orientation and components.

From a business perspective, services are about appropriate packaging of functionality and flexibility. Capturing system knowledge in a way that is appropriate for business users and developers is difficult [4], but services provide a mechanism for packaging functionality in meaningful unit for development, provision, sale and consumption. Moreover, they do so in a way, and with a business model, that affords a high degree of flexibility to provider and consumer alike. It is for this reason that they promise to allow businesses to become more responsive than ever to the needs of individual customers and markets. However, it is not just business systems that promise to benefit from the service model, it is envisaged that embedded systems will also be able to augment their functionality in the face of unusual, or even exceptional, circumstances. It is not surprising that business leaders identify services and service-oriented computing (SOC) with acquiring and maintaining business advantage.

The problem that accompanies a major shift in the way business functionality is packaged and offered is that it threatens to make what already exists obsolete, even when existing systems represent massive investment. This effect is compounded in the case of services because they appear to offer freedom from such a legacy "tie-in": if a new service provider offers a new improved service, you simply change service provider. Of course, much of the hype surrounding the arrival of service and service-oriented marketplace both fuels and feeds upon these issues.

2.2. Understanding the motivations for using SOAs

The problem still remains, though: how should existing systems, in many cases providing core, or even critical, business functionality, be migrated to SOAs? Although there is probably no single answer, it is necessary to unpack the likely motivations for businesses wishing to adopt SOAs. For many businesses, the true value of services is not the possibility of "dynamic service discovery" and "late binding". Instead, it is the ability to rationalise their existing systems into meaningful chunks of business functionality that can be reconfigured easily and quickly to exploit new business opportunities. In other words, the relative immaturity of the standards for service discovery and service-centric system engineering is not necessarily an impediment to SOA adoption. The real hindrance is the lack of methods for the daunting task of unravelling the architectures of existing systems.

Although the technical challenges of "re-factoring" a substantial, mission critical system are considerable, the associated business challenges are just as great. Companies must embark upon a thorough self-examination to determine which business processes, supported by existing/legacy systems, need to be liberated as services in an SOA.

Appreciating the need for this profound business process analysis task allows us to identify the "chicken and egg" nature of services and SOA adoption: businesses will begin to use service technologies internally because of the advantages that the SOA paradigm can bestow. When they do so, they will proceed by representing their key business processes as candidate services. In some case, when these processes are implemented as services, businesses will identify new revenue streams associated with providing their services to external consumers. Conversely, businesses will also identify which of their business processes do not align with their "core competencies", making them prime candidates for out-sourcing (essentially equating to "dogs" in a growth-share matrix). In an SOA world, replacing in-house services with 3rd party provided services should be seamless and painless.

We believe, then, that the answer to how to migrate existing systems to SOAs lies in progressive evolution, involving possibly many intermediate stages where core existing system functionalities are integrated into what amount to SOAs. Initially, this may involve adding functionality as a service, but progressively, obsolete functionality will be replaced by more and more independently implemented services. We can characterise the many forms of intermediate system as "hybrid".

2.3. Hybrid systems

In reality, probably most of the software systems supporting global business are "hybrid" systems. However, our concern here is with the particular issues associate with integrating existing/legacy systems with services and SOAs. An underlying assumption is that an existing system will continue to operate in conjunction with some sort of service-oriented system. This could mean that systems operate in parallel; the existing system providing some subset of business functionality and the new service-based system providing the novel functionality and the two do not interact. This is not what we envisage. Instead, it is expected that part of all of an existing system will be integrated with new functionality that is implemented as services. The new functionality may be developed

and operated in-house, or it may be consumed from an external provider.

Given the different types of system that exist in the installed software base, their form and function and their potential to be evolved for further use in a potentially infinite number of new scenarios, there are a myriad different types of systems that will result from such evolution strategies. However, particular types of system are likely to be prevalent. Their nature will depend on the relationship between the provider and consumer of the service element, and the treatment of the existing system. Although obviously a simplification, such a consideration gives four distinct types of system as shown in Fig. 1.

	Services: Provider/Consumer Relationship	
	Same (Internal)	Different (External)
Existing System: "As is"	Type 1 (ad hoc)	Type 2 (hybrid)
Existing System: "Servicised"	Type 3 (hybrid)	Type 4 (SOC)

Figure 1. Evolved system types

We can consider the nature of these different types separately:

Type 1: Combining parts of an existing system with additional software elements that are implemented internally as services will result from an attempt to use services as an implementation mechanism only. The primary benefits will be the adoption of an interface-driven development strategy for the new functionality, and the availability of a set of standards and protocols to *guide* the development. The term "guide" is used to highlight the ad hoc nature of the development process: difficulties may be overcome by use of non-standard procedures. It is most probable that no deconstruction of a business' processes will have been undertaken if this form of development is followed. Such an ad hoc strategy will not deliver the main potential benefits of services and SOAs but will represent a "first step" into a service world.

Type 2: This type of hybrid system imposes stricter adherence to the norms and expectations of SOC. The externally provided service cannot be adjusted to overcome difficulties and thus the existing system may require deeper modification to make it compatible.

This strategy may be capable of delivering some of the "off-the-shelf" benefits of services and SOAs but will not result in the difficult adoption of a genuine business service culture. Core business functionalities will remain static and fixed but benefits from the use of externally provided services may generate enthusiasm for and commitment to further adoption of SOAs.

Type 3: "Servicising" the existing systems (e.g., wrapping systems to offer functionality as a set of service-based operations) for use with internally provided services suggests a much greater commitment to SOC than Type 1 systems. However, control over provision and consumption still affords greater flexibility in the face of problematic difficulties (e.g. the statefulness or otherwise of the resulting services). The key factor here is whether the "servicisation" process is a "lip service" provision of a service interface or a thorough re-alignment of existing system provision with identified business services. In the latter case, the use of internally developed services to extend functionality is incidental to the commitment to adopt SOA.

Type 4: This represents a wholehearted commitment to adopting SOC within an organisation, especially if it represents the culmination of the business service analysis process described above.

The particular strategy adopted will affect the resulting system, but all to one degree or another share a "hybrid" nature. The inherently greater constraints imposed on Type 4 systems should mean that the most profound problem is the identification and wrapping of existing system elements as a set of services that operate as services is "expected to operate" (e.g. Brown's characterisation [3]). The greater flexibility available for Type 1 systems may make problems easier to overcome, but may result in issues that affect maintainability into the future. Type 2 and Type 3 systems share constraint and flexibility in equal measure.

It can be seen, though, that "progressive evolution" could amount to the gradual shift from a Type 1 system, through Types 2 and 3, to a Type 4 system. Whether such a strategy would deliver the necessary business benefits would depend on the circumstances, but, for some businesses, it might represent a lower risk migration route to SOAs.

Whether systems of these types will behave as expected raises some important questions. Experience in the component-based software engineering world suggest that there will be some significant challenges to overcome, particularly in the area of architectural mismatches.

3. Architectural mismatch challenges

If we accept that the most pragmatic way to exploit services whilst preserving the investment of the installed software base is some kind of progressive evolution towards SOAs, and then we have seen that most forms of integration of existing systems with

services result in what we can only understand as hybrid systems, we need to consider the viability of such systems. In many cases, such systems are becoming the de facto development paradigm [9], but it should not be assumed that there are no associated difficulties. The similarities between services and software components raise some important issues.

Garlan *et al* [10] describe a number of significant challenges that such systems face. They concluded that the integration of independently developed software elements usually results in the following deficiencies:

- ***Code bloat***: Interacting programs may grow excessively large in size.
- ***Poor performance.*** This is the result of the excessive code size and the communication overhead caused by architectural mismatches.
- ***Need to modify the existing components***: Integrated software systems usually have subtle incompatibilities or deficiencies that required considerable time to understand and remedy.
- ***Need to reimplement existing functions***: Even if a capability is present in an existing component, it may be sometimes necessary to reimplement it in order to cooperate with other components.
- ***Unnecessarily complex code***: Simple sequential programs often must become multithreaded tools because of the need to provide concurrent access to clients.
- ***Error-prone construction process***: Building a system from its sources can be a very time-consuming process, due to the high degree of interdependence between the various components.

The problems can be traced back to architectural mismatch (i.e. by conflicts between the architectural assumptions made by the various components). In order to understand architectural mismatch, it is helpful to view a system as made up of components (the high-level computational and data storage entities in the system) and connectors (the interaction mechanisms among the components). There are four primary categories of assumptions that can lead to architectural mismatch:

1. ***Nature of components:***

- ***Infrastructure***: The assumptions a component makes about the underlying support it needs to perform its operations. This support takes the form of the additional infrastructure that the component either requires or provides in the form of operating system, middleware, additional libraries and other components.

  One of the main problems here is that many software technologies do not require to explicitly document the *requires interfaces*. A prominent example is object-oriented technology where only the *provides interfaces* are documented.
- ***Control model***: One of the most serious problems are the assumptions made about what component holds the main thread of control and how individual components control the sequencing of actions. This problem is especially serious if a number of components, each holding its own event loop, are integrated into the same process, as is often the case for services. This may be a particular problem if existing, or legacy, system elements are wrapped as services.
- ***Data model***: Even if simple conversions of the data format are performed by the underlying runtime libraries, assumptions about the nature and organization of the data a component will handle remain critical.

2. ***Nature of connectors:***

- ***Interfaces***: At the syntactic level, interface mismatches are quite easily solved by the introduction of glue software in the form of wrappers and proxies. The semantic level is more subtle and requires careful analysis. The problem here is that the semantics of cooperating components are often not specified at all, only informally specified or formally specified by different formalisms (e.g. pre/post-conditions and ontologies). The first two cases might result in a considerable test effort, while the compatibility of specification mechanisms might be a source of nasty problems in the third case.
- ***Protocols***: Once the interfaces are made compatible, assumptions about the sequence of actions (the protocol) constitute the next problem. Almost all interfaces require particular sequences, be it only that a component must be initialized before it can be used. More subtle is the handling of message sequences for a mix of synchronous calls and events (e.g. generated by a publish/subscribe mechanism). This problem is very relevant for services that often use both communication mechanisms. This means that the requester must do some bookkeeping in order to properly pair requests and responses.
- ***Data model***: Just as the components make assumptions about the kind of data the components will manipulate, so also do they make assumptions about the data that will be communicated over the connectors. The call parameters of different components can be of different types , requiring the introduction of additional translation routines.

3. ***Global architectural structure:***

- ***Topology of the system's communication structure***: Entities that are central to a collection of components often assume a star structure with no direct interaction between the other participants. For services, this is referred to as *orchestration* pattern. The problem arises if other components assume direct component-to-component communication. This corresponds to the *choreography* pattern of interaction. Conflicts between these two interaction patterns can easily result in blocking and deadlocks.
- ***Presence or absence of particular components or connectors***: If a composition of components is not carefully modelled it is possible that not all elements will be available. This is especially an issue given the late-binding nature of SOAs.

4. ***Construction process:***
   Conflicting assumptions about the order in which the components and connectors must be combined to build the system form another hurdle.

- ***Deployment dependencies***: If the underlying platform does not support shared code and resources, these may have to be duplicated.
- ***Runtime dependencies***: A similar problem occurs at runtime if different components make different assumptions about the sequence in which other entities are instantiated.

Gacek and Boehm [11] also identify a set of conceptual architectural features that can give rise to mismatches, such as dynamism, concurrency, distribution, encapsulation, predictable response time and re-entrance. This set of architectural features is less generally applicable to SOC, but illustrates that architectural assumptions may be complex and not readily understood. Successful integration of existing systems and separately developed services will require very careful analysis of the assumptions made on both sides.

4. Process approaches and system re-engineering

A number of different methods and strategies have been described for evolving systems that are, in part, applicable to the problem of migrating existing systems to SOAs. Here we provide brief summaries of three approaches, which address different issues.

4.1. Renaissance

In response to an appreciation of both the functionality offered by existing systems and the investment that they represent, the Renaissance method [5] presents a set of strategies that put reengineering above replacement. This is the implicit foundation of any approach that proposes any form of progressive evolution. The four key requirements that motivate the approach (Table 1) help to identify how it can contribute to the evolution of existing systems to SOA.

Table 1. Renaissance requirements.

1	The method should support incremental evolution.
2	Where appropriate, the method should emphasise reengineering, rather than system replacement.
3	The method should prevent the legacy phenomena from reoccurring.
4	It should be possible to customise the method to particular organisations and projects.

Many of the specific details of Renaissance are beyond the scope of this summary. However, having identified the dilemma that exists between maintenance and replacement [6], the method stresses that an effective way of mitigating the costs and risks associated with replacement, system re-engineering – especially with a view to ongoing system development – is an attractive approach.

Table 2. Renaissance evolution strategies.

Continued Maintenance	The accommodation of change in a system, without radical change to its structure, after it has been delivered and deployed.
Revamp	The transformation of a system by modifying or replacing its user interfaces. The internal workings of the system remain intact, but the system appears to have changed to the user.
Restructure	The transformation of a system's internal structure without changing any external interfaces.
Rearchitecture	The transformation of a system by migrating it to a different technological architecture
Redesign with Reuse	The transformation of a system by redeveloping it utilising some of the legacy system components.
Replace	Total replacement of a system.

Renaissance goes on to list six evolution strategies (Table 2). Examination of these strategies reveals something quite interesting with respect to evolving an existing system to SOA: namely that *all* of these strategies could contribute to successful evolution of this sort. This limits the direct applicability of Renaissance as it is currently expresses, but does not diminish the importance of the recognising that progressive evolution should contain an element of reengineering as part of the evolution approach.

What Renaissance does appear to lack is an explicit recognition of the business context. Thus the evolution strategies are primarily selected on technical and organisational grounds (e.g. availability of system knowledge, documentation, etc.).

4.2. COMPOSE

There is an obvious parallel between services and software components, particularly commercial-off-the-shelf (COTS) components. A process for evolving an existing system using COTS components might be a good candidate for application to evolving systems to SOAs. Kotonya and Hutchinson [7] describe the use of the Component-Oriented Software Engineering (COMPOSE) method to evolve a legacy freight tracking system so that it supported the demanding requirements of the company's larger customers. Again, the specific details of the process are beyond the scope of this paper, but the following aspects of COMPOSE important:

1. COMPOSE interleaves planning and negotiation, development and verification. The purpose of this is that many of the challenges of utilizing COTS components stem from limitations of available documentation. Verification embeds activities that check the viability of the system at every stage, whilst negotiation allows for corrective action.
2. COMPOSE incorporates a viewpoint (VP)-oriented requirements approach [8]. VPs provide an excellent mechanism for modelling legacy system elements, as well as other concerns, as service-consumers.
3. COMPOSE uses the notions of service providers and service consumers as an integral model of the system being developed. Required "services" are used to map between system requirements and available components. There are few significant differences between third party services with COTS components from this perspective.

These aspects of COMPOSE mean that it can be used to model an existing system as a series of refined sub-systems that provide and consume services. The resulting model can then be used as, essentially, a roadmap for progressive evolution.

A potential weakness of applying COMPOSE to progressive evolution of existing systems to SOAs is that it doesn't *explicitly* address the entire business context of the proposed activity. Also, although the viewpoint approach can be used to capture significant details of the existing system context, as described, it is intended to do so only insofar as is required to express the requirements/constraints on the extended development. That said, the use of services as a modelling construct promises a great deal for deconstructing the functionality of an existing/legacy system.

4.3. SOSA

In the vast majority of cases, the need to utilize an SOA is part of a process that is not itself technology-led. In other words, there are external reasons for wanting to adapt an existing system so that it can operate in a SOA, and there are some aspects of the resulting challenge that relate more to those reasons than to the technical challenge.

The Service-Oriented Solutions Approach (SOSA) [9] attempts to address these. Again, many of the details are not relevant, but there are a number of interesting elements, including:

- **Critical Business Issues**. SOSA recognises that the organization that is considering a SOA solution to its system needs is doing so because it has identified critical business issues that have to be addressed. This reminds us that:
  - The system is being developed to implement some sort of business strategy, not as an end in itself.
  - The details of the technical problem are probably not important in themselves, only insofar as they affect the business.
  - An entirely viable technical solution may ultimately be rejected for business reasons (e.g. expensive, too long to develop, etc); similarly, business priorities may favour an inelegant technical solution.
- **Business Process Improvement**. The rationale for the development activities are determined as part of a business process improvement exercise, which involves modelling the existing process, determining the changes that should be made to solve the relevant critical business issues and an explicit attempt to estimate the return on investment (ROI) associated with the proposals. Of particular interest here are:

  - Note the emphasis on the business context.
  - The modelling activity. Even when analysts and developers are familiar with the system being adapted, this activity is necessary as proposals for solutions are sought. However, in the very worst cases of embedded legacy systems, this activity will amount to a type of reverse engineering, potentially providing a model and level of understanding of the system that has long been lost.
- **Enterprise Service Architecture** (ESA). This is effectively a plan for the organisation's business services bus. SOSA's ESA identifies a set of IT services that:
  - Are derived from an enterprise-wide business type model;
  - Offer operations that are business process-neutral as well as being user interface-independent.

Importantly, once developed, this ESA can act as a road map for an incremental, or progressive evolution process where functionality that is provided by existing, legacy, systems is moved to service-based provision. SOSA is primarily intended for companies which intend to implement their SOA using "bespoke" development. As such, it does not explicitly address the challenges of using third party services, nor the process of providing service interfaces to existing systems.

4.4. A combined approach

The three approaches discussed here present some interesting perspectives on the migration to SOA challenge. None of the approaches provides a process that addresses all of the challenges. However, together, they appear to highlight many of the important issues.

Remember that the process of adopting services and SOAs was characterised as both a technical process and a business process. The importance of this was highlighted when looking at different types of hybrid systems. For a business to fully engage in migration to SOA, it must be prepared to "servicise" its existing systems, because it is these systems that support the core business processes. SOSA provides some pointers for achieving this process. The enterprise service model, if adequately mapped on to the functionalities provided by existing/legacy systems, goes some way to identifying a business' key processes. What SOSA does not offer is a mechanism for providing such a mapping.

Renaissance provides some important pointers for determining the viability of reengineering an existing system into services. If the SOSA ESA were used as a further input to the Renaissance process, it might provide useful insights into the feasibility of the "servicisation" process.

COMPOSE could be used to model an ESA and can support the process of mapping service definitions onto "components" than can deliver those services. As such, it could be used to express an ESA that is effectively delivered by one of more existing systems. However, it does not offer explicit support for identifying business services provided by such systems.

5. Conclusion

We believe that the progressive evolution of existing systems to SOA and the resulting hybrid systems are a persuasive way forward to ensure a continued realisation of investment in existing systems – and an avoidance of costs and risk associated with wholesale replacement. However, the lessons learnt in the area of component-based systems suggest that there are significant problems when trying to integrate components, or services, from different sources. Appropriate approaches for progressive evolution of existing systems must address these challenges. We have described a number of approaches that address different aspects of the challenges faced and this suggests that the problems are not insurmountable. However, it is important to recognise that appropriate business processes for migrating to SOAs are as important as technical processes. The potential architectural problems simply make the realisation of those business processes all the more challenging.

6. Acknowledgements

This work is partly funded by the SeCSE project (IST 511680) and we are grateful for the contributions of our partners.

7. References

[1] M.M. Lehman and L. Belady, Program Evolution: Processes of Software Change. London: Academic Press. 1985.

[2] N. Leavitt, "Are Web Services Finally Ready to Deliver?" IEEE Computer, 37(11), 14-18, 2004.

[3] A. Brown, S. Johnston and K. Kelly, "Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications", Cupertino, CA: Rational Software Corporation (IBM) White Paper, October 2002.

[4] D. Dhungana, R. Rabiser, P. Grünbacher, H. Prähofer, Ch. Federspiel and K. Lehner, "Architectural Knowledge in Product Line Engineering". Proc of the 32nd EUROMICRO

Conference on Software Engineering and Advanced Applications, Croatia, September 2006.
[5] I. Warren and J. Ransom, "Renaissance: A Method to Support Software Systems Evolution", Proc of 26th Annual International Computer Software and Applications Conference (COMPSAC), Oxford, UK, pp.415-420, August 2002.
[6] K. Bennet, “Legacy Systems: Coping with Success”. IEEE Software, 12(1). 1995.
[7] G. Kotonya and J. Hutchinson, "A COTS-Based Approach for Evolving Legacy Systems", to appear in Proc of the 6th IEEE International Conference on COTS-based Systems (ICCBSS 2007), Canada, February 26 - March 2, 2007.
[8] G. Kotonya and J. Hutchinson, "Viewpoints for Specifying Component-Based Systems", in Proc of the International Symposium on Component-based System (CBSE7), LNCS Vol 3054, Edinburgh, UK, May 2004.
[9] “Hybrid System Development”, Service Centric System Engineering (SeCSE) Project (IST 511680) Document A3.D7. (http://secse.eng.it) 2006.
[10] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch, or, Why it's hard to build systems out of existing parts”, IEEE Software, 12(6), Nov. 1995.
[11] C. Gacek, and B. Boehm, "Composing Components: How Does One Detect Potential Architectural Mismatches?," in Proceedings of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January 1998