The Challenge of Evolving Existing Systems to Service-Oriented Architectures

John Hutchinson, Gerald Kotonya, James Walkerdine, Peter Sawyer, Glen Dobson and Victor Onditi

Abstract—**Software systems are an integral part of industrial processes at every level, from low-level production control to enterprise planning. The maintenance challenge presented by such systems is about finding an acceptable balance between risk involved in evolving the system and benefits offered by the update. Service-Oriented Architecture (SOA) promises to leverage software systems to become more efficient and responsive to change through service reuse and process agility. However, for existing systems, this compounds the maintenance problem as SOA represents a "paradigm-shift". It leaves business leaders facing a difficult problem: how to minimise the risk to their investment (existing software systems) and exploit the benefits of migrating a SOA. We describe a pragmatic strategy for addressing the problem and outline the significant challenges that remain.**

I. Introduction

SOFTWARE systems are an integral part of industrial processes at every level, from low-level production control to enterprise planning. This suggests a strong relationship between a business' success and the "fitness for purpose" of the software systems that it relies on. 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]. This poses the difficult problem of how to maintain and extend existing systems that support critical business processes. The situation is further complicated when a change involves migration to a new development paradigm. The advent of service-oriented architectures (SOA) presents such a change. 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].

That said, the adoption of service oriented computing (SOC) by industry will bring with it a series of new activities that are peculiar to the paradigm, such as discovery, coordination (orchestration/choreography), etc. Although they represent significant challenges in themselves, they do not explicitly contribute to the maintenance of investment that is central to the aim of evolving existing systems to SOAs.

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. However, we also believe that there are significant challenges 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 services, which are prompting so much attention in business; Section 3 describes some process-oriented approaches to evolving existing systems; Section 4 presents a brief analysis of the nature of systems that integrate existing systems with services, which leads to our summary, in Section 5 of the key challenges that threaten this process. Section 6 provides some concluding remarks.

II. A Service-Oriented Future?

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 software components.

From a business perspective, services are about appropriate packaging of functionality and flexibility.

Manuscript received January 25, 2007. This work was supported in part by the SeCSE project (EU IST 511680).

J. Hutchinson is with the Lancaster University Computing Department, Lancashire, UK (e-mail: hutchinj@comp.lancs.ac.uk).

G. Kotonya is with the Lancaster University Computing Department, Lancashire, UK (e-mail: gerald@comp.lancs.ac.uk).

J. Walkerdine is with the Lancaster University Computing Department, Lancashire, UK (e-mail: walkerdi@comp.lancs.ac.uk).

P. Sawyer is with the Lancaster University Computing Department, Lancashire, UK (e-mail: sawyer@comp.lancs.ac.uk).

G. Dobson is with the Lancaster University Computing Department, Lancashire, UK (e-mail: dobsong@comp.lancs.ac.uk).

V. Onditi is with the Lancaster University Computing Department, Lancashire, UK (e-mail: onditi@comp.lancs.ac.uk).

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 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 effectively makes 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 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.

The problem still remains, though: how should existing systems, in many cases providing core, or even critical, business functionality, be migrated to SOAs? We believe that the answer lies in progressive evolution of existing systems towards SOAs, 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.

III. Process approaches and system re-engineering

Although little work has been done explicitly on the problem of migrating existing systems to SOAs, a number of different methods and strategies have been described for evolving systems that are, in part, applicable to the problem. Here we provide brief summaries of three approaches which address different issues that arise. First we look at Renaissance, which takes a reengineering approach to maintenance. Then we describe biefly COMPOSE, a service-oriented process for developing systems using software components. Finally, we describe some aspects of business process analysis approach, called SOSA.

A. 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 place reengineering over replacement, whilst recognising that, ultimately, replacement is the evolution strategy required if all else fails. 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.

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 1
RENAISSANCE REQUIREMENTS.

No	Requirement
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.

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, but does not diminish the importance of the recognising that progressive evolution should contain an element of reengineering as part of the evolution approach.

TABLE 2
RENAISSANCE EVOLUTION STRATEGIES.

Strategy	Description
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.

B. 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 existing 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 supports the demanding requirements of a company's larger customers. The specific details of the process are beyond the scope of this paper, but the following aspects of COMPOSE important:

1) COMPOSE interleaves planning & 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, such 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.

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.

C. 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 deliver, 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 roadmap 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.

IV. HYBRID SYSTEMS

An underlying assumption in the discussion of strategies for achieving progressive evolution to SOAs 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. However, this is not what we envisage. Instead, it is expected that some combination of the evolution strategies identified in the Renaissance method will be used to integrate some part of the existing system with some new part that is implemented as a service, or a set of services.

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, very different systems will result from attempts to evolve them. 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.

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. Certainly, wider SOC activities, such as discovery, will not be relevant in these situations.

Type 2: This type of 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.

Type 3: "Servicising" the existing system (i.e., wrapping it to offer its 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).

Type 4: This represents a wholehearted commitment to adopting SOC within an organisation – a combination of the "rearchitecture", "redesign and reuse" and the "replace" strategies identified in Renaissance.

There are obvious relationships between these types of system and the evolution strategies identified in the Renaissance method and the other process-oriented

	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)

Fig. 1. Different types of system result from the approach adopted when integrating existing systems and services. Whether or not the services consumed are provided externally is also an important factor. The combination of these determines the type of the resulting system.

evolution approaches. 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 wrapping the existing system as a service that operates as a service is expected to operate. The greater flexibility available for Type 1 systems may make problems easier to overcome, by may result in issues that affect maintainability into the future. Type 2 and Type 3 systems share constraint and flexibility in equal measure.

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.

V. 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. Although this work primarily focuses on component-based systems, its value here comes about because (1) no similar analysis yet exists for service/service and service/non-service integration and (2) the defining feature was the integration of independently developed software elements. They concluded that attempting this usually results in the following deficiencies:

- *Code bloat*: Interacting programs may grew 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 elements). Note too that some of these difficulties assume that elements *can* be modified. If the existing software assets are off-the-shelf, some of the problems encountered may be insurmountable.

In order to understand architectural mismatches, 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 rather 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/postconditions 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 initialised 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 various components and connectors must be combined and instantiated 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.

VI. 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.

The range of architectural mismatches identified suggests that there is a need for active research in this area. Otherwise, businesses that urgently require the benefits promised by SOC will be left to themselves to determine how best to migrate their existing software assets, when in fact they share problems faced by many. On the one hand, any suitable process for supporting migration of existing systems to SOA will involve detailed business process analysis. On the other, it must surely also involve appropriate architectural analysis. We are currently embarked on delivering such a process and hope to report on it in the near future.

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", 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.G. O. Young, “Synthetic structure of industrial plastics (Book style with paper title and editor),” in *Plastics*, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.