An Architectural Model for Service-Based Flexible Software

Keith Bennett, Jie Xu, Nicolas Gold, Malcolm Munro, and Zhuang Hong
Department of Computer Science
University of Durham, UK
keith.bennett@durham.ac.uk

Paul Layzell
Department of Computation
UMIST, UK
paul.layzell@umist.ac.uk

David Budgen** and Pearl Brereton
Department of Computer Science
Keele University, UK
db@cs.keele.ac.uk

** address for correspondence

Keywords: *software evolution, service architectures, process models*

Abstract

The need to change software easily to meet evolving business requirements is urgent, and a radical shift is required in the development of software, with a more demand-centric view leading to software which will be delivered as a service, within the framework of an open marketplace.

We describe a service architecture and its rationale, in which components may be bound instantly, just at the time they are needed and then the binding may be disengaged. This allows highly flexible software services to be evolved in "internet time". The paper focuses on early results: some of the aims have been demonstrated and amplified through an experimental implementation based on e-Speak, an existing and available technology. It is concluded that technology such as e-Speak provides a useful infrastructure and has enabled us rapidly to demonstrate the basic operation and viability of our approach.

1. Objectives

Software maintenance has matured considerably over the past 20 years so that standards representing best practice are emerging [1, 2]. Most approaches to software maintenance derive from three basic process stages [10]: understanding the existing software; modification of the existing software; and revalidation of the modified software.

In industrial practice, these three basic stages may expand to incorporate many sub-stages. Empirically, this process has been successful in supporting the maintenance of large, risk-averse software systems (such as safety critical and business critical systems). Although the "applications backlog" has always been recognised as a problem, users have become familiar with the activity of new releases of software (perhaps every six to 24 months), in which groups of change requests, enhancements and/or bug fixes have been aggregated and made available by vendors. Procedures for "emergency fixes" can be used to resolve urgent problems, though these are a well known source of later difficulty. Even in very mature software maintenance processes, the skills of the maintainers and their system knowledge are usually critical.

In [5], a new model of software maintenance was described, in which it was argued that maintenance should be categorised according to the phase of the maintenance lifecycle. This is a more useful model with which to address the needs of many modern businesses. The internet age has ushered in a new era of highly dynamic and agile organisations which must be in a constant state of evolution if they are to compete and survive in an increasingly global marketplace. These are operating in a time-critical environment, rather than a safety critical application domain. If a change or enhancement to software is not brought to market sufficiently quickly, thus retaining competitive advantage, the organisation may collapse. This era poses significantly new problems for software development, characterised by a shift in emphasis from producing 'a system' to the need to produce 'a family of systems', with each system being an evolution from a previous version, developed and deployed in ever shorter business cycles. In other words, the evolution stage in the above model becomes central. It may be that the released new version is not complete, and still has errors. If the product succeeds, it can be put on an "emergency life support" to resolve these. If it misses the market time slot, it probably will not succeed at all.

It is possible to inspect each activity of the software evolution process and determine how it may be speeded up. Certainly, new technology to automate parts may be expected, supported by tools (for example, in program comprehension, testing etc.). However, it is very difficult to see that such improvements will lead to a *radical* reduction in the time to evolve a large software system. This prompted us to believe that a new and different way is needed to achieve ultra rapid evolution; we term this "evolution in internet time". It is important to stress that such ultra rapid evolution does not imply poor quality, or software which is simply hacked together without thought. The real challenge is to achieve very fast change yet provide very high quality software. Strategically, we plan to achieve this by bringing the evolution process much closer to the business process.

In 1995, British Telecommunications plc (BT) recognised the need to undertake long-term research leading to different, and possibly radical, ways in which to develop software for the future. Senior academics from UMIST, Keele University and the University of Durham, came together with staff at BT to form DiCE (The Distributed Centre of Excellence in Software Engineering). The outcome of this research is summarised in Section 2 of the paper. This work established the foundations for the research described in this paper. In Section 3, we express the objectives of the current phase of research in terms of the vision for software - how it will behave, be structured and developed in the future. From 1998, the core group of researchers switched to developing a new overall paradigm for software engineering: a service-based approach to structuring, developing and deploying software. This new approach is described in the second half of this paper. In section 4, we describe a prototype implementation of the service architecture, demonstrating its feasibility and enabling us to elucidate research priorities. In addition, we are exploring technologies in order to create a distributed laboratory for software service experiments.

2. Developing a Future Vision

The method by which the DiCE group undertook its research is described in [4]. Basically, the group formulated three questions about the future of software: How will software be used? How will software behave? How will software be developed? In answering these questions, a number of key issues emerged.

K1. Software will need to be developed to meet **necessary and sufficient requirements**, i.e. for the majority of users whilst there will be a minimum set of requirements software must meet, over-engineered systems with redundant functionality are not required.

K2. Software will be **personalised**. Software will be capable of personalisation, providing users with their own tailored, unique working environment which is best suited to their personal needs and working styles, thus meeting the goal of software which will meet necessary and sufficient requirements.

K3. Software will be **self-adapting**. Software will contain reflective processes which monitor and understand how it is being used and will identify and implement ways in which it can change in order to better meet user requirements, interface styles and patterns of working.

K4. Software will be **fine-grained**. Future software will be structured in small simple units which co-operate through rich communication structures and information gathering. This will provide a high degree of resilience against failure in part of the software network and allow software to re-negotiate use of alternatives in order to facilitate self-adaptation and personalisation.

K5. Software will operate in a **transparent** manner. Software may continue to be seen as a single abstract object even when distributed across different platforms and geographical locations. This is an essential property if software is to be able to reconfigure itself and substitute one component or network of components for another without user or professional intervention.

Although rapid evolution is just one of these five needs, it clearly interacts strongly with the other demands, and hence a solution which had the potential to address all the above factors was sought.

3. Service-Based Architecture

3.1 The Problem

Most software engineering techniques, including those of software maintenance, are conventional supply-side methods, driven by technological advance. This works well for systems with rigid boundaries of concern such as embedded systems. It breaks down for applications where system boundaries are not fixed and are subject to constant urgent change. These applications are typically found in **emergent organisations**--"organisations in a state of continual process change, never arriving, always in transition" [6]. Examples are e-businesses or more traditional companies which continually need to reinvent themselves to gain competitive advantage [7]. These applications are, in Lehman's terms, "E-type" [9]; the introduction of software into an organisation changes the work practices of that organisation, so the original requirements of the software change. It is not viable to identify a closed set of requirements; these will be forever changing and many will be tacit.

Subsequent research by DiCE has taken a **demand-led** approach to the provision of software services, addressing delivery mechanisms and processes which, when embedded in emergent organisations, give a software solution in emergent terms -- one with continual change. The solution never ends and neither does the provision of software. This is most accurately termed *engineering for emergent solutions*.

We concluded that a "silver bullet", which would somehow transform software into something which could be changed far more quickly than at present, was not viable. Instead, we took the view that software is actually hard to change, and this takes time to accomplish. We needed to look for other solutions.

3.2 Service Architecture

Currently, almost all commercial software is sold on the basis of ownership (we exclude free software and open source software). Thus an organisation buys the object code, with some form of licence to use it. Any updates, however important to the purchaser, are the responsibility of the vendor. Any attempt by the user to modify the software is likely to invalidate warranties as well as ongoing support. In effect, the software is a black box that cannot be altered in any way, apart from built-in parameterisation. This form of marketing is known as supply-led. It is the same whether the software is run on the client machine or on a remote server, or, if the user takes on responsibility for in-house support or uses an applications service supplier (i.e. outsources maintenance).

Let us now consider a very different scenario. We assume that our software is structured into a large number of small components (see K1, K4, K5 above), which exactly meet the user's needs and no more. Suppose now that a user requires an improved component C. The traditional approach would be to raise a change request with the vendor of the software, and wait for several months for this to be (possibly) implemented, and the modified component integrated.

In our solution, the user disengages component C, and searches the marketplace for a replacement C' which meets the new needs. When this is found, it is bound in instead of C, and used in the execution of the application. Of course, this assumes that the marketplace can provide the desired component. However, it is a well established property of marketplaces that they can spot trends, and make new products available when they are needed. The rewards for doing so are very strong and the penalties for not doing so are severe. Note that any particular component supplier can (and probably will) use traditional software maintenance techniques to evolve their components. The new dimension is that they must work within a demand-led marketplace. Therefore, if we can find ways to disengage an existing component and bind in a new one (with enhanced functionality and other attributes) ultra rapidly, we have the potential to achieve ultra-rapid evolution in the target system.

This concept led us to conclude that the fundamental problem with slow evolution was a result of software that is marketed as a product, in a supply-led marketplace. By removing the concept of ownership, we have instead a service i.e. something that is used, not owned. Thus we widened the component-based solution to the much more generic service-based software in a demand led marketplace.

This **service-based model of software** is one in which services are configured to meet a specific set of requirements at a point in time, executed and disengaged - the vision of instant service, conforming to the widely accepted definition of a service:

> "an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production" [8].

Services are composed out of smaller ones (and so on recursively), procured and paid for on demand. This is a radically new industry model, which could function within markets ranging from a genuine open market (requiring software functional equivalence) to a *keisetzu* market, where there is only one supplier and consumer, both working together with access to each other's information systems to optimise the service to each other.

This strategy enables users to create, compose and assemble a service by bringing together a number of suppliers to meet needs at a specific point in time. An analogy is selling cars: today manufacturers do not sell cars from a pre-manufactured stock with given colour schemes, features etc.; instead customers configure their desired car from series of options and only then is the final product assembled. This is only possible because the technology of production has advanced to a state where assembly of the final car can be undertaken sufficiently quickly.

Software vendors attempt to offer a similar model of provision by offering products with a series of configurable options. However this offers extremely limited flexibility -- consumers are not free to substitute functions with those from another supplier since the software is subject to *binding* which configures and links the component parts, making it very difficult to perform substitution. The aim of this research is to develop the technology which will enable *binding* to be delayed until the execution of a system. This will enable consumers to select the most appropriate combination of services required at any point in time.

However *late binding* comes at a price, and for many consumers, issues of reliability, security, cost and convenience may mean that they prefer to enter into contractual agreements to have *some early binding* for critical or stable parts of a system, leaving more volatile functions to late binding and thereby maximising competitive advantage. The consequence is that any future approach to software development must be interdisciplinary so that non-technical issues, such as supply contracts, terms and conditions, and error recovery are addressed and built in to the new technology.

3.3 Bind Once, Execute Once

A truly service-based role for software is far more radical than current approaches, in that it seeks to change the very nature of software. To meet users' needs of evolution, flexibility and personalisation, an open market-place framework is necessary in which the most appropriate versions of software products come together, are bound and executed as and when needed. At the extreme, the binding that takes place just prior to execution is disengaged immediately after execution in order to permit the 'system' to evolve for the next point of execution. Flexibility and personalisation are achieved through a variety of service providers offering functionality through a competitive market-place, with each software provision being accompanied by explicit properties of concern for binding (e.g. dependability, performance, quality, licence details etc).

A software component is simply a reusable software executable. Our *serviceware* clearly includes the software itself, but in addition has many non-functional attributes, such as cost and payment, trust, brand allegiance, legal status and redress, security and so on. Binding requires us to negotiate across all such attributes (as far as possibly electronically) to establish a binding, at the extreme just before execution. This is seen as imperative in a business to business e-commerce environment.

4. Service Implementation – Prototype and Results

4.1 Aims and the model

This section describes the objectives of the experimental system, the rationale for using e-Speak, the results obtained from the implementation, and the conclusions drawn.

The general aim of this prototype was to test ideas about dynamically bound services at run-time within the flexible software service architecture. An earlier experimental prototype concentrated on the problem of service binding with limited negotiation [12]. We describe here a new experiment that has been undertaken using e-Speak, an existing and available technology being developed by HP [11].

The theoretical model is as follows: vendors register services in an electronic service marketplace. A service is a named entity providing either operational functionality or a *composition template* (see below). A service consumer searches the marketplace for a suitable service. Assuming such a service exists (i.e. a match can be made), the service interface is passed to a *broker service*, which is responsible (again on the fly) for utilizing the sub-services from the marketplace. This will either involve interpreting composition templates for sub-services, or using a service that actually delivers a result. The broker will discover and use the most useful sub-service that meets the composition criteria at the *time of need*. Note that the service composition (the design activity) is *not* undertaken by the client or user, but the templates are supplied by vendors in the marketplace.

It can be seen that the architectural model offers an extreme view of late binding; services are dynamically composed at the instant of need and then disengaged afterwards. Of course this raises the question of a service request for which there is no offering in the marketplace. Although in the long term there may be technological help for automatic composition (e.g. using reflection), currently we see this as a *market failure*; the market has been unable to provide the needs of a purchaser.

It is important to distinguish binding and service composition. For example, we may wish to compose a payroll service from sub-services such as "obtain tax codes", or "print payslips". The *design* of this composition is a highly skilled task which is not yet automatable, and there is no attempt at on-the-fly production of designs. However, we can foresee the use of variants or design patterns in the future. We call this design a *composition template*. Once it exists, we can populate the composition template with services from the marketplace which will fulfill the composition. Our architecture offers the possibility of locating and binding such services (possibly following some negotiation) as the payroll service is executed. There is no concept of producing an entire executable for an application; instead, the application is constructed on the fly at the time of need from sub-services. So from one use of the service to the next, we could replace (i.e. evolve) a "print payslips" service to an "email payslips" service as long as it retains a similar external service offering.

Our first experimental system demonstrated the capability of service binding and limited service negotiation [12]. The objectives of the current prototype were to investigate two further aspects of the above theoretical model:

1. Service discovery
2. Service binding

4.2 The e-Speak system

Our initial interest in e-Speak was triggered by its availability in complete form via a download from the HP web site. It offers a comprehensive infrastructure for distributed service discovery, mediation and binding for internet based applications.

e-Speak offers the following advantages as an experimental framework:

- A basic name-matching service discovery environment, with an exception mechanism if no service can be found.
- Issues of distribution and location are handled through *virtualisation*.
- It is based on widely used systems such as Java and XML.

It also has the following drawbacks:

- The dynamic interpretation of composition templates and subsequent binding in our theoretical model need to be implemented outside the core e-Speak system.
- The discovery mechanism does not support a more flexible scheme than name matching.
- It intercepts all invocations of services and clients, potentially resulting in supplier lock-in for organizations using the system.

4.3 The prototype implementation

Our first prototype addressed the service binding and negotiation issues, so it was decided to develop a second prototype to explore discovery and binding using a commercial platform.

A simple client application was implemented on the e-Speak platform. The application requests a high-speed printing service with a specified speed requirement.

Whilst the theoretical model assumes that a service name also describes the required functionality, e-Speak assumes that a client application knows the name of a previously existing service before it contacts the e-Speak system. A theoretical client does not necessarily expect to find the required service in the marketplace; an e-Speak client does. The client connects itself to e-Speak and asks it to find the named service. The dynamic behaviour supported by e-Speak is essentially:

- Using the service name to locate a (remote) server that provides that service.
- Locating an appropriate server for the desired implementation if several exist.
- Returning an exception to the client, if the service is no longer available.

It is not possible for e-Speak to locate a service for the client just based on the description of the client's request; it must be a precise name of the service. Note that the theoretical model assumes that the name *is* equivalent to the description and does support this.

The e-Speak approach allows a single registration and discovery mechanism for both composite and leaf services. This supports our recursive model (Section 3.2) for service composition. The key to the implementation is a class, written outside e-Speak, called DGS (Dynamically Generated Service). When a service – a service composition is returned from the discovery process, the DGS interprets it to invoke sub-services.

For the client application that requests a printing service, the e-Speak engine first attempts to locate a single printing service on a remote host. However, no specification of a single printing service satisfies the speed requirement from the client at the point of need, although a service composition does meet the requirement. Therefore, instead of returning the stub of a single service back to the client, a service composition template is returned.

In this case, the client application invokes the service by sending the composition template to DGS (note that, normally, the client would use the stub of a service to invoke, via e-Speak, the service on a remote host.) DGS serves as a broker and actually invokes three sub-services provided on three distributed printers A, B and C. Under the control of DGS, the original printing task is executed in parallel on the three printers, in a coordinated fashion. The required printing speed is therefore achieved by this composed service.

4.4 Prototype Results and Conclusions

The aim of the prototype experiment has been to explore the feasibility of using a commercial platform, e-Speak, to build two aspects of our architectural model: the dynamic binding and service discovery.

The e-Speak system allows services to be registered and discovered through vocabularies. It does not support the much more powerful and flexible on the fly discovery and binding required by our model. We had to implement this using an external object (the DGS). We also need to extend this to support negotiation on non-functional aspects, and this will have to be undertaken outside e-Speak.

Support for dynamic service binding is provided, but for our purposes, the great majority of the functionality in interpreting compositions has to be undertaken by another external object: a broker. e-Speak does not have the concept of composition templates or patterns, so these also belong externally.

e-Speak does enable a recursive service structure to be used, with fine grain services at the leaves.

5. Future Research Issues

Using the results of both prototypes, we have identified a number of major issues that need to be addressed.

Although e-Speak has saved a considerable amount of coding time, and allowed to explore some of the issues surrounding software service delivery, we do not feel that in its current form, it is an appropriate platform for the distributed laboratory. In addition to further work on the core research issues, we will continue to explore emerging platforms for service-based environments.

Requirements for software need to be represented in such a way that an appropriate service can be *discovered* on the network (handled by ontology). The requirements must convey therefore both the description and intention of the desired service. Given the highly dynamic nature of software supplied as a service, the maintainability of the requirements representation becomes an important consideration. However, the aim of the architecture is not to prescribe such representation, but support whatever conventions users and service suppliers prefer.

Automated *negotiation* is another key issue for research, particularly in areas where non-numeric terms are used e.g. legal clauses. Such clauses do not lend themselves to offer/counter-offer and similar approaches. In relation to this, the structure and definition of profiles and terms needs much work, particularly where terms are related in some way (e.g. performance and cost). Also we need insight to the issue of when to select a service and when to enter negotiations for a service. It is in this area that multi-disciplinary research is planned.

Dynamic binding at the point of need is a third issue that warrants research to understand the performance, security and fault tolerance implications of service based software.

Finally, many issues need to be resolved concerning mutual performance monitoring and claims of legal redress should they arise.

6. Conclusions

We have presented a radical and innovative architectural approach to achieving ultra rapid evolution, which is one of five key requirements identified for 21st century software. This still assumes a software maintenance approach and technology based on that used currently. However, it postulates a completely different software marketplace, which is demand-led, not supply-led. Software is marketed as a service which is used, not owned. A consumer, or supply chain service vendor integrates a number of such services to provide added value. However, the supply services are bound just before execution and disengaged afterwards, so a service can be replaced by an improved one when needed. By binding in such services, not when software is bought, but when it is executed, the software may be continually adapted to meet user requirements.

Of course, this raises many problems, and to start to answer these, a prototype implementation has been built using e-Speak. Binding to a service right at the last minute requires automatic resolution, not only of functional attributes, but also (perhaps mainly) non-functional aspects too. Also we cannot impose a grand universal scheme on service suppliers and users; in a marketplace, each must be free to adopt their own ways of business.

The prototype has given us some insight into the issues involved in specifying and delivering service-based software. We have used e-Speak to support an experimental prototype and it has formed a useful and appropriate infrastructure for some aspects of our model. In addition, it enabled us to construct the prototype in weeks rather than months. An advantage was that e-Speak did not impose a particular business model on our system (maybe service model). It has confirmed that the basic concept of software as a service is feasible, and has highlighted areas for future research.

Acknowledgements

We acknowledge the support of the Leverhulme Trust, British Telecom plc, and the UK Engineering and Physical Sciences Research Council.

References

[1] *IEEE Standard for Software Maintenance* (IEEE Std 1219-1998) in IEEE Standards, Software Engineering, Vol 2, Process Standards, 1999 Edition, IEEE 1999. ISBN 0738115606

[2] *International Standard: Information Technology - Software Maintenance* ISO/IEC 14764:1999

[3] Bennett K. H.,.Layzell P. J., Budgen D., Brereton O. P., Macaulay L., Munro M., *Service-Based Software: The Future for Flexible Software*, IEEE APSEC2000, The Asia-Pacific Software Engineering Conference, 5-8 December 2000, Singapore, IEEE Computer Society Press, 2000.

[4] Bennett K. H., Munro M., Brereton O. P. Budgen D., Layzell P. J., Macaulay L., Griffiths D. G. & Stannet C. *The future of software.* Comm. ACM, vol.42, no. 12, Dec. 1999 , pp. 78 – 84.

[5] Bennett K. H. and Rajlich V. T. *A staged model for the software lifecycle.* IEEE Computer, vol. 33, no. 7, pp. 66 – 71, July 2000, ISSN 0018-9162.

[6] Truex D.,Baskeville R. and Klein H., *Growing Systems in Emergent Organizations*, Comm.ACM, Vol.42, No.8, August 1999

[7] Cusumano M. & Yoffe D., *Competing on Internet Time – Lessons from Netscape and its Battle with Microsoft*, Free Press (Simon & Schuster) 1998

[8] Lovelock C., Vandermerwe S., Lewis B., *Services Marketing*, Prentice Hall Europe, 1996

[9] Lehman M. M. *Programs, lifecycles and the Laws of Software Evolution.* Proc. IEEE, vol. 68, no. 9, September 1980.

[10] Bennett K. H., Cornelius B. J., Munro M., Robson D. J. *Software Maintenance*. In " The Software Engineer's Reference Manual" (Ed. J. McDermid), published by Butterworth , ISBN 0-750-61040-9, 1990.

[11] Hewlett Packard. The e-Speak system is documented on http://www.e-speak.hp.com/ (valid March 2001)

[12] Bennett K. H., Gold N. E., Munro M., Layzell P. J., Budgen D., Brereton O. P . *An architectural model for service based software with ultra rapid evolution*. Submitted to IEEE ICSM 2001 (Florence, 2001)