Prototype Implementations of an Architectural Model for Service-Based Flexible Software

Keith Bennett,
Malcolm Munro, Jie Xu
Dept. of Computer Science
University of Durham, UK
keith.bennett@durham.ac.uk

Nicolas Gold, Paul Layzell
Nikolay Mehandjiev
Department of Computation
UMIST, UK
n.mehandjiev@umist.ac.uk

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

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 two experimental implementations, enabling us to assess the strengths and weakness of the approach. It is concluded that some of the key underpinning concepts – discovery and late binding – are viable and demonstrate the basic feasibility of the architecture.

1. Objectives

Contemporary organisations must be in a constant state of evolution if they are to compete and survive in an increasingly global and rapidly changing marketplace. They operate 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 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. 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 some parts (e.g. program comprehension, testing) may be expected. 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). This work established the foundations for the research described here, and its main outcomes are summarised in Section 2 of the paper.

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 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. In Section 4, we describe two prototype implementations 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 [2]. 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 software

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" [4]. Examples are e-businesses or more traditional companies which continually need to reinvent themselves to gain competitive advantage [5]. These applications are, in Lehman's terms, "E-type" [7]; 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.

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.

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*.

3.2. Service-based approach to software evolution

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 license 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 parameterization. This form of marketing (known as supply-led) applies whether the software is run on the client machine or on a remote server. A similar situation can arise whether the user takes on responsibility for in-house support or uses an applications service provider. In the latter case there is still a "black box" software, which is developed and maintained in the traditional manner, it is just owned by the applications service provider rather than by the business user.

Let us now consider a very different scenario. We see the support provided by our software system as structured into a large number of small functional units, each supporting a purposeful human activity or a business transaction (see K1, K4, K5 above). There are no unnecessary units, and each unit provides exactly the necessary support and no more. Suppose now that an activity or a transaction changes, or a new one is introduced. We will now require a new or improved functional unit for this activity. The traditional approach would be to raise a change request with the vendor of the software system, and wait for several months for this to be (possibly) implemented, and the modified unit integrated.

In our solution, the new functional unit is procured by the use of an open market mechanism at the moment we specify the change in our needs. At this moment the obsolete unit is disengaged and the new unit is integrated with the rest of the system automatically. In such a solution, we no longer have an ownership of the software product which provides all the required units of support functionality. The software is now owned by the producer of each functional unit. Instead of product owners, we are now consumers of a service, which consists of us being provided with the functionality of each unit when we need it. We can thus refer to each functional unit as a *software service*.

Of course, this vision assumes that the marketplace can provide the desired *software services* at the point of demand. 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 software supplier of software services can either assemble their services out of existing ones, or develop and evolve *atomic services* using traditional software development techniques. The new dimension is that these services are sold and assembled within a demand-led marketplace. Therefore, if we can find ways to disengage an existing service and bind in a new one (with enhanced functionality and other attributes) dynamically at the point of request for execution, we have the potential to achieve ultra-rapid evolution in the target system.

These ideas 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 traditional 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 then 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" [6].

Services are composed out of smaller ones (and so on recursively), procured and paid for on demand. An analogy is the service of organising weddings or business travel: in both cases customers configure their service for each individual occasion from a number of sub-services, where each sub-service can be further customised or decomposed recursively.

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.

3.3. Comparison with existing approaches to building flexible software

Software vendors attempt to offer a similar level of flexibility by offering products such as SAP, which is composed out of a number of configurable modules and options. This, however, offers extremely limited flexibility, where consumers are not free to substitute functions and modules with those from another supplier, because the software is subject to vendor-specific binding which configures and links the component parts, making it very difficult to perform substitution.

Component-based software development [11] aims to create platform-independent component integration frameworks, which provide standard interfaces and thus enable flexible binding of encapsulated software components. Component reuse and alignment between components and business concepts are often seen as major enablers of agile support for e-business [14]. Component marketplaces are now appearing, bringing our vision of marketplace-enabled software procurement closer to reality. They, however, tend to be organised along the lines of supply push rather than demand pull. Even more significant difference from our approach is that the assembly and integration (binding) of marketplace-procured components are still very much part of the human-performed activity of developing a software product, rather than a part of the automatic process of fulfilling user needs as soon as they are specified.

Current work in *Web services* does bring binding closer to execution, allowing an application or user to find and execute business and information services, such as airline reservations and credit card validations. Web services platforms such as HP's e-Speak [8] and IBM Web Services Toolkit [12] provide some basic mechanisms and standards such as the Universal Description, Discovery, and Integration (UDDI) that can be used for describing, publishing, discovering and invoking business-oriented Web services in a dynamic distributed environment. We have found e-Speak to be a useful platform for building our second prototype as discussed further down. However, work on Web services is technology-focused and fails to consider the interdisciplinary aspects of service provision such as marketplace organization and trading contracts.

3.4. Novel aspects of our approach

The aim of our research is to develop the technology which will enable *ultra-late binding,* that is the delay of binding until the execution of a system. This will enable consumers to obtain the most appropriate and up-to-date combination of services required at any point in time.

Our approach to service-based software provision seeks to change the nature of software from product to service provision. 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, license details etc).

Such ultra-late binding, however, 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 where they 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 such 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. We have established the

d 6 t a i l s e t c) . (i

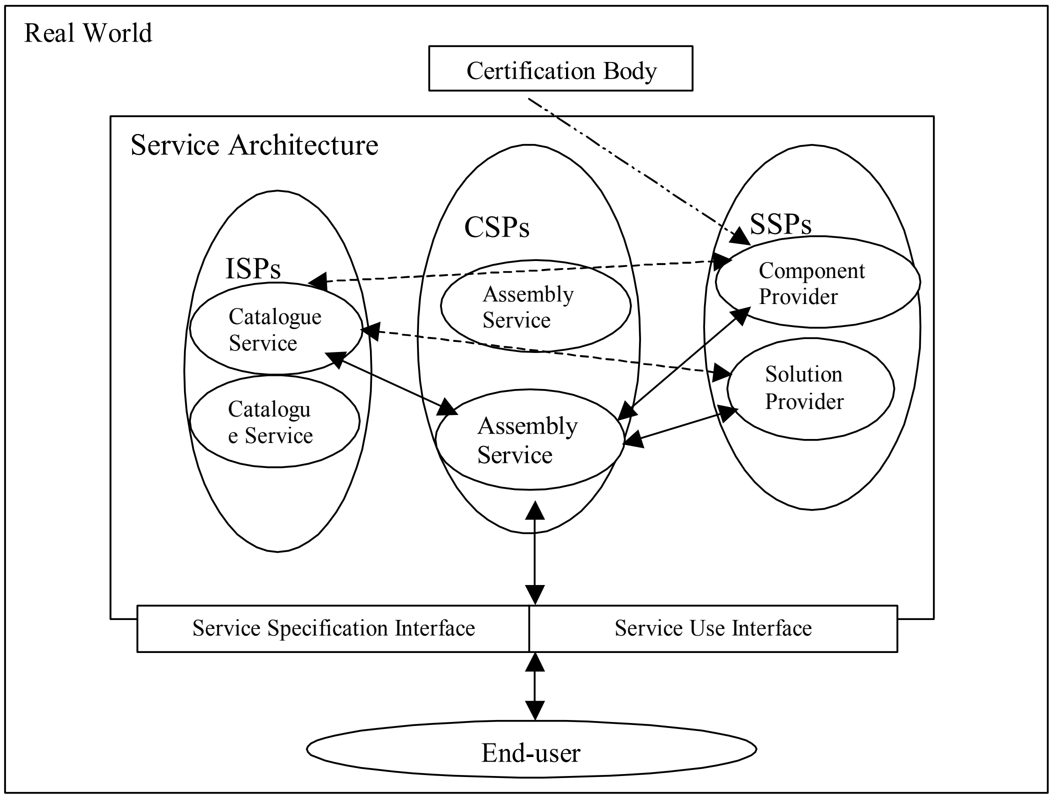


Figure 1: Service architecture

of non-functional attributes with candidate services. Note that the service composition (the design activity) is *not* undertaken by the client or user, but the templates are supplied by SSPs in the marketplace.

It can be seen that this architectural model offers a dynamic composition of services at the instant of need. 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*; where the market has been unable to provide the needs of a purchaser.

It is important to distinguish binding and service composition. The *design* of a 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 sub-services just before the super-service is executed. The application code is replaced by recursive sub-service invocation.

4. Service Implementation – Prototypes and Results

4.1. Aims of the prototype implementation

This section describes the objectives of the two experimental systems (referred to as prototypes 1 and 2), the rationale for using the platforms, the results obtained from the implementations, and the conclusions drawn by bringing together the results of both experiments.

The general aim of the prototypes was to test ideas about the following:

- dynamically bound services at run-time within the flexible software service architecture;
- service binding with limited negotiation;
- service discovery.

To guide the development of our prototype series, we have mapped some of the problems of service-based software delivery into an established transaction model [10]. This model characterises a transaction between buyer and seller and provides the four process phases shown in Table 1. The activities identified within the phases are drawn both from the model and our own work.

Phase	Activities	Prototype no.
Information	Service Description Service Discovery Request Construction	2
Negotiation	Negotiate Evaluate	1
Settlement	Service Invocation Monitoring Claim & Redress	1, 2
After-sales	Evaluate for future	

Table 1: Transaction model for software services

Our first experimental system had the aim of demonstrating the capability of service binding and limited service negotiation [9]. The objectives of the second prototype were to investigate two aspects of the above theoretical model: service discovery, and service binding (see Table 2).

Prototype	Aim	Infrastructure
1. Calculation	Service binding & negotiation	PHP, MySQL and HTML
2. Print service	Discovery & binding	e-Speak

Table 2: Prototype Aims and Infrastructures

4.2. Prototype applications

4.2.1. A calculation service. The first prototype was designed to supply a basic calculation service *to an end-user. The particular service selected was the problem* of cubing a number. Note that due to the service nature of the architecture, we aim to supply the *service* of cubing, rather than the *product* of a calculator with that function in it. This apparently simple application was chosen as it highlights many pertinent issues yet the domain is understood by all.

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

The e-Speak approach allows a single registration and discovery mechanism for both composite and atomic 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 composition is returned from the discovery process, the DGS interprets it to invoke sub-services.

4.3. Experimental infrastructures

4.3.1. Prototype 1: Calculation. This prototype is implemented using an HTML interface in a Web browser. PHP scripts are used to perform negotiation and service composition by opening URLs to subsidiary scripts. Each script contains generic functionality, loading its "personality" from a MySQL database as it starts. This allows a single script to be used to represent many service providers. End-user and service provider profiles are stored on the database, which also simulates a simple service discovery environment.

4.3.2. Prototype 2: Printing. We used e-Speak [8] for building this prototype. It offers a comprehensive infrastructure for distributed service discovery, mediation and binding for Internet based applications. e-Speak has the following advantages as an experimental framework:

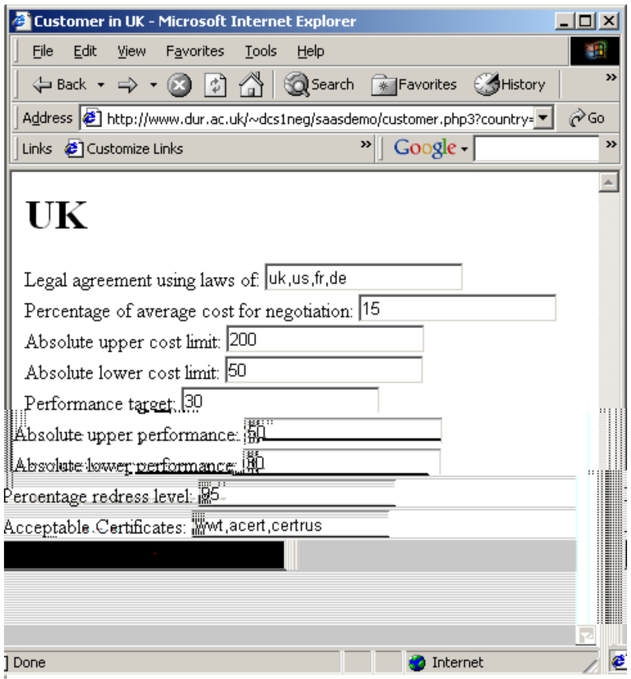
- A basic name-matching service discovery environment, with an exception mechanism if no service can be found.
- Issues of distribution and location are handled through *virtualisation*.
- It is based on widely used systems such as Java and XML.

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.

4.4. The prototype implementations

4.4.1. Calculator prototype. Three main types of entities are involved in service delivery in the prototype: the end-user, an interface, and service providers. The arrows on Figure 1 show the interactions and relationships between them. The interface (in this case, a Web browser) allows the end-user to (a) specify their needs as shown on Figure 2, and then (b) to interact with the delivered software. It is expected that the interface will be light-weight and perhaps supplied free in a similar manner to today's Web browsers.


Customer in UK - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.dur.ac.uk/~dcs1neg/saasdemo/customer.php3?country=
Go
Links Customize Links
Google
Done
Internet

UK

Legal agreement using laws of: uk,us,fr,de
Percentage of average cost for negotiation: 15
Absolute upper cost limit: 200
Absolute lower cost limit: 50
Performance target: 30
Absolute upper performance: 50
Absolute lower performance: 10
Percentage redress level: 95
Acceptable Certificates: wt,acert,certrus

Figure 2: Specifying requirements for the calculator prototype service

Service from the end-user's point of view is provided using the following basic model:

1) The end-user requests a software service.
2) The end-user selects a service domain (e.g. calculation).
3) The end-user selects a service within the domain (e.g. cube).
4) The end-user enters the number they want to cube.
5) The end-user receives the result.

Apart from the notion of requesting the service of cube rather than the product of calculator, it can be seen that the process of cubing is similar to selecting the function from a menu in a software product. However, the hidden activity for service provision is considerable.

Each provision of service is governed by a simple *contract*. This contains the terms agreed by the service provider and service consumer for the supply of the service. The specific elements of a contract are not prescribed in terms of the general architecture; providers and consumers may add any term they wish to the negotiation. However, for the prototype, three terms are required:

1) The **law** under which the contract is made.
2) Minimum **performance** (represented in the prototype by a single integer).
3) **Cost** (represented by a single integer).

In order to negotiate a contract, both end-users and service providers must define *profiles* that contain acceptable values for contract terms. The profiles also contain *policies* to govern how these values may be negotiated.

The profiles used in the first demonstrator are extremely simple. End-user profiles contain acceptable legal systems for contracts, the minimum service performance required, the maximum acceptable cost, and the percentage of average market cost within which negotiation is possible. Service provider profiles contain acceptable legal systems for contracts, guaranteed performance levels, and the cost of providing the service. Negotiation in the prototype thus becomes a process of ensuring that both parties can agree a legal system and that the service performance meets the minimum required by the end-user. If successful, service providers are picked on the basis of lowest cost. Acceptable costs are determined by taking the mean of all service costs on the network for the service in question and ensuring that the cost of the service offered is less than the mean plus the percentage specified in the end-user profile. It must also be less than the absolute maximum cost.

To avoid the overhead of negotiation for basic, common services such as catalogues, it is assumed that external bodies will provide "certificates" which represent fixed cost, fixed performance contracts that do not require negotiation. Both end-user and service provider profiles contain acceptable certificates.

Negotiations take place within the following simplified procedure for service provision:

- A contractor (assembly) service is selected using negotiation, and the requirements passed to the service.
- The selected contractor service obtains available solution domains from the catalogue services, and the end-user selects the calculation service domain.
- The contractor then retrieves services available within the calculation domain. Again, a list is presented and the end-user selects cube.
- Now the contractor negotiates the supply of a cube solution description from a software service provider. The solution tells the contractor service which other services are required to perform cubing and how to compose them.
- The contractor then finds potential sub-services, negotiates contracts with them, and composes them following the template.
- The user uses the cube service.
- Having completed the service provision, the contractor disengages all the sub-services.

4.4.2. Printing service prototype. 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.

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.

4.5. Results

The calculation prototype has demonstrated that the ideas behind our approach and the primitives of our architecture are feasible and viable. A very simple application domain example has been found sufficiently rich to enable demonstration of many of the basic ideas, such as service negotiation, dynamic composition and subsequent disengagement. The prototype has also been extended with little effort to supply an "addition" service. The implementation has shown that a service architecture is not prescriptive, by allowing, with very few exceptions, the use of different negotiation, discovery and description methods. In summary, the experimental work on the calculator prototype has provided two areas of evidence to support our aim of ultra rapid evolution:

- Very late binding, and subsequent disengagement can be achieved for both functional and non-functional service attributes, given suitable discovery, description and negotiation representations.
- It is feasible to build a service architecture which is not committed to particular description notations or negotiation mechanisms.

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

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 component (the DGS class). We also need to extend this to support negotiation on non-functional aspects, again 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, which plays the role of a contractor service provider in our architecture. This is necessary because e-Speak does not have the concept of composition templates or patterns. These have to implemented externally and interpreted by the broker.

5. Future Research Issues

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

Requirements for software need to be represented in such a way that an appropriate service can be *discovered* on the network. 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.

6. Conclusions

We have presented a radical approach to achieving ultra rapid evolution of software by moving from the concept of a software product to a mechanism of service delivery at the point of need. Service provision involves recursive decomposition to sub-services, and atomic services are still produced using traditional software development techniques. However, providing software as a service postulates a completely different software marketplace, which is demand-led, not supply-led. Using demand-led marketplaces ensures availability of up-to-date services, which are bound together at the point of customer need, just before execution, and disengaged afterwards, so a service can be replaced by an improved one when needed. The software may thus be continually adapted to meet user requirements.

Two prototype implementations have been built using two different technologies: scripting and e-Speak. We have found the scripting approach to be very flexible, but with significant performance limitations, while in e-Speak, the innovative aspects of our architecture have to be imprpec(ng apprn)12sxfwsvice deeak, thst2(prpec(n(nts.)]TJETEMC/P /M4ID 2 BDCBT/TT1 1 Tf06.001 Tc4180987 Tw 10.02 0 0 1