A Service Model for Component-Based Development

John Hutchinson, Gerald Kotonya, Ian Sommerville and Stephen Hall
Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{hutchinj, gerald, is, s.hall}@comp.lancs,ac.uk

Abstract

The promise of component-based development – the development of large-scale applications from off-the-shelf software components – may remain elusive unless we complement the facilitating technologies with processes that are designed to take account – and advantage – of components right from the start. Blackbox components present a number of serious challenges whose impact may offset the potential benefits of their use unless they can be addressed successfully. We describe a process that makes use of a service-model to map user-requirements to components, and which offers support for some of the challenges posed: component-oriented requirements elicitation, negotiation, design, verification and change management. Our approach also supports a hybrid component/service-based development where off-the-shelf components and services can co-exist in the same system.

1. Introduction

There are a number of approaches to component-based software engineering (CBSE). Although component-oriented approaches include those supporting, for example, in-house product-line development with an emphasis on grey-, or even white-, box reuse, we are more interested in the potential offered by *standards-based* blackbox off-the-shelf components. The separation of component developers and components users should bring many benefits: efficiency, productivity and commercially available expertise, for example. However, it will also entail a blackbox approach to development and that poses some significant challenges.

Most software engineering thus far has been focussed on processes for identifying users requirements and then developing code to meet those requirements. Ideally, those processes should be reliable, repeatable and predictable, and they should include the means to manage the many challenges inherent in the task. Two central assumptions, though, are that within a certain limit, the right system can be developed according to the user's requirements, and that a different set of requirements will result in a different system. These assumptions are at odds with a blackbox component view of system development because the concepts underlying each central challenge are completely different. On the one hand, we attempt to follow a potentially problem-packed development path, making decisions along the way that will affect the code produced. On the other, we assume that the "code" already exists.

Such issues are at the heart of any off-the-shelf approach to application development:

- *Poor documentation.* A software component's documentation will be according to the choices of the component developer, not the component user. We must assume that the focus will be on functional properties with little indication of how components might behave in different contexts.
- *Limited adaptability.* We assume that blackbox software components are generally not tailorable or "plug and play". We expect that a significant part of the effort of using these components will be in building adaptors and "glue" in order to evolve applications and to tailor components to new situations.
- *Vulnerability risk.* The design assumptions of a software component are largely unknown to the application builder. In a situation where several complex functions are replaced by a single software component this may have serious implications for exception handling, system resources and critical quality attributes.
- *Component volatility.* Software components may or may not be subject to frequent upgrades. But what is certain is that it will be the needs of the component developer/vendor that dictate the upgrade scope and timetable. When this results in disparity in customer-vendor evolution cycles, the result will be unplanned upgrades being forced on customers and associated impact analysis challenges.
- *Component mismatch.* The flexibility of

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

heterogeneous environments means that software components are likely to suffer from mismatches due to different data models, functional mismatches or resource conflicts.

A further challenge is that a purely "component" based solution might not be appropriate – for example, one system might require some custom development, another, the integration of a legacy system and another, the integration of a particular web-service – and yet that might not become obvious until some way into the development process. Our response to these challenges is to propose a process that is component-aware right from the start. Within this context, we then use a service model for mapping between user requirements and available components, which introduces the flexibility, robustness and traceability to component-based designs required to accommodate the different demands, as well as contributing to the support for future management tasks, particularly change impact analysis.

2. Background

Component and service-oriented development pose many challenges to organisations intending to adopt them:

- Traditional software development approaches are unsuitable for developing component and service-oriented systems [1]:
  - In the waterfall model requirements are identified at an earlier stage and components chosen at a later stage increasing the likelihood of there being no components offering the required features.
  - Evolutionary development assumes that additional features can be added if required. This is not the case with blackbox components.
  - There is a general lack of analysis tools that support *development with reuse* (particularly black-box development).
- There is a general lack of methods for mapping functionality to components/services and for grouping them into logical domains [13].

These problems underpin the need for software engineering processes and methods that:

- Can balance aspects of system requirements, business and project concerns, with the assumptions and capabilities embodied in off-the-shelf software components. Current methods for *development with reuse* have focused on specific development activities (e.g. component selection and component specification) rather than the process [4, 14, 12, 3, 8]. This has obscured the correspondence between the different activities and made it difficult to achieve a balanced solution.
- Can support hybrid component/service-oriented development to leverage their different design strengths. There is a general lack of software engineering approaches that support this kind of hybrid development.

3. Development process

Figure 1 shows our 4-phase development process: Component-oriented Software Engineering (COMPOSE) process. The negotiation phase provides a framework for reviewing aspects of system development for trading-off competing attributes. The planning phase sets out a justification, objectives, strategies and tactics for the development project.

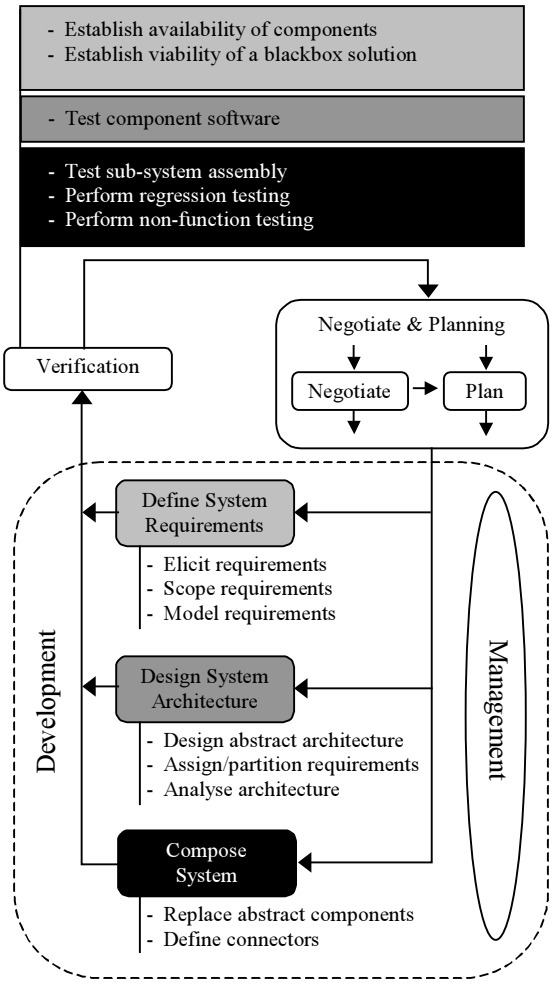

Figure 1. Component-based development process.

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

The development phase implements the agenda set out in the planning phase. The first step in application development is the definition of system requirements. The requirements stage elicits, prioritises and models the system requirements as a set of services and constraints. The design stage partitions the services and constraints into sub-systems and eventually into components that can be mapped onto component- or service-oriented architectures. Like the requirements stage, the design stage proceeds in tandem with the verification and planning phases, and may iterate to the requirements stage from time to time. The composition stage replaces abstract design components with off-the-shelf "equivalents". Off-the-shelf software components are packaged in many different forms (e.g. function libraries, frameworks and legacy applications). The composition process must devise mechanisms for integrating different components without compromising the system quality.

The verification phase is intended to ensure that there is an acceptable match between the software components and the system being built. This is important because the perception of software quality tends to vary amongst component producers. In addition the blackbox nature of most software components diminishes the scope for correcting errors later in the system development. The verification phase varies in focus and detail across the development cycle. (Matching colours have been used to indicate the correspondence between different development stages and verification activities that apply to them.) At the requirements stage, verification is used to establish the availability of software components and viability of an off-the-shelf solution. At the design stage verification is concerned with ensuring that the design matches the system context (i.e. in terms of non-functional requirements, architectural concerns and business concerns). This may require detailed blackbox testing of the software components and architectural analysis. At the composition stage verification translates to design validation, through component assembly testing and system testing.

4. Service-oriented model

We introduce the concept of services as a means of modelling requirements and component functionality, and thus mapping between requirements and components. This is a logical extension of the modelling and scoping associated with the viewpoint oriented requirements definition process we use, all in a context of component availability. Defining services offers several distinct advantages over other component-based software development processes, which rely on defining components interfaces first and then binding them to services. Explicit identification of viewpoints with services makes it possible to create a framework for deriving, integrating and analysing several important aspects in component-based development:

- It provides a scheme for integrating functional and non-functional requirements. Non-functional requirements define the overall qualities or attributes of the resulting system. Non-functional requirements may arise out of user needs, budgetary considerations, organisation policies, need for inter-operability with other component frameworks or external factors such as legislation. Because they are restrictions on system services, non-functional requirements greatly influence the design solution. Addressing them separately obscures the correspondence between them and the services they constrain. A service is naturally associated with quality, the level of which may vary actor requirements and concerns.
- It provides a powerful mechanism for partitioning the system architecture that takes into account service relationships, invocations, interfaces, evaluation criteria and desired quality. The focus is shifted from component interface specification to partitioning services that provide an acceptable match with architectural considerations and available software capabilities.
- A service may be specified in a variety of notations and at varying levels of abstraction. In the method we propose, services are initially modelled as use cases and subsequently developed using a state transition notation.
- Service specifications provide us with a "pluggable" basis for custom development in cases where available software is inadequate to meet the system requirements.
- Service models can be used as a basis for developing test-cases which can later be used to verify COTS software functionality at design.

Figure 2 illustrates the service-modelling layer. The services/constraints are a separate abstraction of both the requirements and components used in the system and are expected to be meaningful in terms of the business function they are fulfilling, modelled according to the availability of suitable components.

How a service is actually expressed very much depends on the circumstances (see Figure 3). In some cases, natural language will suffice, in others, use cases, state

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

models and sequence diagrams. We believe that the degree and form of modelling and specification should be appropriate to the context.

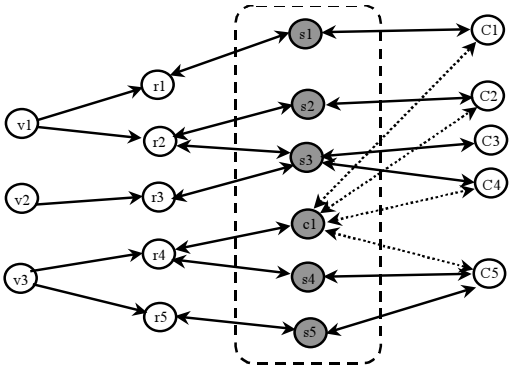

Figure 2. A service model for component-based development. Viewpoints (v) give rise to requirements (r) which are mapped onto components (C) using services (s) and constraints (c).

We also use CADL [10], which is a service-oriented Component Architecture Description Language, to define services and components. (A detailed description of CADL is beyond the scope of this paper.) This allows architectures to be checked for consstency.

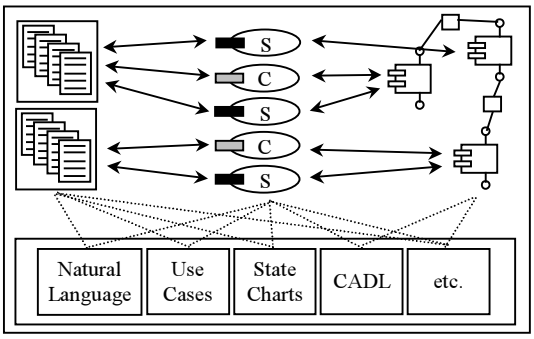

Figure 3. Requirements on the left are mapped to components on the right (represented by the graphical notation for CADL components and connectors) using services and constraints in the middle. Modelling is carried out by the developer using a suitable method–defined by the requirements of the system or the expertise and/or preferences of the developer.

5. Requirements definition and design

5.1. Requirements

The principal challenge in requirements engineering for component-based systems is to develop requirements models and methods that allow us to make the best use of the available off-the-shelf technology by balancing aspects of requirements with the architectural assumptions embodied in available software and the capabilities of that software. However, few approaches address themselves to this problem. Current approaches are based on procurement models in which the requirements process is driven by the availability of existing software [14, 16]. In one such a model, Vidger proposes that system requirements should be defined according to what is available in the marketplace, and that organisations should be flexible enough to accept off-the-shelf solutions when they are proposed.

Vigder [16] notes that overly specific requirements preclude the use of COTS components and should be avoided. This is a reasonable assumption, however, most systems have requirements that are unavoidably 'specific', for example, critical systems. Critical systems are likely to have stringent, often competing quality requirements (e.g. performance, efficiency, safety, reliability etc.). A singular focus on COTS component selection also ignores dependencies and interactions between requirements. It reduces the scope for requirements negotiation, makes it difficult to address system level concerns, and to assess the impact on the system of new components.

Our proposed solution combines the process of requirements definition with software verification and negotiation to provide an intuitive scheme for mapping requirements onto services and then partitioning those in component-based architectures. In the early stages of the requirements definition the verification is used is as a coarse filter for establishing the availability of suitable products. As requirements are modelled and formalised, verification may be used to establish how well selected components match the desired system functionality. At the requirements stage verification may also used to provide project managers with an indication of the viability of a component-based solution. For certain types of system (and requirements), an off-the-shelf solution may not be adequate or even appropriate. The negotiation process is intended to determine an acceptable trade-off between business goals, project constraints, system requirements and software availability.

Our requirements process, COREx (Component-Oriented Requirements Expression) [11] is based on the notion of viewpoints [9]. A VP approach is ideal because it creates a framework for viewing the system from different perspectives and, thus, makes explicit the need to rationalise contradictory requirements and negotiate between competing requirements. COREx uses the following abstract viewpoints to elicit requirements:

- *Actor viewpoints* are analogous to clients in a client-server system. The proposed system (or required component) delivers services (functional requirements) to viewpoints, which may impose

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

specific constraints (non-functional requirements) on them. There are two main types of Actor viewpoints:

- *Operator viewpoints* map onto classes of users who interact with the proposed system.
- *Component viewpoints* correspond to software components and hardware devices that interface with the proposed system.

- *Stakeholder viewpoints* vary radically from organisational viewpoints to external certification bodies. Stakeholders are entities that do not interact directly with the intended system but which may express an interest in the system requirements. These viewpoints often generate requirements that affect the way the system is developed.

A viewpoint template has the following structure:

- **Viewpoint id**: <A unique viewpoint identifier>
- **Type**: <Viewpoint type (e.g. operator, component, stakeholder, etc)>
- **Attribute**: <An optional set of data attributes for the Actor viewpoint>
- **Role**: <Viewpoint's role in the system>
- **Requirements**: <Set of requirements generated by the viewpoint>
- **History**: <Development history>

A requirement can be considered at different levels of abstraction to allow for scoping and ease of understanding. At the user level a requirement has the following structure:

- **Requirement id**: <Requirement identifier>
- **Rationale**: <Justification for requirement>
- **Description**: <Natural language definition> | <Service description> | <Other description>

A viewpoint requirement is modelled as a system service or constraint. Conceptually a service interacts with parts of the system through a number of interfaces. As we noted above, services can be modelled at different levels. At the viewpoint level, a service interface defines the boundary between an actor and a use case (system functionality required by the actor). At the system level a service interface translates to a set of interaction endpoints or ports. At this level a service is viewed as a mechanism for modelling a meaningful group of functionality, which will then map onto a set of related ports.

All classes of viewpoint may generate non-functional requirements. Non-functional requirements vary from constraints on viewpoint services to external requirements related to legal or economic constraints. They also tend to conflict and interact with other system requirements [9]. Our requirements approach allows the engineer to document non-functional requirements, constraints, by associating them with the services they affect and specifying the tests that should be carried out on COTS components to validate them.

A service description comprises the following elements:

- ***Invocation*** <Set of parameters required by a service and how the parameter values are used by service. Parameters correspond to attributes in the COREx service model >
- ***Behaviour*** <Specification of the system behaviour that results from the invocation of the service. This can be described at different levels of abstraction to aid understanding and to help with component selection>
- ***Constraints*** <Description of constraints on service>
- ***Evaluation criterion*** <Tests that should be carried out to evaluate a component's conformance with service>

What the service actually specifies will be dependent on a process of negotiation between competing requirements and viewpoints and what is available.

5.2. Design.

Component-based system design defines how the components that make up the system interact to deliver the required functionality, and how the system reasons about the appropriateness of particular components and services. Different software components may result in different system architectures to achieve desired quality attributes because of the different constraints they impose on the system design [5]. A design trade-off may, for example, require that critical components interact only through "validation" components to ensure that the desired level of system security or safety is maintained, even if this means a loss in system performance.

The design process starts with the partitioning of the system requirements (services and constraints) into logical "components" or "sub-systems". These sub-systems are dictated by the requirements of the system, either explicitly or implicitly. Figure 4 shows how a top-down process may be used to partition the system by clustering services to reflect desired quality attributes and/or component provision. An outcome of this approach is that component boundaries, and therefore interfaces between components, become a consequence of matching services to service providers, rather than driving it.

We use CADL, our service-oriented architecture description language, to describe system architectures.

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

This means that syntactic validation can be used to check that architectures are consistent.

A configuration of components chosen to deliver a given set of services may be entirely appropriate in one situation and not another. This flexibility is inherent to the service modelling approach. Figure 4, coincidentally, also illustrates how the approach supports a move from one configuration to another. In one context, perhaps a moment in time, service provision might be met by an existing component or legacy system, when at another, a different component-based configuration might be introduced. This might later be migrated to a Web service-based solution whilst using the same underlying service model.

We further describe the role that the service model can play in management tasks below, specifically in the case of change impact analysis.

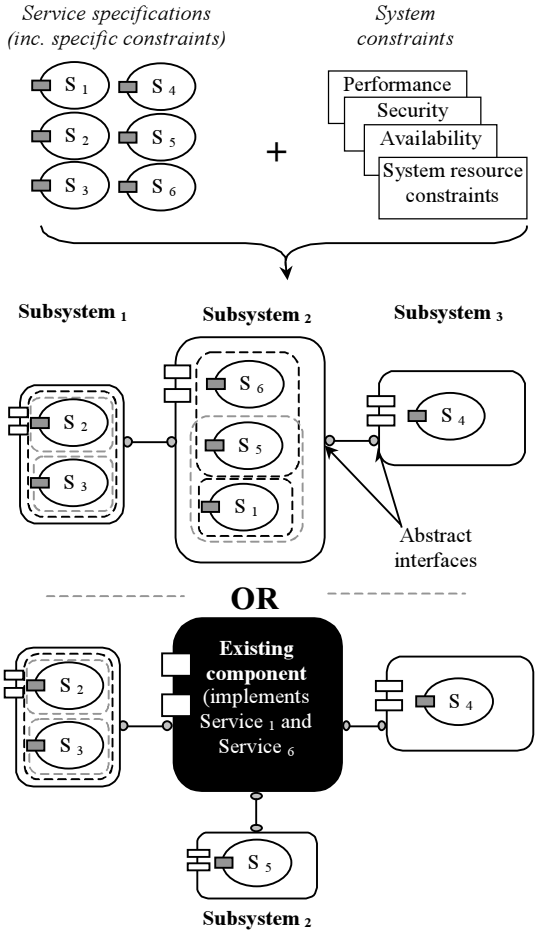

Figure 4. System design.

6. Verification

The objective of the verification process is to ensure that there is an acceptable match between the required system and software components used to build it. In the early stages of requirements definition, verification is used as a filter for establishing the availability of candidate components and services, and to provide the project manager a rough indication of the viability of a component or service-based solution. In design verification is used to establish how well selected components and services match the desired system functionality and how compatible the component interfaces are with the designed sub-systems (see Figure 1). Limitations of commercial software components and architectural considerations may mean that the design has to be modified (i.e. services reassigned to different components) or requirements modified. In extreme cases of incompatibility, parts of the architecture can be viewed as "place-holders" for custom development.

The exact nature of the verification tasks at each stage will depend on the circumstances. Whether checking availability, the viability of an off-the-shelf solution or testing existing components, the service model produced represents the outcome of the verification process. Thus, services express how to meet requirements in the context of component availability.

7. Support for change impact analysis

Our approach to impact analysis is multi-stranded, and includes both CADL [10] and a component-oriented changeability model [7, 2]. Many of the details are beyond the scope of this paper.

Perhaps the most significant element of out approach to change impact analysis depends on modelling the development process, and to do this, we use the notion that an artefact is produced at every stage of the development process, even if the artefact reduces to some known decision point. Since our requirements process is viewpoint driven, we can identify a number of discrete entities in the process. These are VPs, requirements, service and constraint models, architectural elements (components and connectors) and components. In a traditional development cycle, these entities would be associated with a fairly linear process from VPs to components. However, as we have already explained, we believe that component availability defines a context in which the requirements engineering phase of the process proceeds, and service and constraints are the means by which requirements map onto components, and vice versa.

A further consideration is that we are concerned with

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

off-the-shelf components. Because it is component developers and component vendors, not the system developers, who will determine the form and availability of components, we must be able to support the analysis of change that is imposed on the components used in the system regardless of the requirements of the system.

The result of these considerations is that our process model must view services and constraints as being as dependent on components as they are on requirements. Figure 2 can now be seen to illustrate a simple process model, and the mostly undirected nature of the graph can now be interpreted as a result of the two-way dependency of services and constraints. (We expect that in most cases, the arc from VPs to requirements can be directed, as illustrated.)

The nodes representing VPs, requirements and components can all be considered as the targets of possible change. Notice that services and constraints are not considered the targets of possible change because they represent the developer's mapping process between requirements and available components. The potential impact of a change will be all items represented by nodes that are reachable from the node representing the item to be changed. Along with showing us how to identify items which may be impacted by a change to the system, this suggests that in much the same way that there are important design principles to bear in mind in traditional software engineering, there are similar principles in any method proposed for CBSE. The most obvious principle is that there should be a minimum of coupling between requirements and services and constraints and similarly between services and constraints and components. An example shown in Figure 2 is the crosscutting constraint, c1, which affects most of the components of the system. Any change that adversely affects c1 is likely to have a significant impact on the system. Figure 5 illustrates the possible impact resulting from an imposed change to a component used in the system. In this example, c1 might represent a strict platform or OS version constraint.

When we refer to the impact of a change, we are interested in identifying parts of the system that may need attention following a change or proposed change. The process model cannot indicate what the attention required might be – indeed, because of the inequality between requirements, services/constraints and components, there will be many cases where apparently even quite significant changes will have little impact on the system. However, taken as a whole, the process model captures significant "knowledge" about the process by which the system was developed. The entities represented by reachable nodes following a change, or proposed change, can be examined by the developer to determine the true extent of the impact.

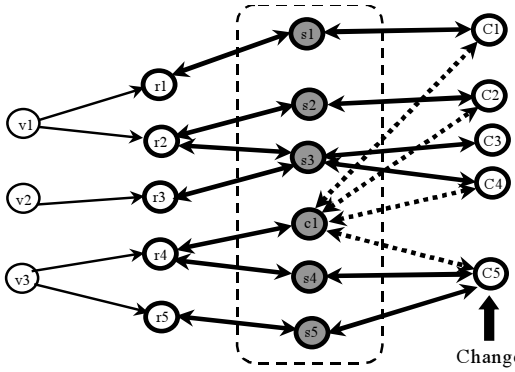

Figure 5. The potential impact of an enforced component change, identified using our process model.

8. Hybrid development

In describing the approach we have presented here, we have generally been able to avoid the arguments about what a component actually is. A primary reason for this is that our notion of a component is more concerned with its off-the-shelf status than a given set of technical characteristics. Of course, this gives us some alignment with the views held by others (e.g. [15, 6]). It also makes our proposals much more technologically neutral.

The growth in service-oriented computing, and specifically Web services, looks likely to have a huge impact on the process of developing software systems. However, many of the challenges that such approaches present are similar to those presented by off-the-shelf components and result from the different production cycles for the service, on the one hand, and the application that makes use of that service, on the other. The proposals we have presented address this issue directly because our service and constraint modelling is carried out in the context of the availability of provision for the services in question. From this perspective, it is of no great interest whether the service is provided by a component, or a web service.

A further benefit of our approach is that it takes account of architectural concerns right from the start. Architectural considerations are important to service-based approaches for a number of reasons including separation of concerns and future maintenance. Moreover, the use of off-the-shelf solutions imposes significant architectural decisions regardless of the user requirements for the system being developed, and these must simply be managed. Our approach offers a way to do that, in such a

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

way that the effort expended can be used to carry out change impact analysis.

Of perhaps greater potential than using the processes we have presented for developing systems using existing services is the notion of hybrid development. The advantages and disadvantages of purely component-oriented, or purely service-oriented, approaches make them appropriate only for certain types of systems (e.g. the communication overhead makes web services unsuitable for user interaction). Independently, this will limit the application of these technologies, but hybrid development that exploits both 3rd-party components and 3rd-party service provision, along with their associated purchasing models is an ideal solution.

Some of the service-oriented challenges (e.g. trust, reliability, etc.) that differ somewhat from those posed by component may be addressed as far as practicable during the verification stages of our process and/or mitigated by architectural decisions at implementations time. For example, using service models and architectural subsystems to model management services or proxies.

9. Conclusion

The potential benefits of a move towards developing systems from off-the-shelf components are huge, but they come at a price. Some of the challenges are technological, but others fundamentally question the approaches we adopt when developing applications. We have presented our proposals for processes that begin to address some of the biggest challenges posed by off-the-shelf components. These ensure that requirements elicitation and design take place within a context of component availability. Verification is used throughout to manage and inform negotiation and trade-off. The central mechanism for achieving the mapping between requirements and components is a service model. This model allows us to meet requirements in the most appropriate way using the components available to us. Moreover, it provides a model of the development process that can be used to effect change impact analysis.

The advent of widely available 3rd-party services will change the way software is developed. We believe that future systems will combine both services and components and therefore the processes used will have to address the challenges posed by both.

Finally, this work is still being developed. However, the method presented here has been partially implemented in the tools associated with the ECO-ADM project, where the service model is used to characterise requirements and components. With appropriate tool support, our project partners have successfully used our approach in several non-trivial systems with encouraging results.

Acknowledgements

The work described in this paper was partly funded by the European Union projects: ECO-ADM (IST programme 20771) and CBSEnet (IST programme 35485). We are grateful to our project partners for their contributions.

References

[1] B. Boehm and C. Abts, "Integration: Plug and Pray", *IEEE Computer* 32(1), 135-138, 1999.

[2] M.A. Chaumun, H. Kabaili, R.K. Keller and F.A. Lustman. "Change Impact Model for Changeability Assessment in Object-Oriented Software Systems". In Proceedings of the Third Euromicro Working Conference on Software Maintenance and Reengineering, pages 130-138, Amsterdam, The Netherlands, March 1999.

[3] I. Crnkovic, B. Hnich, T. Jonsson and Z. Kiziltan, "Specification, Implementation and Deployment of Components: Clarifying Common Terminology and Exploring Component-based Relationships", Communications of the ACM, 45(10), 35-40, 2002.

[4] D.F. D'Souza and A.C. Wills, "Objects, Components, and Frameworks With UML", The Catalysis Approach. Addison-Wesley, 1998.

[5] G. T. Heineman, "A model for designing adaptable software components", Proc 22nd Annual International Computer Software and Applications Conference, pg.121-127, Vienna, Austria, 1998.

[6] G. T. Heineman and W.T. Councill, "Component-Based Software Engineering – Putting the pieces together", Addison-Wesley, 2001.

[7] J. Hutchinson, G. Kotonya, B. Bloin and P. Sawyer, "Understanding the Impact of Change in COTS-Based Systems", Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, USA, 23-26 June 2003.

[8] S.D. Kim, "Lessons Learned From A Nationwide CBD Promotion Project", Communications of the ACM, Vol. 45(10), 83-87, 2002.

[9] G. Kotonya, "Experience With Viewpoint-Based Specification." Requirements engineering, 4(3), 115-133, 1999.

[10] G. Kotonya, W. Onyino, J. Hutchinson and P. Sawyer. "Component Architecture Description Language (CADL)", Technical Report, CSEG/57/2001, Computing Department, Lancaster University, 2001.

[11] G. Kotonya, J. Hutchinson, "Viewpoints for Specifying Component-Based Systems", to appear in Proceedings of the International Symposium on Component-based System (CBSE7), 2004.

[12] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework For Software Architecture Description Languages", Trans. IEEE Software Eng., 26 (1), 70-93, 2000.

[13] E.G. Nadhan, "Service Oriented Architecture Implementation Challenges", EDS.com, 2003. Available at: http://www.eds.com/thought/thought_leadership_so_architecture.pdf.

[14] C. Ncube and N. Maiden. "PORE: Procurement-oriented requirements engineering method for the component-based systems engineering paradigm", in Proceedings 2nd IEEE International Workshop on Component-Based Software Engineering, Los Angeles, California, USA, pp 1-12, May 1999.

[15] C. Szyperski, "Component Software: beyond object-oriented programming", 2nd Edition, 2002, Addison Wesley Longman.

[16] M. Vigder, M. Gentleman, and J. Dean. "COTS Software Integration: State of the Art", Institute for Information Technology, National Research Council, Canada, 1996.