

The Four Major Challenges of Engineering Adaptive Software Architectures

Jun Han and Alan Colman

Faculty of ICT, Swinburne University of Technology

John Street, Hawthorn, 3122, Australia

{jhan, acolman}@ict.swin.edu.au

Abstract

Building an adaptive software system that can cope with changing requirements and changing environments presents four major challenges. These are (1) to receive, represent and reason about changing requirements and goals; (2) to cope with volatility of the computational context in which it executes; (3) to work with other systems that are heterogeneous and distributed; and (4) to do all of the above without becoming too complex to develop and maintain. These challenges can be addressed by architectures that focus on the adaptivity of relationships rather than entities. Role-Oriented Adaptive Design (ROAD) is presented as such an architecture that addresses the challenges.

1. Introduction

Software systems become inexorably more open, distributed, pervasive, mobile and connected. This accelerating trend is being driven by new technologies that take advantage of the rapidly increasing power and falling cost of hardware and networking. Consequently, the environments in which software applications operate are becoming more diverse and dynamic. Furthermore, the applications may also have to adapt to changing requirements related to the goals of various users. In order for applications to be readily evolvable and/or highly available, they will need to be able to adapt to such changing environments and requirements.

Work on making software systems more adaptive can be categorised into two broad classes of research. The first includes those approaches that attempt to solve particular technical problems that arise when we try to achieve desired behaviour from unstable environments (e.g. server load balancing and resource allocation).

A second broad research area into adaptive software attempts to define software engineering principles, methodologies and frameworks that support the design of software applications that exhibit various types of adaptive behaviour. Such adaptive behaviour can be in response either to changing requirements of the

application, or to the changing provision of service by the constituent components or underlying infrastructure. One fundamental approach to building runtime adaptive systems is to construct them of loosely coupled entities. The indirection between entities provides the degrees-of-freedom that are a prerequisite for a system to adapt to changing requirements and environments. These entities in the software composition are dynamically coordinated and reconfigured to meet environmental demands or changing goals. This architectural view of software is the perspective adopted in this paper. That is, software is viewed as being made up of computational entities that are combined with connectors. These connections create a structure that enables interactions between those entities. In particular we focus on how to achieve *adaptive* architectures: generic designs that define how the software structure and behaviour of loosely coupled entities can be managed at runtime. These architectures are able to represent a dynamic software structure, and manage changes to the configuration of that structure at runtime. Typically, such architectures have separate management entities that can sense changes in the environment, and reconfigure the application in response to those changes.

2. The Four Challenges

There have been a number of recent attempts to develop adaptive software architectures (e.g. [1,3,10,13]). However, none of these architectures have yet moved beyond a research context, or even achieved wide-spread acceptance within the research community. To be widely accepted in the current heterogeneous technological contexts, any candidate adaptive architecture will need to address a number of software engineering challenges. Such an architecture must facilitate the development of software systems that can: (1) receive, represent and reason about changing requirements and goals; (2) cope with volatility of the computational context in which it executes; (3) work with various technological contexts by interoperating with other systems that are heterogeneous and

distributed; and (4) do all of the above without becoming too complex to develop and maintain.

We will now examine why it is necessary to overcome each of these challenges as a prerequisite to developing a broadly applicable adaptive architecture.

2.1. Representing requirements at runtime

If a software system is going to adapt itself to changing requirements, it needs to have a representation of those requirements, so that it can have some sense of them, maintain or achieve them, and detect the system behaviour's deviations from them.

In conventional software engineering, the requirements of the software are kept separate from the design and implementation of the system. This separation of the "what" and the "how" is seen as necessary in order to avoid 'implementation bias' that prematurely constrains the solution space. However, a shortcoming of this separation is that the requirements and their specifications are not apparent from the software itself. In order for a software system to be readily evolvable, the specification of requirements needs to be explicitly expressed in the software, but this specification also needs to be kept separate from the functional entities that fulfil/operationalize those requirements so that the existence of the specification is not dependent on any particular implementation.

There have been various attempts to make explicit at least some requirements of a software entity in the software itself. For example, the definition of interfaces as separate entities in languages like Java, CORBA IDL or WSDL can be viewed as an attempt to maintain, within the code, a specification of the type of interaction that can occur with the class or component, while keeping the implementation separate from that specification. Similarly, Design-By-Contract (DBC) [11] extends the specification in the interface by setting preconditions, post-conditions and invariants on interaction with the class. However, these approaches have a number of major limitations in terms of the development of adaptable software.

The first limitation is that requirements for an entity are fixed at design-time. At design-time, a range of *anticipated* adaptations can be incorporated into the design of an entity. This design is then operationalized either by selecting a pre-existing entity(s) that meets those requirements, or by building an entity to that specification. However, the challenge is to have software systems that can cope with unanticipated changes at runtime. For example, in conventional object-oriented and component-based approaches, interfaces define a static *type* to which entities must conform. No mechanisms exist for dynamically updating the requirements expressed in those interfaces, or for operationalizing the system to the new requirements, either by replacing the constituent entities, or having those entities reconfigure themselves.

Secondly, such interface definitions and contracts only specify functional requirements. The interface definitions in Java and the CORBA IDL merely specify the syntax and types of permissible functional interactions. Likewise, although DBC can express assertions related to the behavioural requirements of an entity, it cannot express non-functional requirements (NFRs) related to those interactions. In recent years, suggested enrichment of component and service interface descriptions have included non-functional requirements (e.g. QoS) and the semantics (e.g. [9,17]). However, the heterogeneity of technologies and standards has meant that no interface standard has gained widespread acceptance.

The third limitation of these 'interface-as-specification' approaches is that these requirements only relate to the entity itself, irrespective of the context into which it has been composed. In a composition of entities, this entity-centric specification means that the requirements of the *composition as a whole* are not defined in a single representation, but are scattered through the interface definitions of the constituent entities that make up the whole. In an adaptive architecture, when system-level requirements are decomposed into particular component requirements, we need to preserve those higher-level requirements, and make explicit the mapping between system-level and component-level requirements.

2.2. Accounting for the computational context

Software is typically viewed as a purely logical construct. At runtime, however, software always executes on some physical computers and possibly over networks connecting them, which determine the actual quality of the software's behaviour.

Consequently, the second major challenge for adaptive architectures is to be able to account for the computational and network contexts of the runtime system. This context needs to be taken into account both for individual entities, and for compositions of multiple entities into applications. The runtime qualities of the individual 'situated' entities within a composition need to be aggregated in such a way that the system-level properties can be calculated. (This process of abstraction can be viewed as the inverse to the operationalization of requirements described above).

Computational context can be accounted for in a number of ways. One way is to use 'boilerplate' specifications of a component that present *claims* about particular qualities of the components performance, given a specified computational context. Generally, such specifications are static descriptions of qualities, and assume a stable computational context. A second way to account for computational context is to *measure* some parameter of interest. As mentioned in the Introduction, much research has focused on the development of techniques that measure particular

computational and network qualities; an awareness that enables adaptation to perturbation in the underlying infrastructure. This is typically done by instrumenting an aspect of the infrastructure (e.g. memory, bandwidth, etc.) in order to model the potential impact on performance (e.g. [2,8]), and by defining strategies that mitigate underperformance with respect to the particular quality being monitored.

While it may be possible, if difficult, to model the performance of a single software entity, based either on claims or monitoring, this task is further complicated when we compose these entities into applications. Interdependencies can exist between different qualities (e.g. security, response time); entities may share limited computational and network resources that make modelling of actual runtime performance difficult; and the same qualities may have different representations at different levels of abstraction in a composition.

Furthermore, in open systems, there may be no guarantee of performance of the infrastructure. For example, despite quality claims and best intentions, the response time of a server may depend on the current load placed on that server by other applications. The increasing openness of software systems presents our next challenge to adaptive software architectures.

2.3. Adapting in open systems

Open systems also present other challenges apart from uncertainty of performance. These challenges are how to compose systems from heterogeneous components, and how to manage a distributed architecture.

One of the major impediments to the widespread adoption of adaptive architecture is the heterogeneity of technologies and middleware standards used by components that need to be composed. By using a common middleware, heterogeneous applications are able to communicate and collaborate. Middleware technologies also hide complexity and add value (e.g. reliable messaging, logging, persistence) to these interactions. However, there currently exist no universally accepted middleware standards. For example, in the fast-changing world of Web services, while the basic technology for handling interactions is well established and accepted, standards for handling more complex interactions such as WS-Coordination, WS-Agreement, OWL-S, and so on, are still evolving and are sometimes overlapping or conflicting. In order to build adaptable applications in this changing and uncertain technological context, it would be desirable for applications to be able to make use of heterogeneous standards, while not being bound to any one standard.

A further challenge for adaptive architectures arises from the distributed nature of open systems. As discussed above, adaptive architectures have some form of management that controls the indirection created by the loosely coupled structure. In an open system, the constituent elements are likely to be controlled and

owned by different parties, and consequently be opaque to the manager. Adaptive architectures in open systems need to find ways of managing the components without having access the internal implementation of those components or any fine-grained control over their behaviour. Centralised control in distributed systems can also be problematic. The adaptive architecture may also need to support distributed management, and consequently define a language and means by which the adaptive managers can communicate with each other.

2.4. Managing complexity

Our final challenge for adaptive architectures is to meet all the above challenges, while managing the *complexity* of both the runtime management and the development process of the system.

As mentioned at the beginning of this section, adaptive architectures typically have a ‘manager’ that senses the need for adaptation and changes the application accordingly. While it would be conceivable to have a single manager controlling a large number of interrelated components, organisational theory [12] posits that systems are best decomposed in such a way that the manager of any one composite has a limited span of control. Given their “limited rationality” [16], managers should control only a relatively small set of relationships at a single level of abstraction. Managers should not have to ‘micro-manage’ their components at lower levels of abstraction. Larger adaptive systems will therefore need to have management distributed down through the structure. The challenge for an adaptive architecture is to show how management can be effectively decomposed and distributed while maintaining the coordination of the system as a whole.

A second issue for the runtime management of adaptive systems is the multiplicity of qualities that have to be managed. While in simple systems (e.g. a thermostatically controlled heater) it is possible to identify certain variables of interest that can be sensed, measured and controlled, in software systems there are typically a number of interdependent quality requirements (e.g. reliability, security, throughput, cost, and so on). The manager of the (sub-)system may need to dynamically trade-off these attributes against each other in order to best meet its requirements. The manager of the (sub-)system may also need to be able to translate *between* quality requirements at different levels of abstraction. For example, in a video application a quality requirement may be expressed in terms of resolution and frame-rate. These quality attributes may, in turn, be dependent on lower level of quality abstractions such as bandwidth and latency. The managers of the relationships at these various levels of abstraction will need to be able to translate between these various expressions of quality attributes.

If complexity is to be controlled, it is important that the manager(s) of the systems only monitor those

variables in its environment that are relevant to its performance, and that these ‘qualities of interest’ can be dynamically added and controlled. Adaptive architectures (e.g.[8]) that attempt to take account of various user, computational and network contexts in a generalised way are themselves very complex, and consequently may be impractical. What is needed is an approach that implements a level of adaptivity appropriate to the amount of volatility or uncertainty in the system’s environment. To simplify the development task, adaptive capability should be able to be added incrementally as needed to an application.

Finally, if adaptive architectures are to gain acceptance amongst developers then the concepts in the architectural meta-model should be as simple as possible (and no more). One of the advantages of the object-oriented methodology is that it is based on a few powerful concepts (encapsulation, inheritance, polymorphism, etc.) that can be realised in a relatively simple meta-model (class, object, method, etc.). Likewise, an adaptive architectural model should ideally be based on a few powerful concepts that software developers can readily comprehend and apply. The architecture also needs to be able to be supported by frameworks or tools that make it practical to implement industrial scale systems.

3. A Way Forward: Adaptivity is a Property of Relationships

The state of the art in adaptive architectures is a long way from meeting the challenges posed in the previous section. Attempts to develop more adaptive software applications are caught on the horns of a dilemma. On one hand, an application needs to become more adaptive if it is to cope with changes in requirements and environments. On the other hand, adding this adaptive capacity can vastly increase the complexity of the application, making it difficult to develop and, perhaps paradoxically, difficult to maintain and evolve.

This dilemma arises from a misconceived view of the nature of adaptation. In our view, an entity is never intrinsically adaptable. Rather, an entity is only ever adaptable *with respect to* its existing and potential environments. Adaptation is therefore a property of the *relationships* between entities, rather than an intrinsic property of the entities themselves. It follows that the mechanisms that facilitate adaptation in a software architecture should be *extrinsic* to functional software entities that make up the application. In other words, at a fundamental level, adaptation is a property of the connectors/relationships in the software architecture.

Our approach to building adaptive applications is to resolve the indirection between decoupled entities by creating an *organisation*. An organisation is an interaction structure that is separate to the functional

entities that are loosely bound to it. Organisations define and maintain the relationships between their constituent parts. An organisation also attempts to maintain a viable relationship with its environment.

This organisational approach to architecture goes a long way in addressing the challenges posed in the previous section. Firstly, as the requirements of an entity are always *extrinsic* to the entity itself [18], it is natural to define an entity’s requirements in its connectors rather than in a static interface. These relationships are defined by rich and dynamic descriptions rather than by essential types. Secondly, qualities of an entity, in a particular computational and network context, are always qualities measured with respect to its interactions with other entities. As the required qualities of an entity are always dependent on its context, an entity’s perceived (actual) quality attributes can always be measured by its behaviour over its connectors.

In the next section we briefly describe an adaptive architecture that can represent changing requirements, and measure the performance of its (heterogeneous and distributed) constituent entities with respect to those requirements. Its adaptive capabilities are based on *exogenous management* that can restructure and regulate the entities within a composition. Based on a few concepts, the architecture enables recursive composition of self-managed composites/systems.

4. ROAD: Role-Oriented Adaptive Design

Our approach to the management indirection using organisational structures is called ROAD (Role-Oriented Adaptive Design) [6,7]. ROAD is based on a few fundamental concepts: *roles*, *contracts* and *self-managed composites* that have internal *organisers*.

Fig. 1 below illustrates the key concepts in the ROAD framework. In ROAD, *functional roles* are first-class runtime entities that hold abstract ‘service’ definitions specifying the requirements for any entity that may play that role. A role’s requirements are a dynamic aggregation of its obligations and powers with respect to other roles within its organisational composite. To use an analogy with a human organisation: an employee role has a ‘position description’ that describes the requirements for any person employed in that role; that is, the responsibilities, performance expectations and authority relationships with respect to other roles in the organisation. Roles in themselves perform no domain function; rather all “work” is done by the functional entities. ROAD does not dictate a technology for the role players. They can be objects, components, services, agents, or even humans interacting with the system using an appropriate interface [5,6]. Because the entities that play instances of roles can be transitory (for example, there may be no entity currently available to play a role), roles buffer incoming messages. Roles also perform the function of message routers, as the entities

that are bound to roles in an organisational structure do not directly reference each other. Roles may be contracted to a number of other roles (e.g. roleA in Fig.1), and therefore need to forward messages from their role-player to the appropriate associated role. The coupling between a role (as proxy for the organisation) and the role player is dynamic, and under the control of the organiser of the composite, as described below.

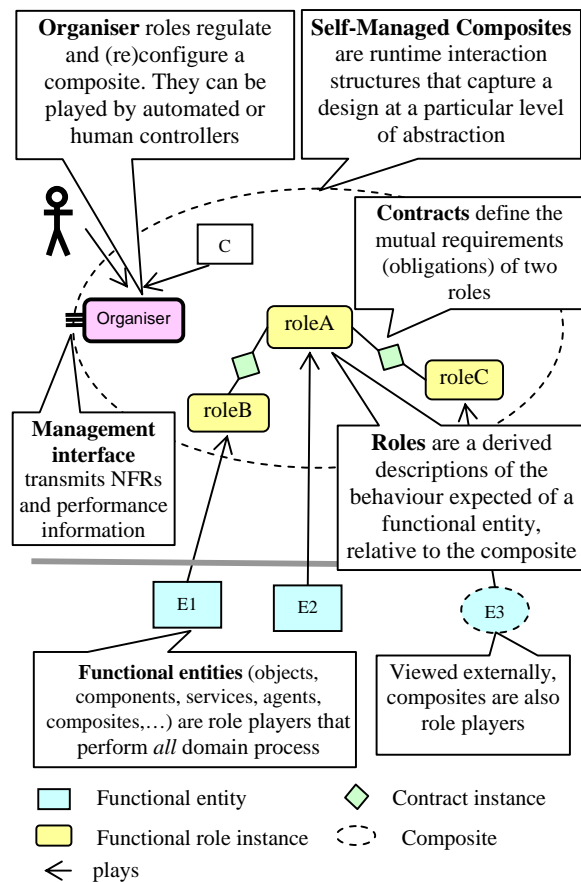


Fig. 1. The Adaptive Organisational Structure of ROAD.

Contracts perform three functions in a ROAD composite role structure: composition of roles; interaction control; and the setting of required performance and the monitoring of actual performance. ROAD contract instances are dynamic and rich connectors between roles. By creating and/or revoking contracts, the topology of the composition of roles can be altered. As all roles (as opposed to players) are internal to the organisation, ROAD contracts are also internal to the organisation. All runtime communication between functional entities bound to the organisation is *via* contracted roles, and, if necessary, contracts intercept the communications between roles. In this way contracts perform a similar function to interceptors in conventional middleware. Terms can be defined in ROAD contracts that handle synchronous,

asynchronous and deferred asynchronous transactions. Contract terms define the mutual obligations of the participant roles in an organisational context. They define the interactions that are permissible or required by the participant roles, and can be used to enforce sequences of interactions [15]. Contracts can also set any non-functional requirements (NFRs), in the form of utility objects, on their roles' interactions, and monitor those interactions for compliance to those requirements.

Organisers are managers that create and destroy roles. An organiser is itself separated into a role and a player, as shown in Fig. 1. An organiser *role* provides the mechanisms for manipulating the composite. Organiser *players* are controller/designers that decide *what* needs to be changed. Organiser roles can be played by either an automated player, or by a human operator/developer. They also make and break the bindings between composite roles and functional entities (entity selection), and create and revoke the contracts between the roles. They can thereby create various configurations of roles and entities. Organisers set performance requirements for the contracts they control, and receive performance information from those contracts. Organisers have reconfiguration strategies they can employ if they detect under-performance in the composite they control. In short, organisers provide the adaptive management of their composite.

Each organiser is responsible for the configuration of a set of roles and contracts: a *self-managed composite* (SMC). In terms of a business analogy, a managed composite in a business organisation would be a department (e.g. manufacturing department). SMCs are runtime interaction structures that define and control interactions between functional entities. An *instantiated composite* (i.e. the roles of the composite have players attached) is itself an executable entity (see entity E3 in Fig. 1). Messages to the composite are delegated to its internal role-players. An application composed of these SMCs can therefore be distributed. As a composite can itself be an entity that plays a role in another composite, it follows that composites can also be recursively composed or decomposed. At the lowest level of decomposition, all players are non-composite basic entities. *Internally*, a composite is a specification and an interaction structure. *Externally*, a composite is an executable functional entity.

Each SMC has a *management interface* which is the external interface of its organiser role. This interface performs a similar function to an 'out-of-band' manageability interface in MoWS [14], in that required and actual performance-measures pass backward and forward over it. When SMCs are recursively composed into larger composites, the network of management connections between SMC organisers forms a separate management system for the application. A more detailed discussion of the transmission of NFRs between ROAD service composites can be found in [6]

In summary, the ROAD architecture provides the basis for addressing the challenges posed in Section 2.

Requirements can be represented in roles. When these roles are operationalized by SMCs these requirements are translated to the appropriate form by the SMC's organiser and written into the composite contracts. If new or changed requirements are received over the management network, the organiser dynamically rewrites the contract terms, or reconfigures the composite by creating new contracts and roles, and binding functional roles players to those roles.

Because all interaction between functional entities in a ROAD composition takes place via contracts between roles, these contracts provided points at which performance of the entities with respect to the composite can be exogenously measured. Arbitrary utility functions can be defined on contract terms. Only those attributes of relevance then need to be measured rather than having to provide heavy-weight instrumentation of the infrastructure. If a finer-grained model of the computational and network context is required, a ROAD composite can be created for this purpose [4].

ROAD composites can compose heterogeneous entities that use different middleware standards, by providing a platform-independent mechanism for interaction and management. The composite interfaces with these entities by attaching appropriate adaptors to the roles. ROAD also supports distributed management at its management is always exogenous. ROAD composites therefore do not require internal knowledge of, or internal access to, the entities that are composed.

Finally, ROAD composites can be recursively composed and distributed. This facilitates the design of relatively simple composites that express a few relationships at a single level of abstraction. Relationships between composites and entities are always explicit and can be dynamically changed. Complexity is also controlled because composites are *self-managed*. All composites and entities are mutually opaque and do not attempt to micro-manage each other. While much work remains to be done to develop composite organisers that can handle multiple quality attributes, ROAD's relational approach to the management of adaptive architecture does provide a foundation for overcoming the challenges posed by volatile requirements and environments.

5. Conclusion

For a system to be able to adapt to more than just a limited number of environmental states, it not only needs to have a reflective understanding of itself and its requirements/goals, but also needs to have an awareness of its 'real world' context. Furthermore, it also needs to work with its constituent entities supported by heterogeneous technologies, while having the ability to flexibly manage the multiplicity of system qualities.

To be adaptable, software applications need to be modular and decoupled at various levels of abstraction. They must also provide mechanisms for resolving any indirection at runtime. The ROAD framework provides an infrastructure that supports the managed indirection of components and services, and facilitates the *exogenous* representation and monitoring of non-functional requirements. This provides the basis for building highly reconfigurable applications.

References

- [1] Batista, T., Joolia, A., and Coulson, G., "Managing dynamic reconfiguration in component-based systems," *Proc. of the European Wkshp on Softw. Architectures*, 2005.
- [2] Chen, S., Liu, Y., Gorton, I., and Liu, A., "Performance prediction of component-based systems" *Journal of Systems and Software*, vol.74(1), 2005, pp. 35-43.
- [3] Cheng, S.-W., Garlan, D., and Schmerl, B.R., "Making self-adaptation an engineering reality," *Self-star Properties in Complex Information Systems, LNCS 3460*, 2005, pp.158-173.
- [4] Colman, A., "Exogenous management in autonomic service compositions," *Proc. of the Third Intl. Conf. on Autonomic and Autonomous Sys. 2007 (ICAS 2007)*, 2007.
- [5] Colman, A. and Han, J., "Roles, players and adaptive organisations" *Applied Ontology, (to appear)*, 2007.
- [6] Colman, A., *Role-oriented adaptive design*. PhD Thesis, Swinburne University of Technology, Melbourne, Australia. <http://www.ict.swin.edu.au/personal/acolman/pub/ColmanROADThesis.pdf>, 2006.
- [7] Colman, A. and Han, J., "Using role-based coordination to achieve software adaptability" *Science of Computer Programming*, vol.64(2), 2007, pp. 223-245.
- [8] Farha, R. and Leon-Garcia, A., "Blueprint for an autonomic service architecture," *Proc. of the Intl. Conf. on Autonomic and Autonomous Systems, 2006. ICAS '06*. 2006.
- [9] Han, J., "A comprehensive interface definition framework for software components," *Proc. of 1998 Asia-Pacific Softw. Eng. Conf. (APSEC'98)*, 1998, pp.110-117.
- [10] Herring, C. E., *Viable software: The Intelligent control paradigm for adaptable and adaptive architecture*, PhD Thesis, University of Queensland, 2002.
- [11] Meyer, B. *Object-oriented software construction*, Prentice-Hall, 1988.
- [12] Mintzberg, H. *Structure in fives: designing effective organizations*, Prentice Hall, 1983.
- [13] Mukhija, A. and Glinz, M., "Runtime adaptation of applications through dynamic recomposition of components," *Proc. of the 18th Intl. Conf. on Architecture of Computing Systems (ARCS 2005)*, 2005, pp.124-138.
- [14] OASIS "Web Services Distributed Management - Management of Web Services 1.0, 9 March 2005".
- [15] Pham, L. D., Colman, A., and Schneider, J., "Dynamic protocol aggregation and adaptation for service-oriented computing," *Proc. of the Aust Soft Eng Conf.*, 2007.
- [16] Simon, H.A. *The sciences of the artificial*, MIT, 1969.
- [17] W3C "OWL Web ontology language for services", 2004.
- [18] Zave, P. and Jackson, M., "Four dark corners of requirements engineering." *ACM Transactions on Software Engineering and Methodology*, 6(1), 1997, pp. 1-30.