An Active-Architecture Approach to COTS Integration

Brian Warboys, Bob Snowdon, R. Mark Greenwood, Wykeen Seet, and Ian Robertson, *University of Manchester*

Ron Morrison, Dharini Balasubramaniam, Graham Kirby, and Kath Mickan, *University of St. Andrews*

The ArchWare framework supports COTS-based software systems that can better respond to both predicted and emergent changes arising from the use of COTS products.

COTS software products are increasingly becoming standard components for building integrated information systems. At the same time, the growth of electronic trading, turbulent market conditions, and a project-style approach to business have created a demand for information systems that can be rapidly adapted to changing business process demands. However, the ongoing development of COTS products is unpredictable as their developers and source code are rarely available.

The inherent contradiction between using long-lived, general-purpose COTS components and the demand for highly adaptable information systems creates a challenging problem.

Flexible information systems use COTS components because they cost-effectively supply required component functionality. A software architecture can capture a system design as a set of interacting components and capture the role of COTS software in "implementing" certain components. However, as we move toward a world in which programmable devices greatly outnumber people, information systems will increasingly need to address this ubiquitous-computing context, or *ambient intelligent environment*. Such dynamic environments require coping with anticipated change, such as the release of new COTS versions. However, they also require coping with emergent behavior, which arises from interactions between a system's components (including its environment) and thus can't always be anticipated.

Classical software engineering has for the most part adopted a reductionist component-engineering style toward COTS-based systems (see the "Related COTS-Based Approaches" sidebar). This style results in developers either disregarding or inadequately dealing with dynamic environments, implying that such environments—and in particular emergent behavior—are traits to suppress. However, because a large class of software systems (including many constructed with COTS components) must exploit their dynamic environments, emergent behavior is not only inevitable, it should be recognized and exploited. The architecture of such flexible systems must not only reflect the components' initial static configuration but also the ongoing reconfiguration of components, capturing the system's evolution at an architectural level.

Related COTS-Based Approaches

Classical software engineering has often given inadequate attention to the dynamic environment when developing COTS-based systems. Ignoring this problem leads to legacy systems that are unaware of their changing environment. Techniques such as parameterization, inheritance, and polymorphism help but only when the potential environment changes are predictable. Plug and socket mechanisms reduce the technical problems of connecting components but provide limited support for the semantics of component composition.

Recent specifications address the deployment and configuration of COTS-based software systems. The Object Management Group has adopted a platform-independent infrastructure that seeks to allow the automated deployment and configuration of distributed component-based systems.[1] Developers can customize the standard for different application domains (such as the CORBA Component Model; Java 2 Platform, Enterprise Edition; and .NET). The Java community also recently adopted a deployment API specification that aims at such a standard across J2EE servers.[2] These standards envisage a classical software engineering approach to the development of COTS-based software systems, however, and thus don't address the inevitable emergent behavior issues of ubiquitous computing systems.

The ArchWare framework's basic evolve-produce structure (see the main article) has many similarities to the autonomic managers and managed elements proposed for autonomic computing.[3] Indeed, both aim to create cellular self-managing components. Autonomics emphasizes the application of predefined management policies. Our evolve elements are open to environmental influence, including external user feedback, and changes can combine automatic and user-supplied elements.

Researchers have addressed some aspects of active model systems—in particular, dynamic reconfiguration. In general, however, research has approached this problem by restricting it and implementing predetermined change-management solutions wherever possible.[4–6] Many solutions include a configuration manager to ensure that no unspecified change occurs. Although using a predetermined set of allowable state changes eases the task of ensuring that a dynamically changeable system remains in an architecturally permitted state, it also excludes the possibility of dealing generally with the concept of emergent behavior.

References

1. Object Management Group, *Deployment and Configuration of Component-Based Distributed Applications*, OMG Specification ptc/2003-07-08; www.omg.org/docs/ptc/04-05-15.pdf.
2. Sun Microsystems, *Java 2 Enterprise Edition Deployment API Specification*, version 1.0, http://java.sun.com/j2ee/tools/deployment/index.jsp.
3. J. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–50.
4. R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proc. Fundamental Approaches to Software Eng.* (FASE 98), LNCS 1382, Springer-Verlag, 1998, pp. 21–37.
5. N. Lynch and A. Shvartsman, "Communication and Data Sharing for Dynamic Distributed Systems," *Future Directions in Distributed Computing*, LNCS 2584, Springer-Verlag, 2003, pp. 62–67.
6. M. Wermelingerl and J.L. Fiadeiro, "Algebraic Software Architecture Reconfiguration," *7th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE), Springer-Verlag, 1999, pp. 393–409.

An active-architecture model that changes as the system evolves can capture the integration of COTS components into a system and the composition of those components. The ArchWare framework is a process-centered approach that implements the core system as a network of evolvable cooperating processes. The active architecture captures each COTS component's role within the system and identifies how the system can incorporate new COTS or replace existing COTS as the process network evolves.

ArchWare framework

We developed and implemented the ArchWare framework approach as part of the ArchWare project.[1] We focused on developing COTS-based software systems that are inherently capable of changing and of being changed. The ArchWare framework lets us integrate COTS components into adaptable distributed software systems—*ArchWare-based information systems*.

An AIS has four fundamental elements:

- COTS components,
- ArchWare transformer/connectors (T/Cs),
- ArchWare Architecture Description Language components, and
- users.

T/Cs provide the wrappers that capture a COTS component's role in the system. The ADL components make up the active-architecture model describing the COTS integration.

The ArchWare framework uses a runtime architecture to integrate COTS components into a flexible information system. Such a system exists in a dynamic environment and must evolve

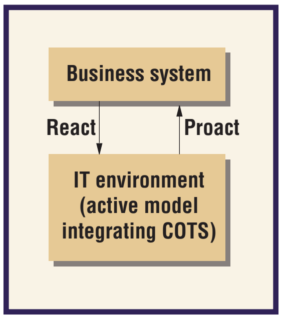


Figure 1. Coevolution of a business system and information technology environment that includes COTS software components.

to incorporate new capabilities from that environment. We've developed a set of mechanisms for integrating COTS components and incorporating control systems principles, letting the architecture model guide the system's ongoing evolution. The ArchWare ADL provides the capabilities for modeling the system's malleability, so the system can evolve while it's in operation.[2] The system evolves to extend its use of COTS components' capabilities, replace one COTS component with another, or refine its own architecture based on the COTS components available in its environment. Given this generic change capability, the ArchWare framework must also provide mechanisms for sensible engineering control of changes.

Flexible COTS integration

Our approach's philosophy emphasizes the sociotechnical nature of organizational systems—that is, the coevolution relationship existing between social and technical domains. It might therefore be appropriate to consider these systems as autopoietic (self-producing). They're continually trying to reinvent themselves to exploit the opportunities presented by their dynamic environment. We incorporate COTS components into flexible systems using an interface that explicitly models the components' roles. This interface also acts as the soft, or *coordination*, layer[3]—the flexible membrane integrating the COTS components into the overall system.

In the ArchWare framework, the soft, flexible layer that integrates COTS components into an AIS is the architecture model. As part of the runtime system, the model helps guide the system's evolution throughout its lifetime. The model is active: it maintains its state as the business system changes (the *react* relationship) and can help constrain or guide changes in the business system (the *proact* relationship), as Figure 1 shows. Executing the runtime architecture model on a process-based server orchestrates the components. Our approach is therefore a "wrapped" COTS approach. We wrap each COTS component to enable two-way communication with the active-architecture model, which manages the information system's coevolution with its sociotechnical environment.

The ArchWare framework places no constraints on the COTS components, so users can reuse available COTS as components in their AISs. A COTS component is effectively a black box that behaves as a message source and destination. The ongoing development of any COTS component is outside the AIS's domain of influence, but users might want the AIS to evolve and incorporate new capabilities from the COTS components.

We implement the active-architecture model as a set of ArchWare ADL components, including integration and control capabilities for other components. One or more ArchWare ADL environments provide the context for defining and modifying ArchWare ADL components through an ADL virtual machine that executes and manages ADL definitions.

Each COTS component has an ArchWare T/C, which forms the bridge between the COTS component and the ArchWare ADL component (see figure 2). The T/C provides a filter, exposing the COTS component capabilities that are used by the AIS. As a connector, the T/C can check that the COTS component's behavior, in terms of the pattern of observed messages, is what the ADL component expects, and vice versa. It also transforms messages between the formats expected by the components it connects.

Users interact with the system in two ways. They can be users of the COTS systems that are components of the overall system, or users who interact with the active ADL model to monitor and evolve the system. An ArchWare environment provides an interface to its own T/C, letting users of appropriate software clients interact with the active ADL model, observe its state, introduce new ADL components, and direct the system's evolution. (Although there's an ArchWare ADL client, a COTS system that interacts with the ArchWare environment's T/C can replace it.)

Figure 2 shows the simplest possible AIS, consisting of one COTS component, one T/C, and one ArchWare environment. The ArchWare environment consists of a single ArchWare component and an ADL proxy acting as the interface to the T/C.

The underlying network is basically independent of this architecture. We're currently using a Web services infrastructure (www.w3.org/2002/ws) because it's rapidly becoming a useful industry standard approach[4] for component integration. In the ArchWare framework, Web services-based behaviors implement the interactions between COTS components and T/Cs and between T/Cs and ArchWare environments.

Integrating COTS systems as components

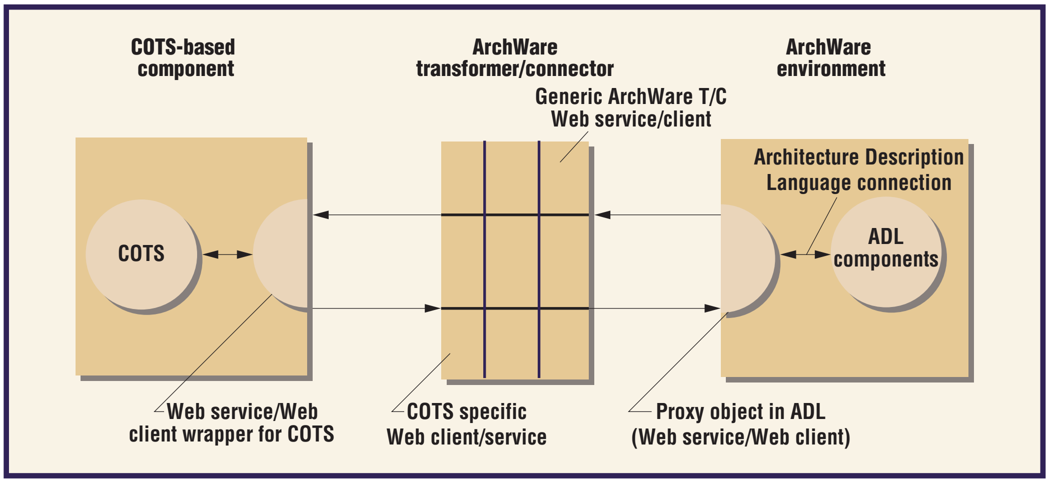

COTS-based component
ArchWare transformer/connector
ArchWare environment
Generic ArchWare T/C Web service/client
Architecture Description Language connection
COTS
ADL components
Web service/Web client wrapper for COTS
COTS specific Web client/service
Proxy object in ADL (Web service/Web client)

Figure 2. Integrating COTS and ArchWare Architecture Description Language (ADL) components with a transformer/connector (T/C).

within an AIS is essentially process integration. The active-architecture model describes the system as a set of coordinated behaviors. Within this active model, each COTS component corresponds to an ADL behavior describing its interaction with the other system components. Each COTS component's T/C performs the necessary syntactic and semantic mediation between the COTS component's domain and the active model domain. The active model-based process integration need not model all of a COTS component's potential interactions, only the current expected interactions. A COTS component that doesn't match its expected behavior indicates that the system should change, using the model's evolution capabilities.

Modeling composition and evolution

The ArchWare ADL supports the composition and evolution of component-based systems.[2] The ADL is based on the π-calculus, a formal process algebra for modeling interactive and mobile systems.[5] In an ADL model, *connections* link a set of concurrent behaviors. The behaviors interact by passing messages along these connections. The ability of behaviors to create new behaviors and connections, and to communicate connections over existing connections, lets us model dynamic component networks. The ADL also provides explicit compose and decompose operators. Through decomposition, one behavior breaks an executing subsystem into its constituent components, which it can change and recompose to form an evolved subsystem.

Hypercode technology supports the ArchWare ADL. We create a hypercode program in the ArchWare execution environment, replacing references that would otherwise need runtime binding with explicit links. A hypercode program is thus an active executing graph linking source code and existing values.[6] Hypercode unifies the concepts of source code, executable code, and data into a single representation (as a combination of source code text and hyperlinks to existing values) of software throughout its life cycle. Multiple links to the same value represent sharing. Hypercode lets developers explicitly refer to existing state, including behaviors, and shared data, including connections, when evolving an ADL model. At all times, anyone can inspect an ADL model by viewing its hypercode representation.

An ArchWare environment is a reflective system consisting of a set of subsystems whose definitions can be manipulated. When one subsystem decomposes another subsystem, it reifies the subsystem's current state, giving a hypercode representation of the subsystem's components and their connections. It can evolve these representations to capture new requirements without losing their context. The ArchWare environment includes a compiler for ADL hypercode as a callable function in the ADL so it can create evolved or new components and bind them back into the system. The ArchWare ADL capabilities aren't COTS specific; they're relevant to any situation in which components need to be composed flexibly and might need to be decomposed, modified, and recomposed while maintaining important shared context.

Exploiting cybernetic principles

Because we want systems to be responsive to both predicted and emergent change, the in-

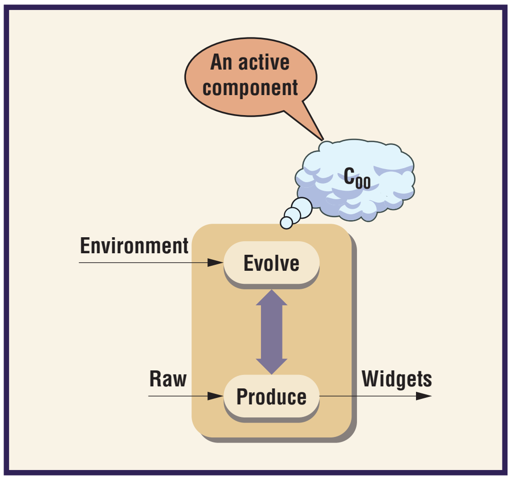


Figure 3. An ArchWare component's basic structure.

tegration of the COTS components must be evolvable at all levels of granularity. The mechanisms binding components together must be dynamically changeable, as must be the wrappings that realize the components' integration. Exploiting cybernetic principles to achieve this, we build the active architecture using elements that conform to a standard structure, ensuring the system's potential for change.

Formally, the ArchWare component is the basic building block of an ArchWare environment. As figure 3 illustrates, an ArchWare component is a process (a *behavior* in ADL terms) formed from a pair of interacting behaviors (hereafter, we'll use this term rather than process when referring to the active model in ADL):

- *Produce* (*P*) represents the component's production or operational behavior—that is, the behavior that fulfills the component's purpose. For example, if an ArchWare component is supposed to transform input "raw" to output "widgets," *P* fulfills this transformation.
- *Evolve* (*E*) represents the component's management behavior and is responsible for ensuring *P*'s effectiveness in circumstances requiring change. So, *E* affects *P*.

The interaction between the evolve and process elements in figure 3 represents both *P*'s feedback to *E* and *E*'s effect on *P*. *E* might affect *P* because *P*'s performance is deficient in some way (a standard feedback control loop). Or, *E* might affect *P* because *E* receives an external stimulus from the environment to change *P*.

This approach has roots in classical cybernetics and its application to software architecture.[7] Similarly, Mary Shaw[8] observed that an important characteristic of the control paradigm is the separation of the *operation* ("produce" in figure 3) from the compensation for external disturbances, the *control* ("evolve" in figure 3).

In general, behaviors *E* and *P* will each be ArchWare components. That is, *E* and *P* can each be formed from a further produce element and a further evolve element. This *E/P* building block lets us structure components both cooperatively and hierarchically. For example, figure 4 shows two components cooperatively bound at one level, together with an evolve coevolution component to form a hierarchical component at the next level.

Thus, we can build ArchWare components from a set of recursive *E/P* components and compose them into more complex structures. The grounding of this recursive structure can occur

- when an ArchWare component is considered to have no *E* behavior—that is, when it has no means of adaptation, and
- when the *P* behavior is considered atomic—that is, the component's architecture exposes no further structure.

For *P*, these circumstances typically occur when a COTS component implements the ArchWare component being considered (figure 5).

We construct an active-architecture model with ArchWare components in which each component's evolve parts use the ADL's evolutionary capabilities. We similarly structure T/Cs with both produce and evolve parts. Just as the produce parts of the ArchWare components and T/Cs will interact during normal operation, their corresponding evolve parts will interact to achieve any required evolutions. COTS components are outside the system's control and can be manually evolved independently. Thus, the evolution capabilities in the ADL model and the T/Cs must react to emergent COTS changes and implement predicted changes when appropriate.

Ongoing and future work

In the ArchWare project, the ArchWare framework integrates both COTS and project-specific tools from various European partners. We're evaluating the working prototype in the industrial partners' environments. The frame-

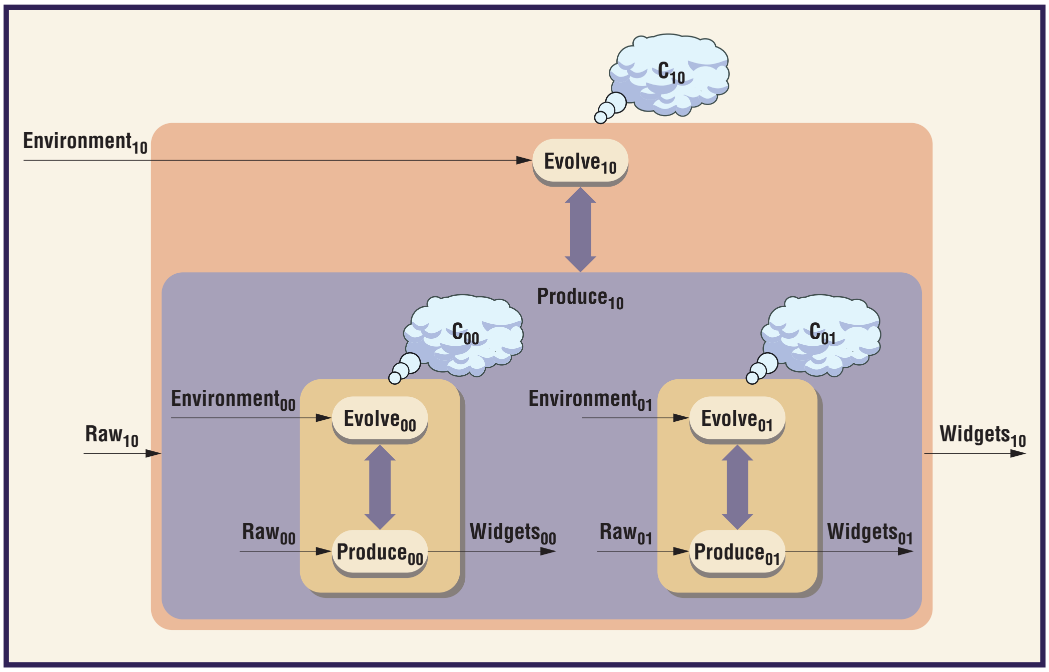


Figure 4. Composition of ArchWare components. Two components cooperatively bound at one level, together with an evolve coevolution component, form a hierarchical component at the next level.

work also builds on experience integrating COTS tools with the ProcessWeb system, an earlier persistent, reflective system designed to support system development through evolution. This is sufficient evidence to convince us that the approach can work technically, but we don't have sufficient independent experience to report any empirical results.

We're exploring two main areas of technical development for the ArchWare framework:

- developing and evolving the active-architecture models in the ArchWare ADL, and
- integrating models with the required COTS tools.

Various ArchWare project tools address the first area: the use of styles to define a domain-specific ADL variant, Unified Modeling Language stereotypes for ADL development, model checking of ADL models, and a generic refinement process that users can focus to their needs.

The second area is one motivation behind the explicit T/Cs. This lets us develop a library of generic wrapping code. By adopting Web services, we can dynamically generate wrapping code based on published interfaces and exploit existing Web service toolkits.

The active-architecture approach's major strength—its ability to incrementally evolve the system—can also be a weakness. The user's ability to evolve an architecture model when he or she notices a mistake can encourage lack of care. This ability doesn't help a user understand a complicated erroneous model and define an appropriate "correcting" evolution. The cooperative and recursive *E/P* structures are one element of good practice, but further research and experience is required. In general, asking users to browse the current system state and define an appropriate evolution is the default (last-resort) technique for resolving problems.

Because highly flexible systems can evolve in undesirable directions, we need evolution strategies to ensure that appropriate engineering discipline is applied to reduce the risk of inappropriate evolutions. Runtime verification techniques can provide early warnings when a system isn't behaving as expected and should be changed. The system evolution process can

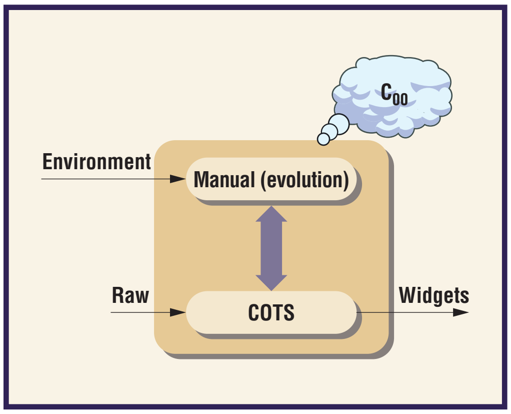
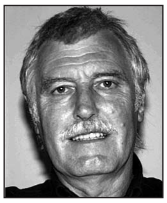
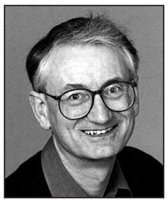


Figure 5. ArchWare component incorporating a COTS product.

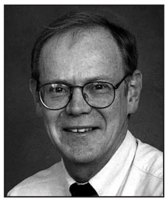
About the Authors

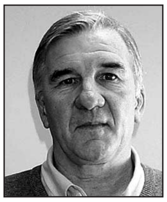

Brian Warboys is a professor of software engineering in the School of Computer Science at the University of Manchester, where he leads the Informatics Process Group. His research interests are in the development of very large software systems, which has led to a long-term interest in process modeling of both software and business systems. He received his BSc in mathematics from the University of Southampton. He's a member of the ACM and a Fellow of the British Computer Society. Contact him at the School of Computer Science, Univ. of Manchester, Kilburn Bldg., Oxford Rd., Manchester M13 9PL, UK; brian@cs.man.ac.uk.


Bob Snowdon is a research fellow in the School of Computer Science at the University of Manchester. His research interests include complex and dynamic systems, cybernetics, process modeling, decision support, and co-evolution of business and software. He received his PhD in computer science from the University of Newcastle upon Tyne. Contact him at the School of Computer Science, Univ. of Manchester, Kilburn Bldg., Oxford Rd., Manchester M13 9PL, UK; rsnowdon@cs.man.ac.uk.

R. Mark Greenwood is a research fellow in the School of Computer Science at the University of Manchester. His research interests include co-evolution of business and software, software architectures, grid and Web services, and process modeling. He received his PhD in computer science from the University of Southampton. He's a member of the British Computer Society. Contact him at the School of Computer Science, Univ. of Manchester, Kilburn Bldg., Oxford Rd., Manchester M13 9PL, UK; markg@cs.man.ac.uk.

Wykeen Seet is a research fellow in the School of Computer Science at the University of Manchester. His research interests include compliant systems, language design and implementation, software architectures, and software evolution. He received his PhD in computer science from the University of Manchester. Contact him at the School of Computer Science, Univ. of Manchester, Kilburn Bldg., Oxford Rd., Manchester M13 9PL, UK; seetw@cs.man.ac.uk.


Ian Robertson is a research fellow in the School of Computer Science at the University of Manchester. His research interests include process modeling, process evolution, decision support, and coordination technology. He received his MSc in computer science from the University of Manchester. Contact him at the School of Computer Science, Univ. of Manchester, Kilburn Bldg., Oxford Rd., Manchester M13 9PL, UK; robertsi@cs.man.ac.uk.


Ron Morrison is a professor of software engineering in the School of Computer Science at the University of St Andrews, where he leads the Systems Architecture group. His research interests include language design and implementation, hyperprogramming, distributed and persistent architectures, and compliant systems. He received his PhD in computer science from the University of St Andrews. He's a Fellow of the British Computer Society and the Royal Society of Edinburgh. Contact him at the School of Computer Science, Univ. of St. Andrews, North Haugh, St. Andrews KY16 9SX, UK; ron@dcs.st-and.ac.uk; www-ppg.dcs.st-and.ac.uk/People/Ron.

Graham Kirby is a senior lecturer in the School of Computer Science at the University of St Andrews. His research interests include autonomic storage architectures, linguistic reflection, hyperprogramming, mobile and distributed computing, and system evolution. He received his PhD in computer science from the University of St Andrews. He's a member of ACM SIGMOD. Contact him at the School of Computer Science, Univ. of St. Andrews, North Haugh, St. Andrews KY16 9SX, UK; graham@dcs.st-and.ac.uk; www.dcs.st-and.ac.uk/~graham.

Dharini Balasubramaniam is a lecturer in the School of Computer Science at the University of St Andrews. Her research interests include compliant systems, language design and implementation, software architectures, type systems, and software evolution. She received her PhD in computer science from the University of St Andrews. Contact her at the School of Computer Science, Univ. of St. Andrews, North Haugh, St. Andrews KY16 9SX, UK; dharini@dcs.st-and.ac.uk; http://www-ppg.dcs.st-and.ac.uk/People/Dharini.

Katherine Mickan is a postgraduate student in the School of Computer Science at the University of St Andrews. Her research interests include hypercode, software architectures, software evolution, and metaprogramming. She received her BSc in mathematics and computer science from the University of Adelaide. Contact her at the School of Computer Science, Univ. of St. Andrews, North Haugh, St. Andrews KY16 9SX, UK; kath@dcs.st-and.ac.uk.

incorporate a range of checks and tests before implementing any proposed change.

We also need techniques for managing user access to the mechanisms for evolving systems. These must be flexible to let users choose and change their change control policies—for example, a user might want the system's change control to be highly centralized rather than distributed. Much of our ongoing research involves refining the basic *E/P* structure to show how we can embed good engineering practice in the basic architectural structures of highly flexible systems.

Finally, we need to effectively incorporate autonomic techniques so we can develop systems that manage themselves where appropriate and allow innovation through their ability to incorporate users' unforeseen changes.

The independent development of COTS is an opportunity as well as a problem. Innovation can arise through examining new COTS features, which are unpredictable. One feature of our approach is that it's also effective for dynamically located off-the-net services, open source software, or user-created exploratory tools, which all exploit the dynamic environment's emergent behavior and require flexible integration mechanisms.

The ArchWare framework doesn't mandate a particular development process, but an AIS can integrate a set of COTS software development tools and a chosen development process. Of particular interest is the case in which both the developing software system and the developed software system are based on the ArchWare framework. Considerable flexibility exists in the relationship between these two systems. The interface between them could be based on standard release versions, with the developed system's evolution capabilities being the deployment of new versions. Alternatively, the developed system could request new capabilities from the developing system as it requires them.

In his definition of the viable system model, Stafford Beer[9] argued that a viable system is one that can maintain a separate existence. Both biological entities and successful social organizations are viable systems—that is, they can survive in their environments with some degree of autonomy. Exploiting the ArchWare framework endows a network of COTS components with this necessary autonomy.

Acknowledgments

The ArchWare European Project (www.arch-ware.org), which supported this work, is partially funded by the Commission of the European Union under contract no. IST-2001-32360 in the IST-V Framework Programme. We thank the anonymous reviewers for their comments.

References

1. F. Oquendo et al., "ArchWare: Architecting Evolvable Software," *Proc. 1st European Workshop Software Architecture* (EWSA 2004), LNCS 3047, Springer-Verlag, 2004, pp. 257–277.
2. R. Morrison et al., "Support for Evolving Software Architectures in the ArchWare ADL," *Proc. 4th Working IEEE/IFIP Conf. Software Architecture* (WICSA 2004), IEEE CS Press, 2004, pp. 69–78.
3. B.C. Warboys et al., *Business Information Systems: A Process Approach*, McGraw-Hill, Information Systems Series, 1999.
4. H. Kreger, "Fulfilling the Web Services Promise," *Comm. ACM*, vol. 46, no. 6, June 2003, pp. 29–34.
5. R. Milner, "Elements of Interaction," *Comm. ACM*, vol. 36, no. 1, 1993, pp. 78–89.
6. E. Zirintsis, G.N.C. Kirby, and R. Morrison, "Hyper-Code Revisited: Unifying Program Source, Executable, and Data," *Proc. 9th Int'l Workshop Persistent Object Systems* (POS9), LNCS 2135, Springer-Verlag, 2000, pp. 232–246.
7. C. Herring and S. Kaplan, "Viable Systems: The Control Paradigm for Software Architecture Revisited," *Proc. 2000 Australian Software Eng. Conf.*, IEEE CS Press, 2000, pp. 97–105
8. M. Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control," *ACM Software Eng. Notes*, vol. 20, no.1, 1995, pp. 27–38.
9. S. Beer, *Diagnosing the System for Organizations*, John Wiley & Sons, 1985.