An Active Architecture Approach to COTS Integration

Brian Warboys[1], Bob Snowdon[1], R Mark Greenwood[1], Wykeen Seet[1], Ian Robertson[1],
Ron Morrison[2], Dharini Balasubramaniam[2], Graham Kirby[2], Kath Mickan[2]
[1]*School of Computer Science, University of Manchester, Manchester M13 9PL, UK*
[2]*School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK*

{brian, rsnowdon, markg, seetw, robertsi}@cs.man.ac.uk
{ron, dharini, graham, kath}@dcs.st-and.ac.uk

Abstract

Commercial off-the-shelf (COTS) software products are increasingly used as standard components within integrated information systems. This creates challenges since both their developers and source code are not usually available, and the ongoing development of COTS cannot be predicted. The ArchWare Framework approach recognises COTS products as part of the ambient environment of an information system and therefore an important part of development is incorporating COTS as effective system components. This integration of COTS components, and the composition of components, is captured by an active architecture model which changes as the system evolves. Indeed the architecture modelling language used enables it to express the monitoring and evolution of a system. This active architecture model is structured using control system principles. By modelling both integration and evolution it can guide the system's response to both predicted and emergent changes that arise from the use of COTS products.

Keywords : software architecture, active architecture, COTS, hyper-code, cybernetics, evolution, integration, composition and decomposition

1 Introduction

Commercial off-the-shelf (COTS) software products are increasingly becoming standard components from which integrated information systems are constructed; it is the latest phase of component engineering approaches to software reuse. Indeed standards for the configuration and deployment of component-based software applications have started to emerge. However, at the same time, the growth of electronic trading, turbulent market conditions and a project style approach to business has created a demand for information systems which can be rapidly adapted to suit business process demands which might be subject to continuous change. The inherent contradiction between long-lived, general purpose COTS components and the demand for highly adaptable information systems creates a challenging problem. The goal is highly-flexible systems that are assembled to exploit their dynamic environment of COTS components, whose capability and availability is emergent and not predicted. In particular, the development of COTS components is unconstrained by the systems that use them.

Our approach to solving this problem is based on the observation that flexible information systems use COTS components because they are cost-effective suppliers of required component functionality. Software architecture can capture a system design as a set of interacting components, and how certain components are "implemented" by COTS software. The architecture captures the role, or roles, of each COTS component within the system. In addition, we observe that increasingly information systems will need to address the context of ubiquitous computing, or what has recently been termed the ambient intelligent environment, a world in which programmable devices greatly outnumber people. An important property of these systems is their ability to cope with this dynamic environment, where there is both anticipated change, e.g. the release of a new COTS component version, and emergent behaviour. Emergent behaviour arises from interactions between a system's components, including its environment, and by definition is not anticipated when the system is constructed.

Classical software engineering has, for the most part, adopted a reductionist component engineering style towards COTS-based systems. One consequence of this is that the dynamic nature of a system's environment has either been:

- ignored, leading to so called legacy software,
- partially treated using techniques such as parameterisation, inheritance or polymorphism,
- dealt with as a connectivity problem using plug and socket mechanisms to soften the effects of misfitting componentry.

All of these approaches leave the impression that the dynamic environment, and in particular 'emergent' behaviour, should be treated as a trait to be suppressed. Our viewpoint is different. Since a large class of software systems (including many constructed through the integration of COTS components) need to exploit their dynamic environment, emergent behaviour is not only inevitable, but should be recognised and exploited. The architecture of such flexible systems must not only reflect the initial static configuration of components. It must also reflect the ongoing re-configuration of components, capturing the system's evolution at an architectural level. Thus, it is important to design systems not merely as products but as processes capable of supporting dynamic evolution. We regard our approach as being process-centred in that the core

system is implemented as a network of evolvable cooperating processes. The architecture captures how each COTS component fits as a cooperating process within the system, and how new COTS are incorporated, or existing COTS replaced, as the process network evolves.

2 The ArchWare Framework

This paper outlines the ArchWare Framework approach that has been developed and implemented as part of the ArchWare project [1]. It addresses the development of COTS-based software systems that are inherently capable of changing and of being changed. The Framework provides an approach to the integration of COTS components into adaptable distributed software systems, ArchWare-based Information Systems (AISs). There are four fundamental elements: COTS components, ArchWare Transformer/Connectors (T/Cs), ArchWare ADL components, and Users. T/Cs provide the wrappers that capture the role of a COTS component within the system. The ADL components are the active architecture model that describes the COTS integration.

The ArchWare Framework is an example of using a run-time architecture to integrate COTS components into a flexible information system. Such a system is not isolated but exists in a dynamic environment, and must evolve to incorporate new capabilities from that environment. Our contribution lies with the development of a set of mechanisms that integrate COTS components and incorporate control systems principles so that the architecture model can guide the ongoing evolution of the system. The ArchWare ADL provides the capabilities for modelling the malleability of the system so that it can be evolved contemporaneously with its operation [2]. Through this approach the system can evolve: to extend its use of COTS components' capabilities, to replace one COTS component with another, and to refine its own architecture based on the COTS components available in its environment. In addition, given this generic change capability, the ArchWare Framework needs to provide mechanisms for sensible engineering control of changes.

3 Flexible COTS Integration

Our approach's philosophy puts emphasis on the socio-technical nature of organizational systems. It stresses that the relationship between social and technical domains is one of co-evolution. This suggests that it may be appropriate to consider these organizations as autopoietic (self-producing). They are not static; they are continually trying to re-invent themselves to exploit the opportunities presented by their dynamic environment. The incorporation of COTS components into flexible systems is based on an interface that explicitly models the role of those components. This interface also acts as the soft layer, previously we have termed this the *co-ordination* layer [3], acting as the flexible membrane integrating the COTS components into the overall system.

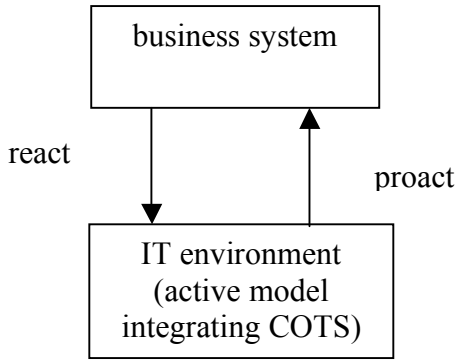

Figure 1. Co-evolution of Business System and IT environment including COTS

In the ArchWare Framework the soft, flexible layer that integrates COTS components into ArchWare-based Information Systems (AISs) is the architecture model. This model is part of the run-time system and used to guide the evolution of the system throughout its lifetime. It is an active model: it maintains its state as the business system changes (the 'react' relationship), and may be used to constrain or guide the changes in the business system (the 'proact' relationship) as in Figure 1. The execution of the run-time architectural model on a process-based server provides the orchestration of the components. This is hence an example of a 'wrapped' COTS approach. Each COTS component is 'wrapped' to enable two-way communication with the active architecture model, which takes responsibility for the management of the co-evolution of the information system with its socio-technical environment.

The ArchWare Framework does not place any constraints on the COTS components. The motivation is to allow users to reuse available COTS as components within their AISs. A COTS component is effectively a black-box that behaves as a source and destination of messages. The ongoing development of any COTS component is outside the domain of influence of the AIS, but users may want the AIS to evolve and incorporate new capabilities from the COTS components.

The active architecture model is implemented as a set of ArchWare ADL components. It includes the integration and control capabilities for other components. One, or more, ArchWare ADL Environments provide the context for the definition and modification of ArchWare ADL components, through an ADL Virtual Machine (ADLVM) that executes and 'manages' ADL definitions.

The ArchWare Transformer/Connectors (T/Cs) form the bridge between the COTS components and the ArchWare ADL components (see Figure 2). There is a T/C for each COTS component. It provides the view of the COTS component as seen from the AIS, as an application may only be using some of its capabilities. The T/C as a connector can check that the behaviour, in terms of that pattern of observed messages, of a COTS component is that expected by the ADL component, and vice versa. The T/C also transforms messages between the formats expected by the components it connects.

Users interact with the system in two ways. They may be users of the COTS systems that are components within the overall system. They may also be 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. This allows a user of an appropriate software client to interact with the active ADL model, observe its state, introduce new ADL and direct the evolution of the system. (Although there is an ArchWare ADL client, this could be replaced by a COTS system that interacted with the ArchWare Environment's T/C.)

In Figure 2 we show the simplest possible AIS composed of one COTS component, one T/C and one ArchWare Environment. The ArchWare Environment consists of a single ArchWare Component together with an ADL Proxy acting as the interface to the T/C.

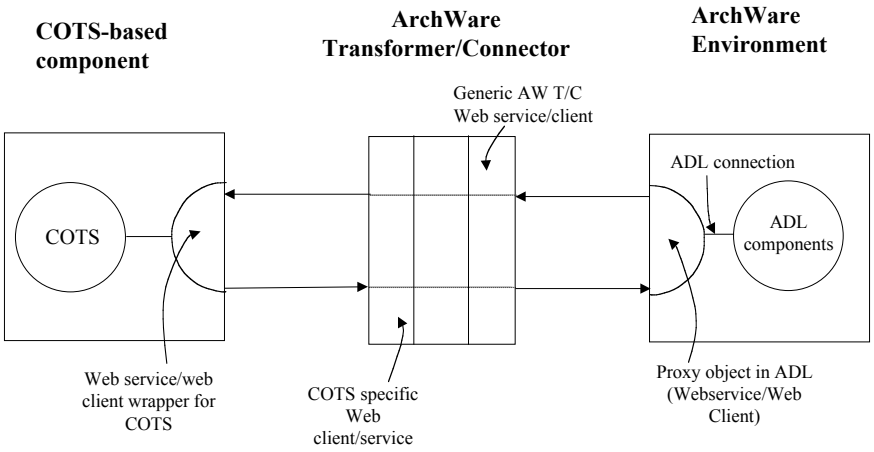

Figure 2. Integrating COTS and ArchWare ADL Components with Transformer/Connector (T/C)

The underlying network is basically independent of this architecture and at present we use a Web Services infrastructure (http://www.w3.org/2002/ws/) since this is rapidly becoming a useful industry standard for component integration. That is, web services based behaviours implement the interactions between COTS Components and T/Cs, and between T/Cs and ArchWare Environment(s).

The integration of COTS systems as components within an AIS is essentially process integration. The active architecture model, written in the ArchWare ADL, describes the system as a set of coordinated behaviours. Within this active model there is one ADL behaviour for each COTS component that describes its interaction with the other system components. The T/C for each COTS component performs the necessary syntactic and semantic mediation between the COTS component's domain and the active model domain. The active model based process integration does not need to model all the potential interactions for a COTS component. It only needs to model the current expected interactions: if the COTS component does not match its expected behaviour then this is the signal for the system to change and is handled by the evolution capabilities.

4 Modelling Composition and Evolution

The ArchWare ADL was designed to support the composition of systems from components and the evolution of such composed systems [2]. It is based on the π-calculus, a formal process algebra that was designed to model interactive and mobile systems [4]. An ADL model consists of a set of concurrent behaviours linked by connections. The behaviours interact by passing messages along these connections. The ability of behaviours to create new behaviours and connections, and to communicate connections over existing connections, allows dynamic networks of components to be modelled. The ADL also provides explicit compose and decompose operators. Decomposition breaks up (part of) an executing system into its constituent components. These can then be changed and recomposed to form an evolved system.

Hyper-code technology is used to support the ArchWare ADL. A hyper-code program is created in its execution environment. References that would otherwise need run-time binding can be replaced by explicit links. Thus a hyper-code program is an active executing graph linking source code and existing values [5]. By unifying the concepts of source code, executable code and data, hyper-code provides a single representation (as a combination of source code text and hyperlinks to existing values) of software throughout its life cycle. Sharing is represented by multiple links to the same value. Hyper-code allows developers to explicitly refer to existing state, including behaviours, and shared data, including connections, when evolving an ADL model. At all times an ADL model may be inspected by viewing its hyper-code representation.

An ArchWare Environment is a reflective system, able to manipulate its own definition. When a system is decomposed, its current state is reified giving a hyper-code representation of its components and their connections. These representations may be evolved to capture new requirements without losing their context. A compiler for ADL hyper-code is provided as a callable function in the ADL so that evolved or new components can be created and bound back into the system. The ArchWare ADL capabilities are not COTS specific; they are relevant to any situation where components need to be composed in a flexible fashion and they may need to be de-composed, modified and recomposed while maintaining important shared context.

5 Exploiting Cybernetic Principles

Since we wish systems to be responsive to both predicted and unanticipated (emergent) change, the integration of the COTS components needs to be evolvable at all levels of granularity. The mechanisms which bind components together need to be dynamically changeable, as do the wrappings which realise the integration of the components. Exploiting cybernetic principles to achieve this, the active architecture is built using elements which conform to a standard structure ensuring the system's potential for change. Each ArchWare Component contains both a 'produce' process, that represents its production or operational behaviour, and an 'evolve' process, that represents its "management" behaviour and is responsible for ensuring the continuing relevance of the production process.

Formally, the basic building block of an ArchWare Environment is the ArchWare Component. An ArchWare Component is a process (a behaviour in ADL terms) formed from a pair of interacting behaviours (we will now use this term rather than process when referring to the active model in ADL) Evolve (E) and Produce (P) as shown in Figure 3.

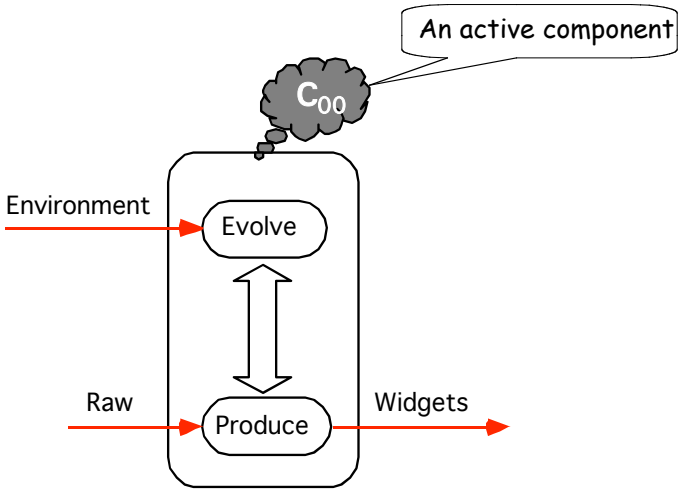

Figure 3. The basic structure of an ArchWare Component.

Produce (P) represents the behaviour which fulfils the purpose of the ArchWare Component. That is, if a particular ArchWare Component is supposed to transform input Raw to output Widgets then this transformation will be fulfilled by the behaviour P. Evolve (E) represents the behaviour which ensures the effectiveness of P in circumstances which may require change. Thus E affects P. The interaction ⇔ in Figure 3 represents both feedback from P to E and the effect of E on P. Firstly, E may affect the behaviour P because P's performance is deficient in some way (a standard feedback control loop). Secondly, E may affect the behaviour P because E receives an external stimulus from the Environment to change P.

Such an approach has its roots in classical cybernetics and its application to software architecture [6]. It was also reflected in Shaw's observations [7] 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).

Each of the behaviours E and P will, in general, be themselves ArchWare Components. That is, each of E and P can be formed from a further produce element and a further evolve element. Thus, this E/P building block allows components to be structured both co-operatively and hierarchically. For example, Figure 4 illustrates two components co-operatively bound at the same level, together with an evolve co-evolution component to form a hierarchical component at the next level.

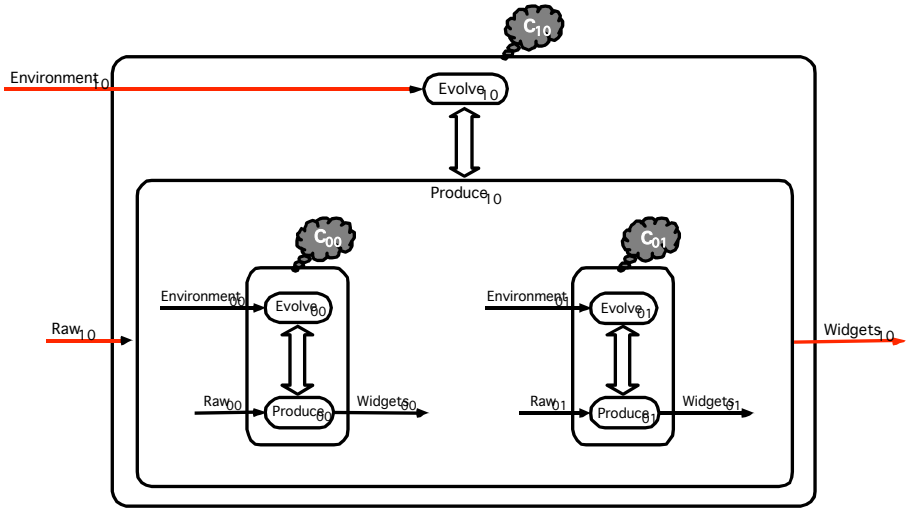

Figure 4. Composition of ArchWare Components

Thus ArchWare Components can be constructed from a set of recursive E/P components, and composed into more complex structures. The grounding of this recursive structure can occur

- when an ArchWare Component is considered not to contain an E behaviour. That is, the ArchWare Component does not have any means of adaptation.
- when the P behaviour is considered "atomic". That is, the architecture of the component does not expose any further structure.

Typically these circumstances, for the produce behaviour, occur when the ArchWare Component being considered is implemented by a COTS component (Figure 5).

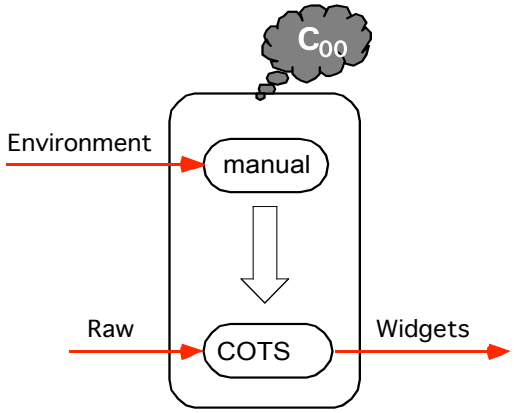

Figure 5. ArchWare component incorporating COTS product

An active architecture model is constructed of ArchWare Components, where the evolve parts of each component use the evolutionary capabilities provided by the ADL. The Transformer/Connectors (T/Cs) will similarly be structured 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 control of the system and can be manually evolved

independently. This means that the evolution capabilities in the ADL model and the T/Cs must be reactive to emergent COTS changes, as well as implementing predicted changes when the appropriate circumstances are recognised.

6 Related Approaches

Recent specifications address the deployment and configuration of COTS-based software systems. The OMG has adopted a platform independent level infrastucture which seeks to allow the 'automated' deployment and configuration of distributed component-based systems [8]. This standard can be customised for different application domains (eg CORBA Component Model (CCM), J2EE and .NET). The Java community also recently adopted a deployment API specification that aims at such a standard across J2EE servers [9]. However these standards envisage a classical software engineering approach to the development of COTS-based software systems and thus do not address the inevitable emergent behaviour issues of ubiquitous computing systems. The basic Evolve-Produce structure has many similarities with the autonomic managers and managed elements proposed for autonomic computing [10]; indeed both aim to create cellular self-managing components. Autonomics emphasises the application of pre-defined management policies. Our evolve elements are open to environmental influence, including external user feedback, and changes may combine automatic and user-supplied elements.

Some aspects of what we term active model systems have been partially addressed. In particular the issue of dynamic reconfiguration has been thoroughly studied but, in general, research has approached this problem by restricting it and implementing predetermined change management solutions wherever possible [11,12,13]. Many solutions include a configuration manager, which actually ensure that no change occurs that is not already specified. Using a predetermined set of allowable state changes of course eases the task of ensuring that a dynamically changeable system remains in an architecturally permitted state. However it also excludes the possibility of dealing generally with the concept of emergent behaviour.

7 Discussion and Further Work

The ArchWare Framework has been implemented as part of the ArchWare project where it is used to integrate both COTS and project-developed tools from various European partners. The working prototype is being evaluated in the environment of the industrial partners. The Framework 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 there is not sufficient independent experience to report any empirical results.

There are two main areas of technical development when exploiting that ArchWare Framework: developing and evolving the active architecture model in the ArchWare ADL, and integrating with the required COTS tools. Various ArchWare project tools address the former including: the use of styles to define a domain-specific ADL variant, UML stereotypes for ADL development, model checking of ADL models, and a generic refinement process that can be specialised by users. The latter is one motivation behind the explicit Transformer/Connectors. This allows the

development of a library of generic wrapping code, and by adopting web services it is possible to introspect over interfaces and exploit existing web service toolkits.

The major strength of the active architecture approach, the ability to incrementally evolve the system, can also be a weakness. The ability to evolve an architecture model when mistakes are noticed can encourage a lack of care. The ability to evolve does not help a user faced with the task of understanding 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.

One danger of a highly-flexible system is that it may evolve in an undesirable direction. Evolution strategies need to be developed to ensure that appropriate engineering discipline is applied to reduce the risk of inappropriate evolutions. Run-time verification techniques may provide early warnings when a system is not behaving as expected and should be changed. The process of system evolution may incorporate a range to checks and tests before any proposed change is implemented. There also need to be techniques that manage users' access to the mechanisms for evolving systems. These need to be flexible so that users can themselves decide and change their change control policies, for example whether it is highly centralised or distributed. Much of our ongoing research can be characterised as the development of refinements of the basic P/E structure to show how such good engineering practice can be embedded in the basic architectural structures of highly-flexible systems. Finally there is the challenge of the most effective incorporation of autonomic techniques so that we are able to develop systems that manage themselves where appropriate, and also allow innovation through their ability to incorporate unforeseen changes provided by users.

8 Conclusions

The ArchWare Framework provides the architectural building blocks for developing flexible information systems that exploit their dynamic environment and incorporating COTS systems is an essential part of this. Indeed the use of COTS as components is a vital way of incorporating innovation into ongoing system development. The independent development of COTS is an opportunity as well as a problem. Innovation may arise through examining new COTS features, which cannot be predicted. One pleasant feature of our approach is that COTS systems are not considered special: using dynamically located off the net services, open-source software, exploratory tools created by users, are all examples of the exploiting emergent behaviour of the dynamic environment and require flexible integration mechanisms.

The ArchWare Framework can be used to develop flexible systems, where the emphasis is on run-time flexibility in the incorporation of COTS systems as components. These systems can themselves be the software development environments for other software systems. The same approach applies to the incorporation of COTS both at run-time, and during the development process. The ArchWare Framework does not incorporate a particular development process, but a specific ArchWare Information System can integrate a set of COTS software development tools and a chosen development process. Of particular interest is the case where both the developing

software system and the developed software system are based on the ArchWare Framework. There is considerable flexibility over the relationship between these two systems. The interface between them could be based on a standard release version style, with the evolution capabilities of the developed system being the deployment of new versions. As an alternative to this 'push' approach, the developed system could request new capabilities from its developing system as and when it requires them. The ArchWare Framework approach gives considerable flexibility in the integration strategy for the COTS components, and value chains of systems can be assembled.

Stafford Beer [14] argued in his definition of the Viable System Model (VSM) that a Viable System is one that is able to maintain a separate existence, a separate identity. This is a characteristic of biological entities and also successful social organizations. That is such entities are viable if they can survive in their environments with some degree of autonomy. We argue that exploiting the ArchWare Framework endows a network of COTS components with this necessary autonomy.

9 Acknowledgements

This work has been supported by the ArchWare European Project (http://www.arch-ware.org) which is partially funded by the Commission of the European Union under contract No. IST-2001-32360 in the IST-V Framework Programme. Thanks are also due to the anonymous reviewers for their advice on improvements

10 References

1. Oquendo, F, Warboys, BC, Morrison, R, Dindeleux, R, Gallo, F, Garavel, H, and Occhipinti, C., "ArchWARE: Architecting Evolvable Software" *Proc. 1st European Workshop in Software Achitecture (EWSA 2004)*, LNCS 3047, Springer-Verlag, 2004, pp. 257-277.
2. Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cimpan, S., Warboys, B., Snowdon, B., and Greenwood, RM., "Support for evolving software architectures in the ArchWare ADL" *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, June 2004, pp 69-78.
3. Warboys B.C., Kawalek P., Robertson T., and Greenwood R.M.: *Business Information Systems: a Process Approach*. McGraw-Hill, Information Systems Series, 1999.
4. Milner, R. "Elements of Interaction", *Comm. of the ACM* vol. 36 no. 1, 1993, pp. 78-89.
5. Zirintsis, E, Kirby, GNC, Morrison, R., "Hyper-Code Revisited: Unifying Program Source, Executable and Data" *Proc. 9th Int. Workshop on Persistent Object Systems (POS9)*, LNCS 2135, Springer-Verlag, 2000, pp. 232-246
6. Herring, C., Kaplan, S., "Viable systems: the control paradigm for software architecture revisited" *Proc. 2000 Australian Soft. Eng. Conf.*, Apr. 2000, pp. 97-105
7. Shaw, M., Beyond Objects: A software design paradigm based on process control, *ACM Software Engineering Notes* vol. 20 no.1, 1995
8. Object Management Group (OMG). *Deployment and Configuration of Component-Based Distributed Applications*. OMG specification ptc/2003-07-08.
9. Java 2 Enterprise Edition Deployment API Specification, Version 1.0.
10. Kephart, J, and Chess, DM. "The Vision of Autonomic Computing" IEEE Computer Journal vol. 36 no.1, 2003, pp. 41-50

11. R. Allen, R. Douence, D. Garlan, “Specifying Dynamism in Software Architectures” *Proc. Foundations of Component-Based Systems Workshop*, Sept. 1997
12. N. Lynch, A. Shvartsman, “Communication and Data Sharing for Dynamic Distributed Systems” *Future Directions in Distributed Computing*, LNCS 2584, Springer-Verlag, 2003.
13. M. Wermelingerl, and J. L. Fiadeiro, “Algebraic Software Architecture Reconfiguration”, FSE’99, 1999.
14. Beer, S., *Diagnosing the system for organizations*, 1985, UK: Wiley and Sons