A Software Architecture Approach for Structuring Autonomic Systems

Dharini Balasubramaniam, Ron Morrison, Graham Kirby, Kath Mickan
University of St Andrews
St Andrews
Fife KY16 9SX, UK
+44 1334 463253
{dharini, ron, graham, kath}@dcs.st-and.ac.uk

Brian Warboys, Ian Robertson, Bob Snowdon, R Mark Greenwood, Wykeen Seet
University of Manchester
Oxford Road
Manchester M13 9PL, UK
+44 161 275 6154
{brian, ir, rsnowdon, markg, seetw}@cs.man.ac.uk

ABSTRACT

Autonomic systems manage themselves given high-level objectives by their administrators. They utilise feedback from their own execution and their environment to self-adapt in order to satisfy their goals. An important consideration for such systems is a structure which is conducive to self-management. This paper presents a structuring methodology for autonomic systems which explicitly models self-adaptation while separating functionality and evolution. Our contribution is a software architecture-based framework combining an architecture description language based on π-calculus for describing the structure and behaviour of autonomic systems, a development methodology for evolution and mechanisms for feedback and change.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *Data abstraction, Domain-specific architectures, Languages, Patterns.*

General Terms

Management, Measurement, Design, Experimentation, Languages.

Keywords

Autonomic systems, feedback, change, structuring, software architectures, producer, evolver.

1. INTRODUCTION

1.1 Autonomic Systems

Autonomic systems manage themselves given high-level objectives from administrators [1]. Common goals of self-management are: self-configuration, self-optimisation, self-healing and self-protection. In order to achieve these, an autonomic system needs to monitor itself continuously, receive feedback from its execution and its environment and utilise this information, along with knowledge of system goals, for self-maintenance.

Thus policy decisions for autonomic systems include what, when and how changes should be made to maintain the system compliant to its objectives. Mechanisms to support these policies include specification of constraints which encode systems goals, feedback from execution, feedback from the environment and ability to make changes to parts of the system as it executes.

We introduce a framework which combines the above facilities to provide an integrated environment within which the development and execution of autonomic systems may take place. By adopting a software architecture approach, we provide a formal, verifiable basis for autonomic systems.

1.2 A Software Architecture Approach

Software architectures [3,4] describe systems in terms of components and their interactions. Components may be composed together to form larger components or final systems. Thus they offer a foundation for describing the structure and behaviour of systems at various levels of abstraction.

Support for a unified, software architecture-based framework for autonomic systems requires the following:

- an architecture description language (ADL) supporting
  - architecture specifications which capture both the structure and the behaviour of components and interactions so that observations and changes may be made with a single framework
  - specification of constraints
  - mechanisms for feedback and change
- support for integrating external (those outwith the ADL domain) components into ADL systems
- a structuring methodology designed for change which reconciles architectures and the above mentioned mechanisms

ArchWare ADL [5] is the architecture description language used by the framework. It is a strongly-typed executable architecture description language based on higher-order polyadic π-calculus [6] and was developed as part of the ArchWare project [7]. It was specifically designed for modelling dynamic, evolving architectures and thus is particularly suited to capturing the structure and behaviour of autonomic systems. The language and its support for constraints [8], feedback [9], change [10] and integration of COTS components [11] have been described elsewhere. This paper presents the structuring methodology for autonomic change based on the concepts of producers and evolvers [12]. A brief summary of the required mechanisms is provided in Section 3 for use in later examples.

2. RELATED WORK

Various frameworks and methodologies have been proposed for the development of autonomic systems.

IBM's autonomic vision [1,2] envisages systems consisting of collections of autonomic elements. Each such autonomic element consists of an autonomic manager and one or more managed elements. The managed elements are designed to enable the manager to monitor and control them. The manager observes feedback from the elements and their environment and uses this information, along with knowledge of system goals, to plan and execute changes to the managed elements.

Accord [13] is a component-based framework in which autonomic applications are defined by dynamic composition and management of autonomic components. In addition to its functionality, an autonomic component encapsulates rules, constraints and mechanisms for self-management including a rule agent which manages the component's execution. Rules may be defined dynamically.

The Rainbow framework [14] uses external control mechanisms based on software architecture models to monitor and adapt executing systems. A primary goal of this framework is to enable adaptation strategies and infrastructure to be reusable across different systems. Abstract architecture models are maintained by the architecture layer of the framework while the application is executed in the system layer. Both layers interact through a translation layer.

Sterritt and Bustard [15] describe an autonomic environment containing autonomic elements which communicate asynchronously. Each element consists of a managed component and an autonomic manager. The manager uses feedback on the state of the managed component and the state of the environment to make any necessary changes to the managed component. Each component also emits a 'pulse' on a global signal channel. These are used by monitors (either dedicated or distributed) to track the health of components.

The vGrid [16] architecture for autonomic systems consists of three layers: the autonomic problem solving environment, the vGrid infrastructure services and the autonomic grid application execution environment. The first layer is a software development environment which allows high level autonomic policies to be specified during application development. The second layer provides enhanced grid middleware to support autonomic applications while the third layer is responsible for monitoring and controlling execution.

Some of these frameworks support autonomic facilities by using structures based on external managers or global monitors. In addition, they do not address issues arising from the need to potentially evolve every part of the system including feedback and managers themselves. The advantage of our approach is that systems are structured by a methodology allowing localised change in every component thus providing them with integrated support for autonomics.

3. SUMMARY OF ADL MECHANISMS

We introduce the required mechanisms for the autonomic framework using the ArchWare approach in this section. The feedback mechanism uses software probes for generating feedback and connections for communicating probes and feedback between sources and sinks. The change mechanism utilises the concepts of change functions and change connections. The framework also supports a mechanism for integrating external components into ADL systems.

Both feedback and change mechanisms make use of the hyper-code technology. A hyper-code [17] program is an active executing graph linking source code and existing values. Hyper-code provides a single representation (as a combination of text and hyperlinks) of software throughout its lifecycle. Sharing is represented by multiple links to the same value.

3.1 Feedback

Feedback acts as a trigger for change in autonomic systems. A component may receive feedback from any of the following sources:

- another component written in the ADL
- the execution engine on which the ADL is being evaluated
- the environment external to the ADL

The ArchWare ADL provides a uniform mechanism to deal with feedback from all these sources. The feedback mechanism is structured in terms of feedback sinks, feedback sources, software probes, probe connections and feedback connections.

Each feedback source is designed to publish its feedback interface, accept functions from the probe connection, interpret them as probes, bind them internally and invoke them at the appropriate time.

Feedback sinks are structured to define software probes, send them to feedback sources via the appropriate probe connections, receive the feedback via feedback connections and take appropriate action to correct any anomalies.

Software probes are defined by functions using application constraints and hyperlinks to the observable features published by the target feedback source. The function body consists of an if-do clause with the if-part representing the negation of a constraint and the do-part the feedback to generate if the condition is true.

Connections for communicating probes and feedback are defined differently for each type of feedback source but once created can be used identically. Feedback connections act as the event distribution network.

3.2 Change

3.2.1 Support for Change

The following kinds of change for autonomic adaptation are supported by the ADL:

- update and replacement
- static and dynamic generation of new components
- dynamic evolution (decomposition, reification, reflection, recomposition)

All language mechanisms required to support the above changes maintain type safety in the ArchWare ADL.

Components are modelled by behaviours (analogous to processes in π-calculus) in the ArchWare ADL. They communicate via connections (channels in π-calculus) using send and receive actions. Behaviours can be collaboratively and hierarchically composed to form a system. A *compose* operator (akin to "|" in the π-calculus) creates a single handle to a number of executing behaviours. Abstractions abstract over behaviours just as functions abstract over expressions. Mutability is explicitly modelled by locations.

Update is performed when a location is assigned a new value. Replacement of statically defined components is supported by assigning a new component to a location containing the old one. There are two ways of generating new instances of component types. If the number of components and time of creation are known statically then abstraction definition and application can be used. If component creation depends on some dynamic input then replication ("!" in π-calculus) may be used.

A more challenging adaptation is where part of a system has to be (partially) disassembled, changed and put back together to create an evolved system while the unaffected part continues to execute. This change requires support for

- decomposition,
- reification,
- reflection, and
- recomposition.

Decomposition [18] takes (part of) an executing system, breaks it up into its constituent components and returns them in a partially suspended state. The ADL supports a *decompose* operator which takes a composite component and returns a sequence of its constituent components (behaviours).

Reification allows introspection of a component so that its specification can be used as the basis for any change. The specification of a component is always available including during execution and after decomposition via the ArchWare ADL hyper-code system [19].

The specification of a component can be edited using the hyper-code system to produce an evolved specification. Using hyperlinks to denote existing values allows us to preserve shared data through this evolution.

Reflection allows new or evolved components to be bound back into an executing system. The evolved specification is brought into the execution domain by dynamic compilation. A callable compiler is provided by the ADL to implement reflection.

Recomposition takes the evolved set of components and composes them together to form a new system. The *compose* operator provided by the ADL can be used to achieve this.

3.2.2 Change Mechanism

Since feedback triggers change, the change mechanism can be described as an extension of the feedback mechanism. In addition to feedback sinks, feedback sources, feedback connections, probe connections and probes, we introduce the notions of change connections and change functions.

Feedback sources publish a change interface in addition to their feedback interface. This interface contains hyperlinks to locations of units of functionality within the source which may be modified by feedback sinks. These units are typically behaviours.

When a feedback sink receives some feedback from a probe executed by a feedback source, it may decide that changes are necessary to some part of the feedback source. It then defines a change function to update the necessary part of the source.

Change functions are constructed using the change interface of the feedback source and the ADL constructs for supporting change described in section 3.2.1. The former gives access to the required units of functionality within the source while the latter implement the required change.

The feedback sink sends the change function along a change connection to the feedback source. The source is designed to interpret messages on the change connection as change functions and execute them accordingly.

This mechanism supports changes within a feedback source. A complementary change mechanism, also supported by the ArchWare framework, allows a feedback sink to directly manipulate a feedback source, either by decomposing the source, evolving its constituents and recomposing them to form a changed source or by replacing one source component with another. The examples in this paper deal with changes to parts of feedback sources.

3.3 Integration of External Components

Commercial off-the-shelf (COTS) components are increasingly being used as building blocks for software systems. Thus any framework for realistic software development must provide facilities for incorporating COTS components into systems.

The ArchWare architecture framework was designed to allow external components, including COTS and legacy systems, to be incorporated into ADL systems [11]. Furthermore these systems are designed to be capable of evolving and being evolved.

Each external component in the framework has a corresponding Transformer/Connector (T/C) and an ADL proxy component. The former acts as a bridge between the external component and the ADL domain providing a view of the external component as seen from the ADL. It also translates messages into formats required by either side. The latter is used as an ADL interface to the T/C.

The underlying network to support communication between external components and the ADL domain is independent of this architecture. A web services infrastructure is currently used for this purpose.

4. STRUCTURING METHODOLOGY

Structuring systems in a manner amenable to change will ease and localise the process of autonomics. Producer/Evolver (P/E) [12] is a methodology developed by the process modelling community for building evolvable systems.

4.1 Producers and Evolvers

Each component in the system is modelled as a pair of subcomponents: a producer and an evolver. The producer carries out the functionality of the component while the evolver deals with its autonomic aspects, changing the producer in response to environmental or internal feedback. There are two ways of developing a component using this methodology:

- the developer may define both the producer and evolver subcomponents

- the developer may define an evolver which creates and installs a producer at initialisation

Capturing the functionality and evolution of a component in different subcomponents serves to maintain the separation of concerns and reduce the overall complexity of an autonomic system by localising change.

4.2 Combining P/E, Constraints, Feedback and Change

The producer/evolver model conforms to the autonomic vision described in [1]. It makes use of the mechanisms for feedback and change described in Section 3 to satisfy the goals set for each component. In this case the evolver acts as feedback sink and the producer as feedback source.

The evolver contains policies and goals for the component encoded as constraints. Its functions include:

- sending probes to the producer via the probe connection to monitor the execution of the producer
- receiving feedback from the probes via feedback connections
- receiving feedback from the environment (users and other components) via the environment feedback connection
- comparing the feedback to constraints specified for the component
- defining a change function for any necessary corrections to the producer
- sending the change function to be executed by the producer via the change connection

The producer in turn is designed so that it supports the activities of the evolver in addition to carrying out its functionality. Its duties include

- receiving probes from the evolver via the probe connection, binding them into its execution and invoking them at the appropriate time
- receiving change functions from the evolver via the change connection and invoking them

Probes are defined so that they communicate directly with the evolver via feedback connections. Thus producers are not required to send feedback to their evolvers.

The structure of a component developed using P/E is shown in Figure 1.

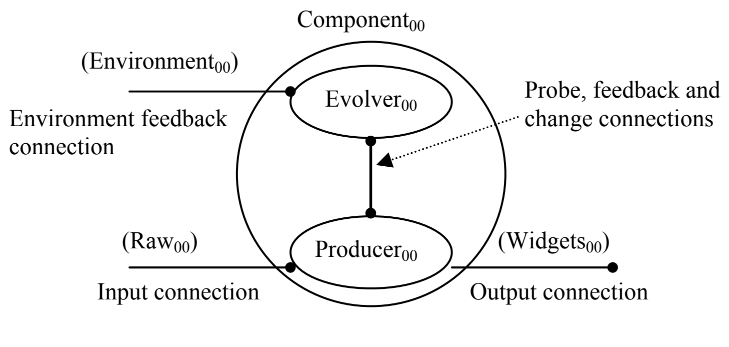

Figure 1: Structure of a Component

In Figure 1, the functionality of *Component$_{00}$* is to receive *Raw$_{00}$* via its input connection, process it to produce *Widgets$_{00}$* and send the result via its output connection. This function is carried out by the *Producer$_{00}$* subcomponent. *Evolver$_{00}$* monitors and maintains *Producer$_{00}$*. Probe and feedback connections between the two subcomponents enable the communication of probes from *Evolver$_{00}$* to *Producer$_{00}$* and feedback from probes executing in *Producer$_{00}$* to *Evolver$_{00}$*. *Evolver$_{00}$* also receives feedback from the environment via the environment feedback connection. *Evolver$_{00}$* sends any necessary change functions to be performed on *Producer$_{00}$* via the change connection.

In accordance with these requirements, the definitions of producers and evolvers are expected to conform to structures similar to those shown in Figures 3 and 4 specified in the ArchWare ADL. For clarity, non-generic types and values are shown as hyperlinks. This code assumes that the evolver and the producer are both defined by the developer and that all probes share a single feedback connection.

Figure 2 below defines the required connections for a component.

```
! external connections
value input_conn = connection( input_type )
value output_conn = connection( output_type )
value env_feedback_conn = connection( feedback_type )

! connections between producer and evolver
value probe_conn = connection( probe_type )
value feedback_conn = connection( feedback_type )
value change_conn = connection( function[] )
```

Figure 2: Definition of Connections

input_conn, *output_conn* and *env_feedback_conn* are the input, output and environment feedback connections for the component with message types *input_type*, *output_type* and *feedback_type* respectively.

probe_conn, *feedback_conn* and *change_conn* are the connections to be used between the producer and evolver subcomponents for exchanging probes, feedback and change functions. They communicate messages of types *probe_type*, *feedback_type* and *function[]* respectively.

Figure 3 shows the structure of the producer subcomponent defined as the abstraction *producer_abs*. An instance of the subcomponent can be obtained by applying this abstraction.

```
! producer structure
value producer_abs = abstraction()
{   value producer_state = location( initial_state )
    publish_feedback_interface( producer_state )
    value producer_unit = location( behaviour{
        replicate {
            via input_conn receive raw
            value widgets = do_produce( raw )
            execute_probes()
            via output_conn send widgets }
        } )
    publish_change_interface( producer_unit )
    compose{
        b1 as { replicate {
                    via probe_conn receive probe
                    install_probe( probe ) }
              } and
        b2 as { replicate {
                    via change_conn receive delta
                    execute_change( delta ) }
                }
    }
}
```

Figure 3: Structure of Producer

The producer defines its initial state as a location and publishes it as its feedback interface. It then defines its main functionality, *producer_unit*, as a location of behaviour. This behaviour repeats the following actions: it receives its input on the input connection *input_conn* and binds the input to the name *raw* in its execution; it then produces *widgets* using *raw*; it executes all the probes it has received; finally it sends *widgets* on the output connection *output_conn*. The repeated functionality is modelled by the *replicate* construct, guarded by input on *input_conn*. The producer then publishes its change interface containing *producer_unit*.

The interaction of the producer with the evolver is defined as a composition of two behaviours. The *compose* operator takes a number of behaviours, each with a unique label, and returns a handle to the parallel execution of these behaviours.

The behaviour corresponding to the label *b1* repeatedly receives a probe on the probe connection *probe_conn*, binds it to the name *probe* and installs the probe as required.

The behaviour labelled as *b2* repeatedly receives a change function on the change connection *change_conn*, binds it to the name *delta* and executes *delta*.

The parallel execution of the behaviour contained in *producer_unit* and the two behaviours mentioned above forms the functionality of *producer_abs*.

Figure 4 shows the structure of the evolver subcomponent defined as the abstraction *evolver_abs*.

```
! evolver structure
value evolver_abs = abstraction()
{   replicate {
        choose{
            { via feedback_conn receive feedback }
            or
            { via env_feedback_conn receive feedback } }
        value need_feedback_change = location( false )
        value need_producer_change = location( false )
        process_feedback( feedback, need_feedback_change,
                          need_producer_change )
        if 'need_feedback_change do
        {   value new_probe = define_new_probe()
            via probe_conn send  new_probe }
        if 'need_producer_change do
        {   value change_function = define_evolution()
            via change_conn send change_function }
    }
}
```

Figure 4: Structure of Evolver

The evolver repeatedly performs the following actions. It receives feedback either on *feedback_conn* from the execution of the producer or on *env_feedback_conn* from the environment. In either case, feedback is bound to the name *feedback* in the execution. The *choose* construct in the ArchWare ADL ("+" in π-calculus) is used for the non-deterministic selection of one behaviour from two or more.

The evolver then defines two Boolean variables *need_feedback_change* and *need_producer_change*. These are used to track the changes required by feedback. The evolver processes the feedback and updates the Boolean variables as necessary.

If a change to the feedback being received is required then the evolver defines a new probe using the feedback interface of the producer and sends it to the producer on *probe_conn*. If a change to the producer subcomponent is necessary then the evolver defines a change function to carry out the required change using the change interface of the producer and sends the function to the producer on *change_conn*. Both changes can be triggered by the same feedback.

Given these definitions of a *producer_abs* abstraction and an *evolver_abs* abstraction, a component can be defined as shown in Figure 5.

```
! definition of component
value component = compose{  P as producer_abs() and
                            E as evolver_abs() }
```

Figure 5: Definition of A Component

An application of *producer_abs* with the label *P* and an application of *evolver_abs* with label *E* are composed together to form *component*.

4.3 Composition of Components

Components can be collaboratively and hierarchically composed to form composite components. One or more components may form the producer part of another component at a higher level of abstraction. Figure 6 shows a composite consisting of an evolver and a producer made up of two components.

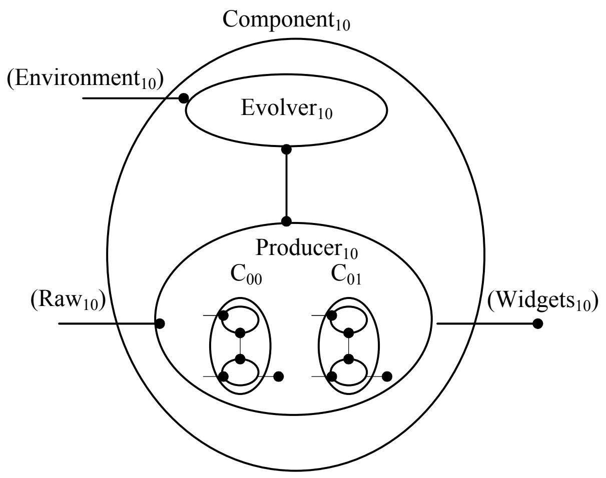

Figure 6: A Composite Component

In Figure 6, the functionality of *Producer$_{10}$* is to convert *Raw$_{10}$* into *Widgets$_{10}$*. This is achieved by the collaboration of *C$_{00}$* and *C$_{01}$*. Each of these components itself conforms to the producer/evolver model.

The *compose* operator of the ArchWare ADL can be used for the composition of components into composites as well as the composition of a producer and an evolver to form a component. Thus the example in Figure 6 can be specified in the ADL as shown in Figure 7.

```
value producer10 = compose{ c00 as c00_abs() and
                            c01 as c01_abs() }

value component10 = compose{  P as producer10 and
                              E as evolver10_abs() }
```

Figure 7: Specification of a Composite Component

Our thesis is that this methodology is well-suited to building autonomic systems. P/E requires an evolver component to be produced for every functional component of the application at construction time. This ensures that systems are built with evolution in mind in addition to achieving an elegant separation of functionality and change. At the highest level of the compositional hierarchy we have a component (the final system) without a corresponding evolver. Evolution at this level requires external intervention.

4.4 Grounding the Model

All systems, at the finest level of granularity, are made up of atomic components. The producer/evolver structuring is grounded at the level where a component can no longer be divided into such a pair. The granularity for grounding in such a case is determined by the goals of the autonomic system. An external component may also be deemed atomic if it provides no suitable interface for adaptation.

Figure 8 shows a composite component whose producer part consists of two atomic components C_{00} and C_{01}.

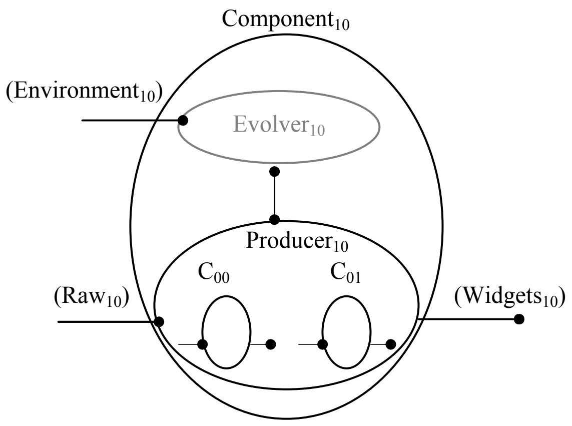

Figure 8: Atomic Components

Change at this level also requires a different mechanism. If, for example, feedback from $Producer_{10}$ indicates to $Evolver_{10}$ that the functionality of C_{00} might need to be altered then a possible strategy for change would be for $Evolver_{10}$ to decompose $Producer_{10}$ to gain access to its subcomponents, replace C_{00} with a new component C_{00}' and recompose C_{00}' and C_{01} to form the altered $Producer_{10}'$.

The above evolution may be specified in the ArchWare ADL as shown in Figure 9.

```
value decomposed_parts = decompose producer10
value c00 = decomposed_parts::1.behaviour
value c01 = decomposed_parts::2.behaviour
value c00_dash_abs = alter_component( c00 )

value producer10_dash = compose{   c0 as c00_dash_abs()
                                   and
                                   c1 as c01}
```

Figure 9: Change to Atomic Components

The decomposition of *producer10* returns a sequence containing behaviours *c00* and *c01*. Individual behaviours can be accessed by indexing into this sequence. An evolved abstraction *c00_dash_abs* is defined based on the definition of *c00*. The *producer10_dash* is then defined by composing the application of *c00_dash_abs* and the original *c01* together.

5. CONCLUSIONS

This paper has presented a structuring methodology based on the concepts of producers and evolvers as part of a unified framework for building autonomic systems. The novelty of our approach is the combination of feedback and change mechanisms within a π-calculus based strongly typed executable architecture description language and a development methodology designed for change. Our contention is that given all the above facilities, autonomic systems can be constructed and maintained within a single framework and with a formal basis for checking.

The ArchWare ADL hyper-code system has been used to develop substantial amount of code using the producer/evolver methodology presented here. Our experience suggests that it is beneficial in developing evolvable systems.

One of the avenues for further research is the integration of constraints into the evolution framework. At present we are able to evolve both architectures and behaviours as they execute. Our contention is that the framework will be complete when constraints can be evolved in a similar manner. Further work also needs to be done regarding the implications of that ability for autonomic systems.

6. ACKNOWLEDGMENTS

Some of the work presented in this paper has been supported by the EC grant IST-2001-32360 (the ArchWare project). Our thanks are due to fellow partners of the project for useful discussions relevant to the work described.

7. REFERENCES

[1] Kephart, J, Chess, DM. *The Vision of Autonomic Computing.* In: IEEE Computer Journal, Vol. 36, No.1. 2003. pp 41-50.

[2] IBM Autonomic Computing. http://www-306.ibm.com/autonomic/index.shtml.

[3] Perry, D, Wolf, A. *Foundations for the Study of Software Architecture.* In: ACM SIGSOFT Software Engineering Notes, Vol 17, No 4. 1992. pp 40-52.

[4] Garlan, D, Shaw, M. *An Introduction to Software Architecture.* In: Advances in Software Engineering and Knowledge Engineering, Vol 2. 1993. pp 1-39.

[5] Balasubramaniam, D, Morrison, R, Kirby, GNC, Mickan, K, Norcross, S. *ArchWare ADL - A User Reference Manual*. 2004. ArchWare Project Report.

[6] Milner, R. *Communicating and Mobile Systems: The Pi-Calculus*. 1999: Cambridge University Press.

[7] Oquendo, F, Warboys, BC, Morrison, R, Dindeleux, R, Gallo, F, Occhipinti, C. *ArchWare: Architecting Evolvable Software*. In: *Proc. First European Workshop on Software Architecture (EWSA'04)*. 2004. St Andrews, UK. Springer-Verlag. pp 257-271.

[8] Cimpan, S, Oquendo, F, Balasubramaniam, D, Kirby, GNC, Morrison, R. *ArchWare ADL:Definition of Textual Concrete Syntax*. 2002. ArchWare Project Report.

[9] Balasubramaniam, D, Morrison, R, Mickan, K, Kirby, GNC, Warboys, BC, Robertson, I, Snowdon, R, Greenwood, RM, Seet, W. *Support for Feedback and Change in Self-adaptive Systems*. In: *Proc. ACM SIGSOFT Workshop on Self-managed Systems (WOSS'04)*. 2004. Newport Beach, CA, USA. ACM.

[10] Morrison, R, Kirby, GNC, Balasubramaniam, D, Mickan, K, Oquendo, F, Cimpan, S, Warboys, BC, Greenwood, RM. *Support for Evolving Active Architectures in the ArchWare ADL*. In: *Proc.4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004*). 2004. Oslo, Norway. IEEE Computer Society. pp 69-78.

[11] Warboys, BC, Snowdon, R, Greenwood, RM, Seet, W, Robertson, I, Morrison, R, Balasubramaniam, D, Kirby, GNC, Mickan, K. *An Active Architecture Approach to COTS Integration*. Submitted to: IEEE Software Special Issue on Incorporating COTS into the Development Process. 2004.

[12] Warboys, BC, Kawalek, P, Robertson, I, Greenwood, RM. *Business Information Systems: A Process Approach*. 1999. McGraw-Hill.

[13] Liu, H, Parashar, M, Hariri, S. *A Component Based Programming Framework for Autonomic Applications.* In: *Proc. First International Conference on Autonomic Computing (ICAC'04)*. 2004. New York, USA. IEEE Computer Society. pp 10-17.

[14] Cheng, S, Huang, A, Garlan, D, Schmerl, B, Steenkiste, P. *Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure.* In: *Proc. First International Conference on Autonomic Computing (ICAC'04)*. 2004. New York, USA. IEEE Computer Society. pp 276-277.

[15] Sterritt, R, Bustard, D. *Towards an Autonomic Computing Environment*. In: *Proc.14th International Workshop on Database and Expert Systems Applications (DEXA'03)*. 2003. Prague, Czech Republic. IEEE Computer Society. pp 699-703.

[16] Khargharia, B, Hariri, S, Parashar, M, Ntaimo, L, uk Kim, B. *vGrid: A Framework for Building Autonomic Applications.* In: *Proc. International Workshop on Challenges of Large Applications in Distributed Environments (CLADE'03).* 2003. Seattle, WA, USA. IEEE Computer Society.

[17] Zirintsis, E, Kirby, GNC, Morrison, R. *Hyper-code Revisited: Unifying Program Source, Executable and Data*. In: *Proc. 9th International Workshop on Persistent Object Systems*. 2001. Lillehammer, Norway. Springer-Verlag. pp 232-246.

[18] Warboys, BC, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Morrison, R, Munro, DS. *Collaboration and Composition: Issues for a Second Generation Process Language*. In: *Proc. 7th European Software Engineering Conference (ESEC'99)*. 1999. Toulouse, France. Springer-Verlag. pp 75-91.

[19] Mickan, K, Morrison, R, Kirby, GNC, Balasubramaniam, D, Zirintsis, E. *Using Generative Programming to Visualise Hyper-code in Complex and Dynamic Systems*. In: *Proc. 27th Australasian Computer Science Conference (ACSC2004)*. 2004. Dunedin, New Zealand. Australian Computer Society. pp 377-386.