Support for Feedback and Change in Self-adaptive Systems

Dharini Balasubramaniam, Ron Morrison, Kath Mickan, Graham Kirby
University of St Andrews
St Andrews
Fife KY16 9SS, UK
+44 1334 463253
{dharini, ron, kath, graham}@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
Self-adaptive systems modify their own behaviour in response to stimuli from their operating environments. The major policy considerations for such systems are determining what, when and how adaptations should be carried out. This paper presents mechanisms for feedback and change that support policy decisions for self-adaptation within a computationally complete architecture description language based on the π-calculus. Our contribution is support for feedback through software-encoded probes, gauges and an event distribution network together with support for change through decomposition, reification, reflection, recomposition and hyper-code.

Categories and Subject Descriptors
D.2.11: Software Architectures – *Data abstraction, Domain-specific architectures, Languages*

General Terms
Management, Measurement, Design, Languages.

Keywords
Adaptation, autonomics, composition, constraints, decomposition, feedback, hyper-code, mechanism, policy, probes, recomposition, reflection, reification, self-adaptive systems, software architectures.

1. INTRODUCTION

1.1 Self-adaptive Systems
A self-adaptive system modifies its own behaviour in response to changes in its operating environment [1, 2] with the motivation of prolonging the usefulness of the executing software system.

A common approach for self-adaptation in control systems is to insert a set of probes into an executing system to observe and quantify significant events. Information gathered from the probes is stored in gauges and at an appropriate time is sent to an adaptation engine via an event distribution network. The adaptation engine uses the input from the probes, together with its model of system goals, usually expressed as constraints, to decide when evolution is appropriate.

The major policy considerations for self-adaptation are deciding what, when and how adaptations should be carried out. The mechanisms required to support these policy decisions [3] include facilities for defining constraints, feedback and change.

1.2 Our Approach
Software architectures [4, 5] describe systems in terms of components and their interactions. In order to support and guide self-adaptation within a unified framework, architectures need to specify the behaviour of their components as well as constraints on the structure and cardinality of their components and interactions. We present a software architecture-based approach for self-adaptation where policies may be encoded within languages using mechanisms for supporting constraints, feedback and change.

Constraints pertaining to an application are the conditions that must hold at all times during its execution. We concentrate here on the mechanisms for feedback and change within a computationally complete architecture description language based on typed higher-order polyadic π-calculus. Feedback is supported through software probes, gauges and an event distribution network and change through the adaptation engine using the concepts of decomposition, reification, reflection, recomposition and hyper-code.

2. RELATED WORK
Various frameworks, languages and methodologies have been proposed for the construction of self-adaptive systems.

Containment Units are used to build adaptive systems that deal with anticipated change in [6]. ArchStudio [7] is a tool suite developed to support self-adaptation in the C2 architecture style. Georgiadis et al [8] use specific component managers to identify external architecture changes by listening to events, and then react in order to preserve architecture constraints. Architecture styles, augmented with adaptation operators and repair strategies, are used as the basis for self-repair by Garlan et al [9]. IBM's autonomic computing initiative [10] aims for systems which are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSS'04, Oct 31–Nov 1, 2004 Newport Beach, CA, USA.
Copyright 2004 ACM 1-58113-989-6/04/0010…$5.00.

self-configuring, self-optimising, self-protecting, and self-healing. Dearle et al [11] describe a framework for autonomic management of component-based distributed applications using a constraint solver and an autonomic deployment and management engine. Java [12] provides facilities for observers to be notified when an observable object changes its state.

Some of these approaches are designed for providing self-adaptation to existing systems. Information gathered by inserting a set of probes into an executing system is compared to a separate architectural model, which is kept up to date with the implementation. If constraints in the model are violated then one of a set of adaptation algorithms is chosen to correct the anomaly. These algorithms must have complementary methods in the implementation so that changes can be made [6, 7, 8, 9]. The advantage of our approach is that a single framework provides all these facilities thus making it amenable to an integrated system of checking.

3. MECHANISMS FOR FEEDBACK AND CHANGE

Our facilities for feedback and change are implemented within the ArchWare ADL [13, 14]. We provide a brief overview of some relevant features of this language before describing the mechanisms themselves.

3.1 The ArchWare ADL

The ArchWare ADL is a strongly-typed executable architecture description language designed and implemented as part of the EU-funded ArchWare project [15]. It extends an expression language with typed higher-order polyadic π-calculus [16] and constructs to support composition, decomposition, dynamic evolution and recomposition of systems.

Hyper-code technology [17, 18] is used to support the ArchWare ADL. A hyper-code program is an active executing graph linking source code and existing values. By unifying the concepts of source code, executable code and data, 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. Hyper-code also allows state and shared data to be preserved during evolution. Thus at any point during the computation the state of the execution may be inspected by viewing the hyper-code. In the following examples hyperlinks are shown as underlined identifiers.

Components are modelled by behaviours which communicate via connections using send and receive actions. Behaviours can be collaboratively and hierarchically composed to form a system. Abstractions abstract over behaviours just as functions abstract over expressions.

Figure 1 illustrates some of the above features. Connections *request* and *reply* are defined to communicate integers as messages. The abstraction *server_abs* repeats the following actions whenever a message arrives on connection *request*: it receives a value, implicitly declared as *x*, via *request*, increments a location *count*, defined in the global scope and represented as a hyperlink, and sends double the value of *x* via *reply*. Applying *server_abs*, with any parameters (in this case none) enclosed in brackets, yields an executing behaviour.

```
value request = connection( integer )
value reply = connection( integer )
value server_abs = abstraction()
{   replicate{
        via request receive x
        count := ‘ count + 1
        via reply send 2 * x }
}
server_abs()
```

Figure 1: Components and interactions

A compose operator (akin to "|" in the π-calculus) creates a single handle to a number of executing behaviours. Figure 2 shows how a system may be defined by composition using *server_abs* from Figure 1 and a *client_abs* abstraction defined elsewhere and represented here as a hyperlink. The identifier preceding the *as* keyword allows a unique label to be given to each behaviour. The value *server_2clients* is a composite component with one server and two clients as its constituents.

```
value server_2clients = compose{
                                s as server_abs()
                    and         c1 as client_abs( 256 )
                    and         c2 as client_abs( 400 )
                }
```

Figure 2: Composition

For later use we introduce the ADL's infinite union type, *any*, along with inject and project operations over it. Values of any type can be injected into an *any* and then projected back on to the original type.

For building self-adaptive systems, the ADL supports facilities for specifying constraints, providing feedback and effecting change. Constraints on structure, cardinality and dynamic behaviour of components can be specified in the style layer of the ArchWare ADL [19]. Styles have been described elsewhere and are not included in this paper. In the following sections, we concentrate on feedback and change.

3.2 Feedback

Feedback acts as a trigger for change in self-adaptive 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

We define a uniform mechanism to deal with feedback from all these sources. The feedback mechanism is structured as follows:

- a component which requires feedback (feedback sink)
- a feedback source and its interface
- a software probe defined by the feedback sink
- a probe connection to send the probe from the feedback sink to the source, published as part of the source interface
- a feedback connection to send the feedback from the probe to the feedback sink

Each feedback source is designed to publish its feedback interface consisting of observations available to probes, accept an observation name and function from the probe connection, interpret the function as a probe, bind it internally and invoke it at

the appropriate time. The events of interest are updates to the state of the source, modelled as assignments to locations defined within the source.

Feedback sinks are structured to define a software probe, send it to the source via the appropriate probe connection, receive the feedback via the feedback connection 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. Software probes can perform the duties of both probes and gauges by generating and storing feedback.

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

Within each source a register of received probes and the observations required by them is maintained. After each update all probes registered for that location are executed. A feedback register is used for storing the feedback interfaces of all sources at a well-known place such as a published location in the persistent store.

The advantages of this approach are:

- it provides a uniform mechanism to deal with feedback from different types of sources
- feedback sources do not require knowledge of constraints driving the sinks as this information is encapsulated within probes
- probes are specialised to the requirements of feedback sinks which receive only the feedback relevant to them

3.2.1 Feedback from another Component

Receiving feedback from another component written in the ADL requires no special provision from the language. Feedback sink A may send the name of an observation and a probe to component B via a connection of message type (*string, function[]*). Component B is designed so that it recognises the function as a probe, stores it in its probe register and calls it at the appropriate time. The probe can then generate and send the specified feedback via another connection to the originator if the condition becomes true. Since both components are in the ADL domain, there is no restriction on the type of feedback generated and hence on the type of feedback connection.

3.2.2 Feedback from Execution Engine

Feedback from the execution engine provides information on significant features of system execution, not normally available to programs written in the language. The execution engine implements a published interface of observations which can be used by probes. Two special functions, *execution_probe* and *execution_feedback*, are used to create connections of message types (*string, function[]*) and *any* respectively. The former allow probes to be sent by a feedback sink to the execution engine while the latter enable feedback to be sent to the feedback sink from the execution engine. The engine is designed so that it treats messages from connections defined by *execution_probe* as probes. Since pre-defined connections created by *execution_feedback* are the only means of getting feedback from the execution engine to an ADL component, all feedback messages are typed as *any*.

3.2.3 Feedback from External Environment

Feedback from external environment requires communication with sources outwith the ArchWare ADL domain. The ADL supports I/O connections created by a special function called *stdio* using well-known connection names. A list of well-known connection names is published to all tools and clients wishing to communicate with the ADL system. Messages sent through I/O connections are typed as *string*.

Since communication in this case is with an agent, probes are not sent to external sources. However feedback can still be sent to an ADL component by the external source using an agreed message format.

3.2.4 An Example

The feedback register can be modelled as a list in the ArchWare ADL as shown in Figure 3. Each element of the list consists of a name for the source, a probe connection on which the source will receive probes sent by sinks and the feedback interface for the source. The interface itself is a list containing a descriptive name and value (injected into an *any*) of observable locations defined by the source. We assume the definition of two functions over these lists. Function *generate_interface* takes a source name and a probe connection and creates an entry in the feedback register while *publish_observation* adds an entry to a given interface.

```
recursive type feedback_interface_type is view[
                observation_name : string,
                observation_val : any,
                next : location[feedback_interface_type] ]
recursive type feedback_register_type is view[
                source_name : string,
                probe_conn : connection[ string, function[] ],
                feedback : location[feedback_interface_type],
                next : location[feedback_register_type] ]
```

Figure 3: Definition of feedback register

The probe register within each source may be defined as shown in Figure 4. It is a list containing pairs of observations and probes. Three functions are defined to manipulate probe registers. *generate_probe_register* creates an empty probe register, *add_probe* adds an observation name and a probe to the register and *execute_probe* takes an observation name and executes all probes registered for it.

```
recursive type probe_register_type is view[
                observation_name : string,
                probe : function[],
                next : location[probe_register_type] ]
```

Figure 4: Definition of probe register

Figure 5 shows the definition and use of a software probe by means of a simple example. Consider a system in which a component *monitor_clients* (feedback sink) monitors the number of clients processed by another component *process_clients* (feedback source).

```
value client_conn = connection( string )
value process_clients = abstraction()
{   value pc_probe_conn = connection(string, function[] )
    value pc_interface = generate_interface("process_clients",
                                            pc_probe_conn)
    value pc_probes = generate_probe_register()
    value count = location( 0 )
    publish_observation( pc_interface, "count", any(count) )
    compose{
        b1 as {  replicate{
                    via client_probe_conn receive obs, probe
                    add_probe( pc_probes, obs, probe ) } }
    and
        b2 as {  replicate{
                    via client_conn receive request
                    count := 'count + 1
                    execute_probes("count")
                    process_request( request ) } }
    }
}
value monitor_clients = abstraction()
{   value feedback_conn = connection( integer )
    value client_probe = function()
    {   if ( '( count ) mod 10) = 0 do
        via feedback_conn send 10 }
    via client_probe_conn send "count", client_probe
    replicate{
        via feedback_conn receive client_count
        record_clients( client_count ) }
}
```

Figure 5: Feedback from another component

In Figure 5, connection *client_conn* is used for interacting with clients. Component *process_clients* defines a probe connection *pc_probe_conn* for receiving probes and creates an entry in the feedback register for its feedback interface using a hyperlink to the pre-defined function *generate_interface*. It defines a register for holding the probes it receives. It initialises the location *count* which is used to keep track of the number of clients processed and publishes the name and value of *count* as observable in *pc_interface*. It then initiates two behaviours in parallel using the *compose* construct. For every message received on *client_probe_conn*, the first behaviour updates the probe register using *add_probe*. For each request received on *client_conn* connection, the second behaviour increments *count*, executes all probes associated with *count* and processes the request.

Component *monitor_clients* defines connection *feedback_conn* to be used for feedback and a probe *client_probe* as a function referencing the published *count*. *client_probe* contains a hyperlink to the value *count* from *pc_interface* in the feedback register and specifies that if the value contained in *count* is a multiple of 10 then 10 should be sent on connection *feedback_conn*. Once the probe is defined, *monitor_clients* sends it via *client_probe_conn*, also linked to from the *pc_interface*. Every time a value is received via *feedback_conn* it records the value.

In this simple example the feedback could have been communicated to the monitor without a probe. However in larger and more complex systems, a mechanism which permits component A to plant a probe in component B is useful as it separates the concerns of functionality and feedback.

3.3 Change

Mechanisms for change in the ArchWare ADL have been described elsewhere [13]. We present a summary of these mechanisms here.

We consider the following kinds of change for self-adaptive systems:

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.

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 as shown in Figure 1. If component creation depends on some dynamic input then replication ("!" in the π-calculus) is suitable as illustrated in Figure 5.

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 still executes. This change requires support for decomposition, reification, reflection and recomposition.

Decomposition [14, 20] 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 a 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 [21]. The hyper-code system can be invoked from within ADL by the *edit* function which takes as a parameter and returns as result values of type *any*.

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. The new specification has to be brought into the execution domain by dynamic compilation. A callable compiler, invoked as function *compile*, 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.

4. CONCLUSIONS AND FUTURE WORK

We have presented a software architecture based approach to building self-adaptive systems. The novelty of this approach is the combination of feedback and change mechanisms within a π-calculus based strongly typed executable ADL. We described a generic mechanism for feedback from different sources using software probes and connections. Decomposition, reification, reflection and recomposition form part of the change mechanism.

The ArchWare ADL hyper-code system used in the examples has been implemented. It is currently being evaluated by academic and industrial partners of the ArchWare project.

One of the areas for further research is support for transformations from one hyper-code program to another. Research into programmable interfaces to the ADL hyper-code system [22] for achieving such transformations is ongoing.

A framework for self-adaptation can be provided by combining architecture and language support with a development methodology. We have identified process for process evolution (P2E) [23] as a suitable methodology for developing self-adaptive systems. P2E 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 functionality and change.

Most solutions for self-adaptation deal with a pre-defined set of constraints, triggers and adaptations but do not address issues arising from the need to evolve potentially every part of the system. As business and application requirements evolve, all aspects of a system including constraints, architecture, feedback and change mechanisms may need to evolve to keep in step. A framework combining P2E with the software architecture based approach presented in this paper will enable all these aspects including the framework itself to evolve.

5. REFERENCES

[1] Kephart, J, Chess, DM. *The Vision of Autonomic Computing.* In: IEEE Computer Journal, 2003.

[2] Oreizy, P, Gorlick, MM, Taylor, RN, Heimbigner, D, Johnson, G, Medvidovic, N, Quilic, A, Rosenblum, DS, Wolf, AL. *An Architecture-based Approach to Self-adaptive Software.* In: IEEE Intelligent Systems and Their Applications, 1999.

[3] Morrison, R, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Munro, DS, Warboys, BC. *A Compliant Persistent Architecture.* In: Software - Practice and Experience, Special Issue on Persistent Object Systems, 2000.

[4] Perry, D, Wolf, A. *Foundations for the Study of Software Architecture.* In: ACM SIGSOFT Software Engineering Notes, 1992.

[5] Garlan, D, Shaw, M. *An Introduction to Software Architecture.* In: Advances in Software Engineering and Knowledge Engineering, 1993.

[6] Cobleigh, J, Osterweil, LJ, Wise, A, Lerner, BS. *Containment Units: A Hierarchically Composable Architecture for Adaptive Systems*. In: *Proc.10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 2002. Charleston, SC, USA.

[7] Oreizy, P, Medvidovic, N, Taylor, RN. *Architecture-Based Runtime Software Evolution.* In: IEEE, 1998.

[8] Georgiadis, I, Magee, J, Kramer, J. *Self-Organising Software Architectures for Distributed Systems.* In: *Proc.1st ACM SIGSOFT Workshop on Self-Healing Systems*. 2002. Charleston, SC, USA.

[9] Garlan, D, Cheng, S-W, Schmerl, B. *Increasing System Dependability through Architecture-based Self-repair*. In: *Architecting Dependable Systems*. 2003: Springer-Verlag.

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

[11] Dearle, A, Kirby, GNC, McCarthy, A. *A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications*. 2004. Technical Report, University of St Andrews.

[12] Liskov, B, Guttag, J. *Program Development in Java*. 2001: Addison-Wesley.

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

[14] 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.WICSA 2004*. 2004. Oslo, Norway.

[15] 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.

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

[17] Zirintsis, E. *Towards Simplification of the Software Development Process: The Hyper-code Abstraction*. 2000. PhD Thesis, University of St Andrews.

[18] 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.

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

[20] 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.

[21] 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.

[22] Mickan, K. *Hyper-code for Evolution*. To be completed in 2005. PhD Thesis, University of St Andrews.

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