Meta-adaptation in Autonomic Systems

Jamie Hillman & Ian Warren
Computing Department, Lancaster University, Lancaster, UK
Email: j.hillman | iw @comp.lancs.ac.uk

Abstract

Autonomic computing aims to make self-healing, self-tuning, self-configuring and self-protecting systems a reality. This self-adaptive behaviour is achieved by carrying out changes upon a system when particular constraints or requirements are not met. Existing work focuses on how to determine when a change is necessary and what that change will be. We argue that the mechanism used to carry out the change is equally important and can determine if the change heals, tunes or protects the system as it was intended to. We present OpenRec, a framework which facilitates autonomic computing where the change mechanism is also self-adapting. Novel features of our work include a clean separation of concerns in the self-adaption process, support for an extensible set of change mechanisms and the use of an evolving knowledge base which guides the change process.

1 Motivation

Autonomic computing aims to enable four classes of system; self-healing, self-tuning, self-configuring and self-protecting. We direct the reader to [7] for a comprehensive definition of these terms. For the purpose of this paper the important feature present in each of these classes of system is that they adapt themselves in order to maintain performance in the face of changing operating conditions. The traditional approach to achieving this self-adaptation is a feedback loop where a system's performance is monitored and maintained through carrying out repair upon detection of a constraint or requirement not being met. Carrying out change in order to ensure the conformance of a system to constraints and requirements placed upon it is not a simple process.

The dynamic reconfiguration field has seen the development of algorithms which ensure change to a system is carried out consistently and without damaging the system. These algorithms vary in their applicability and performance however. One algorithm may not be suited to every situation. Existing approaches to developing autonomic systems [1] assume that there exists a safe mechanism for carrying out change and there is one such mechanism suitable for every change. In order to effectively ensure a system performs its function we argue that it is necessary to carry out change using whatever mechanisms will ensure the change is carried out quickly, effectively and with minimal disturbance to the application. This is not likely to be the same mechanism in every situation.

2 Underlying Technologies

2.1 Components

A component is a unit of composition with well-defined provided and required interfaces [16]. Since components are compositional, they are intended to be assembled to form a configuration of components. Similarly to an object, a component may encapsulate state and its provided interfaces define services the component offers to others. Components are distinguished from objects in that they are *configuration independent*, which is defined by:

- *Explicit required interfaces*. In addition to provided interfaces, components define required interfaces which represent the services they require from other components. A component's required interfaces are bound at run-time to other components' provided interfaces.
- *Interconnection independence*. Components do not contain implicit references to other components. Instead, all interaction between components is handled by a connector which binds required interfaces to provided interfaces. Common types of connector include those with messaging, (remote) invocation and publish/subscribe semantics.
- *Conformance to a binary standard*. Components can be separately compiled and composed into a configuration without access to their source code.

Closely related to components is the notion of component frameworks, which Szyperski [16] defines as "collections of rules and interfaces that govern the interaction of a set of components plugged into them." In essence, a component framework is a reusable architecture which provides a means of enforcing architectural properties. Such architectural properties may relate to components, connectors and connectivity. For example, components may or may not be allowed to encapsulate state, connectors may be required to implement particular semantics, and connectivity invariants may be specified. Component frameworks are a similar concept to architectural styles [15] from the software architecture community.

Component technology is also well suited to realising reconfigurable systems. Configuration independence means that components are inherently loosely coupled and any dependencies are explicit and visible which eases the process of changing a configuration at run-time. In general, a component configuration can be manipulated dynamically by adding, removing and replacing components and changing connectivity.

2.2 Reflection

Reflection is the ability for a system to reason about and act upon itself [11]. Reflective systems comprise two levels, a base level and a meta level, that are causally connected. A change in the base layer thus causes an automatic change in the meta level and vice versa. The motivation for reflection, in all areas where it has been applied which include middleware [4] and component models [6], is to promote openness and flexibility.

Of particular relevance to our work is the integration of reflection and components. With reflective component models, the base-level corresponds to application components which are augmented with meta-level interfaces. The meta-level interfaces can be invoked at run-time and offer two complementary forms of reflection, structural and behavioural.

Structural reflection is concerned with exposing the structure of components and connectors that realise a system. Meta-level interfaces provide operations which allow the interfaces a given component provides and requires to be inspected. Beyond inspecting a system's structure, the meta-level interfaces also allow this structure to be modified. *Behavioural reflection* is concerned with system activity, such as component interaction. Interceptors are typically used to listen on connectors for inter-component communications and to take action for purposes such as monitoring or accounting.

Reflective component models bring openness and self-awareness to component-based systems. Using structural reflection, a component's neighbours can be discovered, its interfaces can be identified and other meta information such as whether a particular component encapsulates critical state can be found. In addition, structural reflection offers the necessary primitives for making structural changes to a configuration.

3 The OpenRec Framework

3.1 Overview

The OpenRec framework allows the design of autonomic systems which can be adapted in two dimensions. Firstly the application can be adapted, as in any autonomic system, in order to exhibit self-tuning, self-healing and self-protecting behaviour. Secondly, OpenRec allows the mechanisms and tools that carry out change upon an application to be adapted in order to ensure the effective execution of any necessary change. This is achieved through clean separation of the concerns within the adaptation process, which is discussed further in section 2.2. OpenRec allows the use of many adaptation mechanisms and also allows new mechanisms to be developed using a simple API. This is discussed further in section 2.3. Finally, OpenRec's autonomic functionality is discussed in section 2.4.

3.2 Separation of Concerns

The OpenRec framework, show in figure 1, is divided into three layers; Change Driver, Reconfiguration Manager and Application. The Change Driver contains the functionality necessary to determine when a change is necessary and what that change will be. If the Change Driver decides a change is necessary it issues a description of the change required to the Reconfiguration Manager, expressed in OpenRecML (an XML-based change description language). The Reconfiguration Manager executes the requested change on behalf of the Change Driver by manipulating the application layer through its meta-interfaces. Finally, the application layer contains the components which make up the application and exposes through it's meta-interfaces the means to manipulate the application without concerning the application programmer. The meta-interfaces of the application layer allow connections between applications components to be inspected and manipulated, meta-data to be added to components and calls to be intercepted. These interfaces allow significant change to change the application.

The separation between these three layers of the framework brings several benefits to the design of an autonomic system. First of all the programmer does not have to design the application with autonomic behaviour in mind, the application can be designed as a simple static system. As long as the component model of OpenRec is used in implementing the application the Reconfiguration Manager can affect reconfiguration through the meta-interfaces of the component model. The autonomic functionality is separated from the application layer / component model and so need not concern the application programmer.

Secondly, the mechanisms that carry out change (contained in the Reconfiguration Manager) are separated from both the application and the decision process which determines the need for change (contained within the Change Driver). This means that new adaptation mechanisms can be designed independently of any particular application and independently of the decision process, providing great flexibility. It also means that suitable change mechanisms can be selected for each change the Change Driver requests.

3.3 Adaptation mechanisms

Adaptation mechanisms, as has been mentioned previously, carry out a requested change by manipulating the application layer through its meta-interfaces. The Reconfiguration Manager hosts these Adaptation Mechanisms and provides the Change Manager, through its meta-interfaces, with the means to select mechanisms and introduce new mechanisms. An adaptation mechanism consists of a dynamic reconfiguration algorithm component and a set of change constraint components.

3.3.1 Dynamic Reconfiguration Algorithms

Although reflective component models offer a powerful means for dynamic change, they offer only limited safeguards and do not guarantee that run-time changes will leave systems with their integrity intact. Two fundamental issues concerned with preserving system integrity during periods of dynamic reconfiguration are synchronisation and state management.

Synchronisation is necessary as arbitrary changes to a configuration will likely compromise its integrity. For example, using its meta interface a component might be disconnected from its neighbours. If the disconnection is performed once the component has started to process a request on behalf of one of its neighbours but before it has returned a response, the neighbouring component might wait indefinetely for the response.

State management is also important because components may encapsulate state that must survive reconfiguration. In another scenario involving replacing a component, for example, the replacement component might need access to the state encapsulated by the original component. This is invariably the case for any component whose state is subject to change since changes can influence its future behaviour. A simple example is a component which generates unique identifiers; where this component is to be replaced, the replacement component relies on knowledge of which identifiers have already been generated in order to continue to create unique identifiers. Naive use of meta interfaces could easily result in such state being discarded at run-time, with the consequence that system integrity is compromised.

Dynamic reconfiguration is concerned with bringing order to the change process with the aim of maintaining a system's integrity. In general, this is achieved using an algorithm which examines a set of proposed configuration changes and identifies a subset of of the configuration to direct towards a safe state where change is deemed synchronised with the executing system. The safe state may be reached when a component involved in reconfiguration is prevented from processing incoming requests and from initiating further requests on others. Having reached the safe state, an algorithm typically effects configuration changes and may take action to preserve persistent state before allowing the configuration of components to resume normal execution.

Reconfiguration algorithms differ in terms of behaviour and applicability constraints. To illustrate behavioural differences, algorithms can be classified according to two dimensions; their approach to reaching the safe state and the unit of disturbance during change.

Static algorithms, for example [12] [17], use knowledge of the configuration structure to identify components to make safe. As such, static algorithms will always identify the same set of components for a particular system reconfiguration. In contrast, dynamic algorithms, including [14] [5] [8], use run-time information such as component interaction to determine the set of components to make safe. In many cases, dynamic algorithms disrupt less of a system than static algorithms since they make safe only those components that are *actually* communicating with a component involved in change. Static algorithms, on the other hand, make safe the set of components that might *potentially* communicate with a component involved in reconfiguration.

Algorithms differ in terms of their unit of disturbance, which is either the component or connector. Algorithms of the former type, including [12] [8], are relatively coarse-grained since they halt components, thereby preventing them from executing further. Connector-blocking algorithms, such as [17] [18], allow components to continue to execute, albeit with degraded functionality using their unblocked connectors. Algorithms that block connectors therefore impact less on an executing system.

Common constraints on algorithms include prescribed communication paradigms, the need for components to implement change management interfaces, and prerequisite knowledge of component properties. For example, a number of algorithms have been developed for distributed objects[1] ([2] [14] for CORBA and [5] for Java RMI). These algorithms therefore assume method invocation as the com-

[1] Despite being developed for distributed object systems, these algorithms can be used to manage reconfiguration of components where connectors implement method invocation semantics.

munication paradigm. Kramer and Magee's algorithm [12] is less restrictive in that it allows components to interact using transactions, where a transaction is defined as one or more message exchanges between two or more components. However, Kramer and Magee's algorithm requires that components implement a change management interface correctly in order for the safe state to be reached.

To illustrate prerequisite knowledge, Chen's algorithm [5] distinguishes between mutator and selector operations by allowing mutator requests to complete execution and aborting selector requests. The rationale in this case is to force a component to reach the safe state more quickly allowing only state modifying operations to complete. Mutator operations must be allowed to complete in the interest of component integrity. Once the component has completed executing any mutator requests, it is deemed to have synchronized. To implement this behaviour, the algorithm needs knowledge of which of a component's operations are mutators and which are selectors.

The above discussion exposes some fundamental differences in reconfiguration algorithms found in the literature. We argue that given such differences, algorithms will vary in terms of the costs they impose on an executing system. In addition, developers need to be aware of the constraints governing applicability of any algorithm. Hence, a framework which allows multiple algorithms to be modelled, implemented, and compared is highly desirable in determining the most appropriate algorithm to use, given a particular application.

3.3.2 Change Constraint Components

Change constraint components provide a higher-level control to the change process. As advocated in [3], change can be constrained in architectural-style or domain specific ways. We believe OpenRec to be unique in that it allows control of change at the architectural-level and the lower-level controlled by reconfiguration algorithms. We also believe it is advantageous to be able to combine this two types of change constraint arbitrarily. The change constraint components are informed of each change directive that is requested and can accept or reject a change directive as it deems fit. This open implementation allows any set of constraints to be imposed upon a requested change.

3.4 The Autonomic Change Driver

3.4.1 Overview

OpenRec's change driver layer requests a change to be executed by the Reconfiguration Manager, upon the application. This change driver can take any form but here we will discuss a particular instance of the change driver, the autonomic change driver. The autonomic change driver monitors a system and decides whether a change is necessary or not. If change is necessary then it makes a second decision to determine what mechanism should be used to carry out this change most effectively. The autonomic change driver itself is a framework for building autonomic behaviour and consists of three components; context, strategy and decision, which are described in the following three sections.

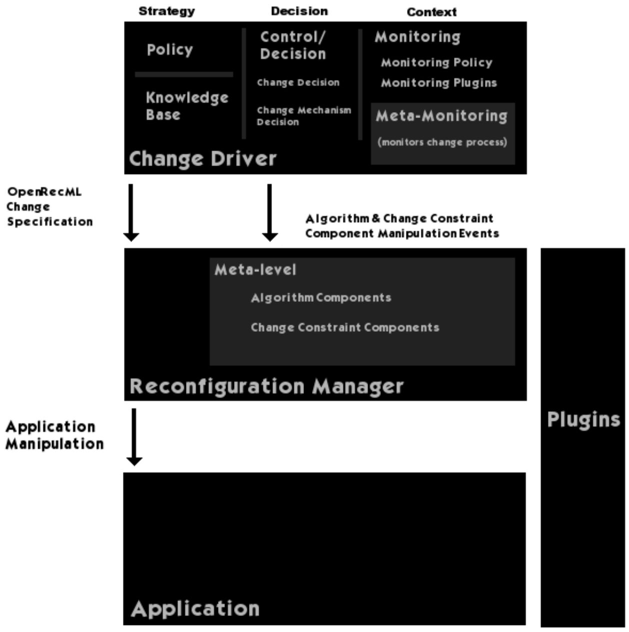

Figure 1. Framework architecture

3.4.2 Context

This part of the change driver represents the context for a particular decision. It consists of a collection of monitoring plug-ins which monitor parts of the application through the plug-in interfaces of the application layer. These plug-in interfaces are designed with extensibility in mind and make it easy for a monitoring plug-in to capture information about application behaviour.

Monitoring plug-ins may also monitor the system via remote tools to get a more realistic perspective from which to gauge performance. For example, the throughput of the system may be more accurately monitored from a remote host which accesses the system via the network than by a local monitoring plug-in which does not incur the overhead of the network and so may report the service as performing better than clients perceive.

3.4.3 Strategy

The strategy component represents how the autonomic change driver should make decisions in a particular context. It consists of a policy and a knowledge base.

Policy The policy determines what changes are necessary if a given metric that is monitored falls outside of an acceptable range. The policy component can be implemented to conform with industry standards, for example it could take the form of a service level agreement expressed in WSLA[].

Knowledge Base The knowledge base indicates which change mechanisms are suited to a given change, based on priorities expressed within the policy. For example the SLA can be annotated to specify that speed of change is more important than disturbance to the application for a given change. The knowledge base contains a set of metrics and the relative merits of each algorithm with respect to those metrics. If it has been specified that speed of change is most important for a particular change then an algorithm which carried out change quickly, such as [13], mentioned earlier, would be suggested by the knowledge base. Or where limitation of disturbance is a priority then algorithms such as [12] may be avoided in favour of others.

3.4.4 Decision

The decision component periodically looks at the information provided by the monitoring plug-ins and checks the policy/SLA for any constraints which are not satisfied. In this event it will then look at the change specified to rectify the unsatisfied constraint. If the change description is annotated with priorities for the change process then the knowledge base will be consulted to find a suitable algorithm. If not a standard general-purpose algorithm will be used. Once the reconfiguration manager has been configured with the appropriate change mechanism, a description of the necessary change will be sent to the reconfiguration manager for execution.

3.4.5 Evolution

The knowledge base indicates the relative strengths of different algorithms with respect to given metrics. Initially the data in the knowledge base is created through observation of the algorithms. The knowledge base is not limited to this observation-based input however. The context component performs meta-monitoring in addition to the monitoring described earlier. Meta-monitoring monitors the process of reconfiguration through the plug-in interfaces of the reconfiguration manager. Meta-monitoring plug-ins exist for each of the metrics in the knowledge base and monitor the change process in order to assess the algorithm in terms of that specific metric. Beyond conforming to a simple meta-monitoring interface, the implementation of these plug-ins is fairly open. Existing protocols and programming libraries can be used to ease the monitoring process. Information from these plug-ins is then recorded in the knowledge base to inform future decision processes. As the the constituent components of the change driver, and the change driver itself, are replaceable, the process of meta-adaptation is itself adaptable. This component-replacement approach was favoured over further layers of reflection in order to maintain simplicity and avoid infinite towers of reflection.

4 Status

4.1 Overview

The OpenRec framework has been implemented in Python and several example applications have been developed and subjected to reconfiguration. Several reconfiguration algorithms have also been developed, in addition to several statistical analysis tools which observe the behaviour of these algorithms. The autonomic change driver framework has also been implemented and tested using a basic autonomic system. Due to space restrictions the details of the implementation cannot be discussed here. We refer the interested reader to [9] and [10] for more technical detail.

This section will describe the development of an autonomic system using OpenRec and the autonomic change driver. The autonomic system is a simulation of an application operating on a mobile device. As the mobile device moves to different networks, the throughput and reliability of the network varies and the application must be reconfigured to continue performing adequately. The user of the application should not be concerned with the details of the networks he is using but requires the application to gracefully adjust itself to cope with new conditions.

The change considered in this example involves replacing a communications component when the failure rate of transactions exceeds a specified threshold. This may be due to moving to an unreliable network for example. Failed or timed-out requests are reported to a logging component. If the number of errors reported to this component exceeds a critical threshold then the communications component is replaced with a more robust implementation which, although it introduces extra overhead, copes with an unreliable network more competently.

The following subsections describe this application in more detail with respect to the three core functions of the autonomic change driver; context, strategy and decision.

4.2 Context

Monitoring plug-ins in the application monitor the number of transaction per second at the error logging component. This is achieved through registering interceptor plug-ins in the application using the meta-interfaces of the component model.

4.3 Strategy

4.3.1 Policy

The service level agreement held by the change driver specifies that the number of failed transmission errors reported to the logging component should be less than 4 per second. It also specifies that if this constraint is violated then the ordinary communications component should be replaced with the implementation that handles an unreliable network connection. Finally it specifies that this change should be carried out with a preference to limit disturbance to application components.

4.3.2 Knowledge Base

The knowledge base contains details of several algorithms and stores, along with a pointer to the algorithm, the relative strengths of the algorithm in terms of several key factors. At the moment these key factors are limited but more factors are being considered. So far the knowledge base rates algorithms on the following factors:

Speed of Change The total time taken for the reconfiguration process.

Disturbance to application The percentage of application components directly disturbed in any way by the reconfiguration process.

Application Contribution Whether or not the algorithm requires the application to implement certain interfaces for reconfiguration.

Architectural Style An architectural style/styles for which the algorithm has been optimised and for which the above value apply.

For the mobile device scenario there is an algorithm optimised for the pipe and filter architectural style which the system conforms to. The knowledge base chooses this algorithm as it exhibits low disturbance to the application.

4.4 Decision

The decision process first examines the SLA and steps through each of the assertions it contains. For each assertion it checks with monitoring plug-ins to determine if they are true. Once the decision process reaches the error rate assertion described earlier and discovers from monitoring that this assertion is false, it begins to decide upon a remedy. The change specified in the SLA is parsed and the priorities for the change process are used to select an algorithm from the knowledge base. The reconfiguration manager is configured to use this algorithm through its meta-interfaces. The change description is then generated in OpenRecML and the reconfiguration manager is sent this change to execute.

Reconfiguration Times		
Case	*Optimised*	*General Purpose*
Best Case	9.1ms	52.3m
Worst Case	56.3ms	103.2ms

Figure 2. Reconfiguration times for optimised and general purpose algorithms

4.5 Results

Initially this scenario was executed with only one general purpose algorithm in the knowledge base. The error rate was increased by artificially introducing failures into the network. Performance of the reconfiguration process was recorded using OpenRec's statistics plug-ins.

The algorithm optimised for the pipe and filter architectural style was then added to the knowledge base and the scenario was re-run. This time the optimised algorithm was used to carry out the change and the results of this process were also captured. Figure 2 shows the results of reconfiguration using these two algorithms.

5 Conclusions

We have argued that autonomic systems should have the ability not only to adapt the system in order to meet requirements and performance goals but to adapt the adaptation process itself in order to ensure that changes are carried out effectively. The OpenRec framework was introduced, which allows autonomic systems to be developed with a clean separation of concerns. Mechanisms used for carrying out change, which consist of reconfiguration algorithms and change constraint components, can be considered independently of both the application and the decision process which drives change. The autonomic change driver was introduced, which makes decisions as to what change is necessary and what mechanism should be used to carry out that change. This decision is made in terms of the current context, a policy or SLA and a knowledge base, which evolves as it observes the performance of algorithms in practice.

This framework enables our goal of adaptation in the adaption process itself; meta-adaptation. We believe this added dimension of flexibility is essential in systems which are to adapt themselves in the face of changing operating conditions, whether this is in the context of self-healing, self-tuning, self-configuring or self-protecting systems. Fixed change mechanisms will not perform adequately in every situation and so may further hinder the functioning of the system instead of improving it as they should.

References

[1] J. Appavoo and et al. Enabling autonomic behavior in systems software with hot swapping.

[2] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th International Conference on Configurable Distributed Systems, pages 35–42. IEEE Computer Society Press*, 1998.

[3] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware*, pages 164–184, 2000.

[4] G. S. Blair, G. Coulson, A. Anderson, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. In *Distributed Systems Online, Vol. 2. No. 6*, 2001.

[5] X. Chen and M. Simmons. Extending RMI to support dynamic reconfiguration of distributed systems. In *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[6] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[7] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. In *IBM Systems Journal, Automonic Computing. Vol 42. No 1.*, 2003.

[8] K. M. Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, London, 1999.

[9] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *International Conference on Software Engineering (ICSE)*, 2004.

[10] J. Hillman and I. Warren. Quantative analysis of dynamic reconfiguration algorithms. In *Design, Analysis and Simulation of Distributed Systems (DASD)*, 2004.

[11] G. Kiczales, J. Rivieres, and D. Dobrow. The art of the metaobject protocol. In *MIT Press*, 1991.

[12] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[13] S. R. Mitchell. *Dynamic Configuration of Distributed Multimedia Components*. PhD thesis, University of London, 2000.

[14] J. Paula, A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, 2001.

[15] M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. *Prentice-Hall, Englewood Cliffs, NJ*, 1996.

[16] C. Szyperski. Component software: Beyond object-oriented programming. *Addison Wesley*, 1998.

[17] I. Warren. *A Model for Dynamic Configuration which Preserves Application Integrity*. PhD thesis, Lancaster University, 2000.

[18] M. Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97)*, 1997.