A Deliberative Model for Self-Adaptation Middleware Using Architectural Dependency

N. Badr, A. Taleb-Bendiab, M. Randles, D. Reilly
School of Computing and Mathematical Science,
Liverpool John Moores University,
Byrom Street, Liverpool
L3 3AF, UK
{cmsnbadr, a.talebbendiab, cmsmrand, d.reilly}@livjm.ac.uk

***Abstract*: A crucial prerequisite to externalized adaptation is an understanding of how components are interconnected, or more particularly how and why they depend on one another. Such dependencies can be used to provide an architectural model, which provides a reference point for externalized adaptation. In this paper, it is described how dependencies are used as a basis to systems' self-understanding and subsequent architectural reconfigurations.**

The approach is based on the combination of: instrumentation services, a dependency meta-model and a system controller. In particular, the latter uses self-healing repair rules (or conflict resolution strategies), based on an Extensible Beliefs, Desires and Intention (EBDI) model, to reflect reconfiguration changes back to a target application under examination.

I. INTRODUCTION

The term *autonomic computing* was "coined" by IBM to draw an analogy with the autonomic nervous system: "the autonomic nervous system frees our conscious brain from having to deal with vital, but lower-level functions" [1]. The concept of an autonomic computing system is one that "knows itself" to such an extent that it is capable of self-diagnosis and self-healing whenever internal problems and/or external disturbances are encountered.

The realization of a system that "knows itself" is no mean feat and stretches our existing computing and software paradigms to the limit. For this reason, techniques to self-understanding have drawn inspiration from other fields, such as control engineering [2] and biology [3].

Through this paper, a system is developed that is capable of imparting *externalized* reconfiguration in the event of runtime conflicts. The term externalized is used in the same context as in [4] to refer to reconfigurations handled outside the application, which is under examination. The target application is monitored from outside a given application. Thus, adhering to the emerging separation of concerns principle, a target application is monitored, from outside itself by meta-level middleware services. These provide self-management utilities to monitor the system's states and act upon the observed, or discovered, situations of concern (adaptation triggers) such as failure or system tuning event notification requiring reconfiguration actions. The approach is based on the combination of: instrumentation services, a dependency meta-model and a system controller. The dependency meta-model is determined and maintained by probe instrumentation services. The model is then used by the system controller to carry out externalized reconfigurations, which are reflected back onto the actual application, via the probes. The controller operates in a feedback regime to at best rule out, or at least minimize the effects of runtime conflicts.

The paper is structured as follows: Section II, provides the background material. Section III, describes the concept of dynamic dependencies. Section IV describes the logical architecture of the adaptation system. Section V describes a case study and, finally, Section VI draws conclusions and mentions future work.

II. BACKGROUND

Previous research has focused on the use of conventional engineering principles to assist the understanding and management of distributed systems [16]. In particular, research has been undertaken into the use of software instrumentation and controller concepts to monitor runtime behaviour and stabilize "misbehaving" applications respectively [23]. The choice of such conventional engineering principles stems from their pragmatic yet sound mathematical basis.

Some researchers believe that the key to self-understanding lies in the maintenance of architectural models [4, 5, 6]. Through such architectural approaches, models of what the system is currently doing can be compared against models of what the system should be doing. Discrepancies resulting from the comparisons may then be used as the basis for internalized or externalized reconfiguration. At the heart of such approaches are architectural models, which are developed using Architectural Description Languages (ADLs) such as Acme [7] and XADL [8], which describe software architecture in terms of components and connectors.

Whilst the use of architectural models may be advocated, as the basis of self-understanding and self-adaptation, it is in the choice of basic elements, where this approach differs to those of other researchers [5-7]. It is preferable to describe architectures in terms of service providing components and the dependencies that components have on one another. Essentially, this provides a higher level of abstraction above

the component connector model in terms of: components, services and service dependencies. Also where components provide and use services and a dependency exists when one component uses or relies upon a service provided by another.

The concept of service dependencies is not new, as it has been used before in database systems [9] and network management [10]. Other research work has additionally used dependencies to further an understanding of component-based distributed applications [11, 12].

(a)   Conceptual representation of services dependencies

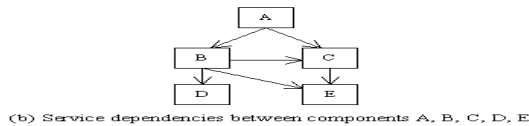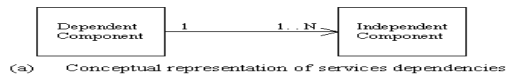(b) Service dependencies between components A, B, C, D, E

Figure 1: Dynamic Dependency Representation

III. AN ARCHITECTURAL MODEL FOR SELF-UNDERSTANDING

As illustrated in *figure 1* a dependency can be regarded as a directed relationship between two components in the sense that one component (the dependent) depends on the service provided by another component (the independent). A dependency is a dynamic artefact, which has a lifetime and it ceases to exist when the dependent component no longer relies on the independent component's service.

A. *Dependency Probes*

In previous research a dependency probe instrument has been developed [11]. A dependency probe may be dynamically deployed and attached to a component to determine its complete dependency graph. By "complete", it is meant that all dependencies (primary, secondary, tertiary, etc.), which could result in the component being deprived of a particular service, are plotted on the graph.

To do so, the probe uses a visitor design pattern [13] to traverse the component graph and calculate, for each component, its current binding. However, the use of dependency probes does require a compromise that developers must include an administration object within their application components, which is used to access the component's bindings.

The current prototype implementation, used here, utilises Jini middleware [14] and more particularly Jini's `Administrable` interface. The `Administrable` interface allows application developers to attach service-specific information to their services so that the information may be accessed and even changed by any client or any other services within a Jini federation. Such information may be retrieved, as an administration object, by invoking the method `getAdmin`.

Also, as shown below, a `Dependable` interface and a `ServiceAdmin` object are used to represent dependencies.

```
public interface Dependable {
      public Class getDeclaringClass();
      public Object[] getBindings();      }
```
A `ServiceAdmin` object implements this interface.

```
    public class ServiceAdmin implements
                Dependable {
      public Object obj = null;
      public Object[] bindings = null;
      public ServiceAdmin(Object obj) {
         this.obj = obj;            }
      public Class getDeclaringClass() {
         return obj.getClass();           }
      public Object[] getBindings() {
         return bindings;     }      }
```

Essentially the compromise requires that any application-level component, which is likely to be dependent on services provided by other components, must include a `ServiceAdmin` object. The component must also implement the `Administrable` interface and therefore the `getAdmin` method, which returns the `ServiceAdmin` object. Probe instruments may then recursively access consecutive `ServiceAdmin` objects that they encounter, as the visitor design pattern executes, and build an in-memory digraph as a series of nodes, representing services, and edges, representing the dependencies, which provides the dependency snapshot for a particular component.

Of course, to maintain the current dependency snapshot a probe must also be aware of any changes in a component's dependencies. To do so, probes register to receive event notifications from each Jini lookup service to which the application-level component holds a reference.

B. *Reconfiguration Based on Dependency Digraphs*

A self-adaptive system controller prototype has been developed, which combines the probes together with method invocation monitor and remote event monitor instruments. The method invocation monitors are capable of intervening in method invocations and repackaging the invocations as new meta-objects. In a similar fashion, event monitors register to receive application-level events, which are repackaged as new meta-objects.

The system controller fuses data from the monitors and maintains an in-memory representation of the application's digraph. By examining meta-objects, the controller can detect runtime conflicts. The simplest form of conflict is a Java exception, whereas a more complex conflict occurs when a component enters a fail-stop state. When conflicts are detected the controller can examine any portion of the applications digraph, or indeed the whole digraph in order to formulate an appropriate course of action.

Such remedial action takes the form of a conflict resolution strategy. In such a strategy the controller typically instantiates an appropriate repair plan (series of actions) intended to eliminate or reduce the conflict leading to the application's reconfiguration. Each self-healing session is supervised by an associated controller service, which is implemented following

the Extensible Beliefs, Desires and Intention (EBDI) model [15-19], considered further in Section IV. Repair strategies are represented as plans and encoded in XML.

Throughout the enactment of a given self-healing plan each action (move) of a conflict resolution strategy is first applied to the application's digraph. If the move proves successful (i.e. no further conflicts occur) the move is then reflected onto the actual application via the dependency probe instruments. This in turn leads to changes in the application's dependency digraph, which must be redrawn to give a faithful "up to date" representation of the application.

Using this approach the dependency model is used like an "electronic bread-board" to try out and experiment with various circuit designs. When a successful design is reached it is then reflected onto a "production quality" circuit. So far only relatively simple conflict/resolution pairings have been considered, such as fail-stop and hot swapping. In future work, it is intended to address more substantial conflict/resolution pairs.

IV. SYSTEM ARCHITECTURE

As shown in Figure 2, the logical architecture is based on a three-layered model namely:

- Application Layer: which contains the application components that provide a federation of application services. Figure 2, shows how the application may be reconfigured by the adaptation layer. We see how the application cloud on the left has been modified to that on the right, where a new server S4 has been introduced to counteract a fail-stop condition in S3.
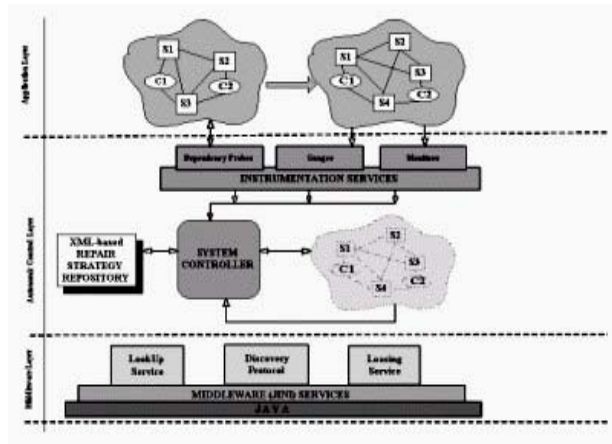

Figure 2. Logical Architecture of the Adaptation System.

- Adaptation Layer: which contains the meta-representation of the application's component dependencies in the form of a dependency digraph. The adaptation layer also contains: method and event monitor services, gauge services, dependency probes and the system controller.

- Middleware core services Layer: which contains a registration service, discovery service and space service. The middleware control service can establish the required communication via its registration service, as the application services register with a middleware service locator.

A. System Repair Strategies

Repair strategies are an essential aspect of autonomic computing, determining *when*, *where* and *how* the repair, and any subsequent adaptation, is applied. This approach to repair strategies relies on a dynamic solution treatment, which uses XML documents. A lightweight system interpreter is used to translate the XML based description of resolution strategies into executable operations. The interpreter is used to map and bind the XML-encoded "tags" of a repair strategy to Java method calls. This process is implemented using the Java reflection API, which allows the Java methods to be invoked based on their XML encoded counterparts.

The resolution strategies are used to represent the effect of various alternative resolutions. Initially, strategies were to be based on the BDI model, which represents the Beliefs, Desires and Intentions. However, the BDI model lacks support for policy and norm constructs, which are central to the developed deliberative autonomic self-healing approach. Various extensions to the original BDI model were proposed to address some of its documented weaknesses, including a normative model, expounded in EDA [20] and BOID [21]. However, these extensions do not address the usage of the norms and policies for optimizing its selected repair strategies (i.e. intentions).

The development of system repair strategies was based on the proposed Extensible Beliefs, Desires and Intention (EBDI) model [15]. The EBDI model is a policy-based model that provides a highly suitable architecture for the design of situated intentional software that continuously monitors and/or observes its environment and acts to change in accordance with its situated BDI.

- Beliefs, which correspond to service information derived and/or accessed from a range of sources, including domain, environment or beliefs of other services.

- Desires, which represent the state of affairs (i.e. in an ideal world), which often maximize the services' own goals. By comparing a system beliefs set (observed system states) against its desires the system may detect a mismatch and trigger intentions (instantiate a set of intentions) [22].

- Situated intentions, which represent action sets for the system to undertake in a given situation to achieve its specified desires and/or to address the mismatch between the system environment (beliefs) and the system's desires (goals).

- Normative intention, which represents a set of actions to be undertaken to ensure a specified set of policies.

The policies, which may include obligation and responsibility rules, are observed before a given intention is enacted.

- Utility intention, which represents a set of system actions examined by the policies to optimize its goal-oriented intentions, such as cost or QoS.

B. System Reconfiguration

To achieve "optimal" reconfiguration the controller takes into account a number of considerations to avoid any further downstream inconsistencies. These inconsistencies may include performance degradation and/or failures. Adopting a similar approach to [23], a set of reconfiguration operators is defined to perform an application's architectural reconfiguration. These operators are encoded in external XML-based strategies. Typical operators are: `getClient()`, and `notifyClient()`.

V CASE STUDY

A prototype of the control system has been demonstrated through an industrial case-study, namely EmergeITS [24]. EmergeITS (Fig. 3) is a software simulation that was developed using Jini middleware technology to demonstrate the concept of intelligent networked vehicles. EmergeITS was developed and prototyped in collaboration with the Merseyside Emergency Fire Service.
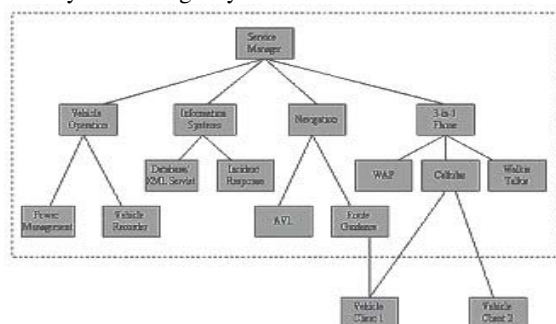

Figure 3: EmergeITS Architecture

One crucial service of EmergeITS is the *3-in-1* Phone service, which allows a mobile phone or PDA device to be used in one of three different modes: a cellular phone, a WAP phone or a walkie-talkie. The *3-in-1* phone service may be used for either voice communication or to receive multimedia content, subject to the requirements of the user and availability of a communication service provider. A prototype of the system controller and its associated probe instruments was incorporated into the case study to provide the control software to support EmergeITS over the network.

The *3-in-1* phone service is hosted by an in-vehicle application server, which also provides a gateway service. The service manager is used to deploy application components along with invocation monitor and event monitor services, which repackage method invocations and events sent and received as meta-objects. In the event of conflict, the controller will select an appropriate repair strategy, thereby providing a degree of conflict resolution and fault-tolerance.

The sequence of actions, of the controller, proceeds as follows:

- The system controller monitors the method invocations made by clients, such as `connect()`, `disconnect()`, and `send()`, `receive()`. If an invocation should fail the controller will try to repair the failure using its repair operators, such as `notify()`, `add_Client()`, etc.
- The controller uses two main criteria for the selection of the appropriate operator for a specific conflict. The first criterion is the satisfaction of the application-level component's attributes and parameters. The second criterion is the execution of the operator without exceptions, whilst also accounting for the runtime system's current structure and behaviour.

A repair operator is : `remove_Client()`, which may be invoked by the controller to remove or disconnect a client who has a low priority and has been connected for more than a predefined time. In such a case, the client would be removed from the list of clients that are using the service as soon as the service's rules are checked. The rule to enforce this is:

service.num_connections > *service.max_connections*

where *num_connections* is the current number of connected clients and *max_connections* is the maximum number of connected clients. If further exceptional behaviour prevails, the system controller will trigger additional conflict resolution moves and subsequent reconfigurations as appropriate. For example, in the *3-in-1* phone the invocation of the `connect()` method on a GSM service, may result in a `RemoteConnectionException` due to the unavailability of a GSM service. This exception is then checked by the controller, resulting in the activation of a suitable resolution strategy based on the EBDI model. For instance, the repair strategy may first attempt a specified number of connection retries. If the retries are unsuccessful, the strategy then searches for an alternative GSM service provider or connects to another suitable alternate service such as WAP. Figure 4, shows an example of an EBDI-based conflict repair strategy encoded as an XML document.

VI CONCLUSIONS AND FUTURE WORK

In this paper an approach to externalized reconfiguration based on the combination of: instrumentation services, a dependency meta-model and a system controller has been described. It has been shown how the dependency meta-model is used to maintain a faithful architectural model of a distributed computing application. It is believed that such a model can be used to assist the understanding of an application in terms of its structure and behaviour and serve as the basis for externalized architectural reconfigurations.

The operations of the controller, which combines diagnostic and EBDI-based resolution strategies, to minimize

or rule out runtime conflicts, have been summarized. The use of the controller has been demonstrated for managing a *3-in-1* phone service and dealing with simple conflicts that may arise.

In future work it is intended to further develop the use of conflict resolution strategies for dealing with more complex conflicts. Additionally conflicts relating to security, such as authorization and authentication conflicts will be addressed.

Figure 4: An XML Conflict Repair Strategy

REFERENCES

[1] A.G. Ganek and T.A. Corbi, "The Dawning of the Autonomic Computing Era", IBM Systems Journal, Vol. 42, No. 1, 2003.

[2] S. George, D. Evans and L. Davidson, "A Biologically Inspired Programming Model for Self-Healing Systems", in Proceedings of First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, USA, 2002.

[3] Kokar, K. B., and Eracar, Y., "Control Theory-Based Foundation of Self-Controlling Software." IEEE Intelligent Systems: 37-45, 1999.

[4] D. Garlan, B. Schmerl and J. Chang, "Using Gauges for Architecture-Based Monitoring and Adaptation", Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

[5] D. Garlan, B. Schmerl, "Model-Based Adaptation for Self-Healing Systems", in Proceedings of First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, USA, 2002.

[6] S.W. Cheng, D. Garlan, B. Schmerl, J. Pedro Sousa, B. Spitznagel, P. Steenkiste, and N. Hu, "Software Architecture-Based Adaptation for Pervasive Systems", in Lecture Notes in Computer Science Volume 2299, H. Schmeck, T. Ungerer, L. Wolf (Eds).

[7] D. Garlan, R.T. Monroe and D. Wile, "Acme: An Architecture Description Interchange Language", in Proceedings of CASCON '97, November 1997.

[8] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages", in Proceedings of the 24th International Conference on Software Engineering (ICSE2002), Orlando, Florida.

[9] H.C. Smith, "Database design: composing fully normalized tables from a rigorous dependency diagram", Communications of the ACM Vol. 28, Issue 8, pp. 826-838, August 1985.

[10] G. Kar, A. Kar and S. Calo, "Managing Application Services Over Service Provider Networks: Architecture and Dependency Analysis", in Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS 2000), Honolulu, HI, USA, April 2000.

[11] P.Hasselmeyer, "Managing Dynamic Service Dependencies", Twelfth International Workshop on Distributed Systems: Operations and Management (DSCOM 2001), ISBN 0-8194-4245-3, Nancy, France, October 2001.

[12] H. Cervantes and R.S. Hall, "Automating Service Dependency Management in A Service-Oriented Component Model", in Proceedings of Sixth ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, Portland, Oregon, USA, May 2003.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides and G. Booch, "Design Patterns", *Addison-Wesley Publishing Company,* ISBN: 0-201-63361-2, 1995.

[14] Sun Microsystems Inc. "Jini Architecture Specification v.1.2, 2001.

[15] N. Badr, "An Investigation into Autonomic Middleware Control Service to Support Distributed Self-Adaptive Software", PhD Thesis, School of Computing and Mathematical Sciences, Liverpool John Moores University, Liverpool, 2003.

[16] N. Badr, D. Reilly and A. Taleb-Bendiab. "A Conflict Resolution Control Architecture for Self-Adaptive", in Proceeding of Proceedings of International Workshop on Architecting Dependable Systems WADS 2002 (ICSE 2002). May 2002. Orlando, Florida.

[17] M. Allen, N. Badr, E. Grishikashvili, and A. Taleb-Bendiab. "Adaptation Engine: an Agent-Based Framework for ad-hoc Service Life-Cycle Management for Meta-Computing". in Proceeding of Processing of AISP symposium on (IA and Grid Computing). 2002. Imperial college London.

[18] E. Grishikashvili, N. Badr, D. Reilly, M. Allen, M.Yu, and A.Taleb-Bendiab. "Autonomic computing: A Service-Oriented Framework to Support the Development and Management of Distributed Applications". in Proceeding of 3rd Annual Postgraduate Symposium on The Convergence of Telecommunications, Networking & Broadcasting (PGNet2002). 2002. U.K.

[19] E. Grishikashvili, N. Badr, D. Reilly, and A. Taleb-Bendiab, "From Component-Based to Service-Based Distributed Applications Assembly and Management". in Proceeding of Proceedings 29th EuroMicro Conference. 2003. Turkey.

[20] J. Filipe, "A Normative and Intentional Agent Model for Organization Modelling", in Third International Workshop on Engineering Societies in the Agents World, Madrid, Spain, 2002.

[21] "Home of the BOID", http://boid.info/, accessed December.2003.

[22] M. Bratman, "Intentions, Plans and Practical Reason", Harvard University Press, 1987.

[23] D. Garlan and R. Stratton, "Rainbow: Architecture-Based Adaptation of Complex Systems "http://www.cs.cmu.edu/~able/rainbow/ accessed 2002.

[24] D.Reilly and A.Taleb-Bendiab. "A Service Based Architecture for In-Vehicle Telematics Systems". in Proceeding of IEEE Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002). 2002. Vienna Austria