Self-Organising Software Architectures for Distributed Systems

Ioannis Georgiadis, Jeff Magee and Jeff Kramer
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK
{jnm,i.georgiadis,jk}@doc.ic.ac.uk

ABSTRACT
A self-organising software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification. The objective is to minimise the degree of explicit management necessary for construction and subsequent evolution whilst preserving the architectural properties implied by its specification. This paper examines the feasibility of using architectural constraints as the basis for the specification, design and implementation of self-organising architectures for distributed systems. Although we focus on organising the structure of systems, we show how component state can influence re-configuration via interface attributes.

Keywords
Software Architecture, self-configuring, constraints.

1 INTRODUCTION
Software Architecture [21; 23] describes the high-level structure of a system in terms of components and component interactions. In design, architecture is widely recognized as providing a beneficial separation of concerns between the gross system behaviour of interacting components and that of its constituent components. Similarly this separation is also beneficial when considering deployed systems and evolution as it allows us to focus on change at the component level rather than on some finer grain.

For instance, previous work described some of the issues involved in specifying a limited form of dynamic software structure for distributed systems in which the set of components and their interaction change as execution progresses and the system evolves [16]. A change to the software architecture could occur either as the result of some computation performed by the system or as a result of some external management action such as to insert a new component and to change those connections within the system to accommodate the new component. Management actions are performed by a configuration manager[4] which maintains an overall view of the structure of a system in terms of components and their interconnections and performs changes in the context of that view. In essence, the configuration manager is responsible for ensuring that an executing system conforms precisely to its architectural specification. This approach can however be too restrictive for current dynamic, open systems.

In this paper we consider systems in which it is neither necessary nor desirable to explicitly manage structure. For example, in large open distributed systems components may appear dynamically as the result of individual user action and disappear as the result of user action or failure. There is no overall management control of the system, which may span many organisational boundaries. Components must bind to the services they require as a result of their own actions without the help of explicit configuration (structure) management. They are expected to be self-organizing.

Why an architectural approach? In addition to the autonomy inherent in self-organizing systems, we wish to retain the benefits of an overall software architecture specification so that, despite the introduction and removal of components, the system will remain well-formed with respect to its specification. In this way the *system can be made to preserve the architectural properties implied by its specification*. The architectural specification of a self-organising system is not a precise description of component instances and their interconnection but rather a set of *constraints* on the way components may be composed. In this sense, constraints can be considered akin to an architectural style [5], and can be used to generate and/or check a specific architectural instance for conformance. Furthermore, if a disturbance occurs, correcting changes could be generated.

This paper discusses the feasibility of using architectural constraints as the basis for the specification, design and implementation of self-organising architectures for distributed systems. In section 2, we address the problem of ensuring that the architecture specification constrains the system to have the required set of structures. Although we focus on organising the structure of systems, we indicate how component state can influence re-configuration via interface attributes. In section 3, we present a runtime architecture that ensures that after component introduction or failure, the system stabilises with a structure that satisfies the specified constraints. We also present some results from our initial Java implementation of this architecture. In section 4, we discuss related work and in section 5, conclude with an evaluation of the current work and present a research agenda of unresolved issues.

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 '02, Nov 18-19, 2002, Charleston, SC, USA.

© 2002 ACM 1-58113-609-9/02/0011...$5.00

المنارة للاستشارات
www.manaraa.com

2 ARCHITECTURE SPECIFICATION

In specifying the architecture of self-organising systems, we retain the structural view embodied in the Darwin ADL [15]. System architecture is a directed graph in which the nodes are component instances and the arcs specify bindings between a service required by one component and the service provided by another. However, to enable analysis of architectural specifications, we have chosen not to use Darwin directly but to model Darwin components in Jackson's Alloy language [9] and express structural constraints directly in Alloy. This has the advantage that the Alcoa tool [10] can then be used to generate, explore and analyse the specifications. It has the further advantage of allowing us to explore the semantics and features required of an ADL for self-organising systems without the burden of syntax design. At the current stage of our research, we feel language design would be premature although this is clearly a future direction. Alloy has successfully been used in analysing the properties of COM [11] systems.

In the following, we present the Alloy specification for Darwin components and bindings and use in determining the structural constraints for a simple pipeline architecture. We use this simple example to explain the approach, however we have both specified and implemented a much more complex replicated file system architecture [6].

2.1 Darwin Component Model

A component in Darwin, is a container of provided and required services. Services are provided and required via ports. Ports are typed with the interface used to access the service. In Figure 1 below, the component C has two provided ports {P0, P1}, the filled in circles and three required ports {P2, P3, P4}, the empty circles. Each port has an interface type e.g. P0 has type T0.

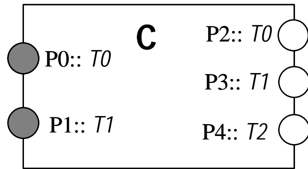

Figure 1 – Component

Model Elements

The basic elements of the Alloy model of Darwin are the components (Comp), service ports (Port) and interface types (Type):

```
domain {Port, Comp, Type}
```

Components and ports denote instances, not types. Components are implicitly associated with a type through set membership. A port (i.e. port instance) is associated directly with a port type (interface) and indirectly with all its supertypes.

Ports

Every port instance is associated with exactly one port type – the port's interface. Port types may inherit from other port types. A port type may have multiple immediate base types, i.e. multiple inheritance is supported.

```
type (~inst): Port -> static Type!
baseType: static Type -> static Type
```

Ports come in two flavours - provided and required:

```
partition Prov, Req: static Port
```

Components

A component is always associated with the same fixed number of ports during its lifetime. Conversely, ports should always be contained by the same simple component during their lifetime. The set of ports of a component is the union of its provisions and requirements. Thus `port` derives from the `prov` and `req` relations.

```
prov: static Comp! -> static Prov
req: static Comp! -> static Req
port: static Comp! -> static Port
```

Bindings

By definition a requirement can be bound to at most one provision. Required ports may be bound to different provided ports at different times. The model supports rebinding semantics by making all binding relations non-static.

```
reqBind (~provBind): Req -> Prov?
```

A provision can be bound to a requirement of the same type or a super type.

```
all p:Prov | p.provBind.type in p.type.*baseType
```

Although, we have a complete model for Darwin that includes hierarchical component composition, the above definitions are sufficient for the following constraint based which we have implemented as a self organising system.

2.2 A Pipeline Architecture

In the following, we outline the Alloy constraints for a pipeline architecture. Each component in a pipeline has exactly one provision and one requirement and the components are bound together to form a single chain. A binding constraint on the left end of the pipeline prevents it from becoming a ring. This restriction extends implicitly to the right side of the pipeline due to the cardinality constraint on ports and bindings.

?? Every component has one provision and one requirement:

```
all c:Comp | (one c.prov) && (one c.req)
```

?? Cardinality constraint: a provided port has at most one required port bound to it and vice-versa:

```
all c:Comp | sole c.prov.provBind
```

?? The pipeline forms a single chain:

```
some c:Comp | c.*connected = Comp
```

?? The pipeline does not form a ring, i.e. the provided port of the left end of the pipeline should not be connected to any component of the pipeline chain. The left end can be unbound.

```
some leftEnd:Comp
     | no (leftEnd.prov.bind.~port & Comp)
```

The last two constraints require global knowledge of the system. We will see that these global constraints are more difficult to enforce in a self-organising system than local constraints that can be enforced by a component with no knowledge of the rest of the system. For example, the cardinality constraint is local.

www.manaraa.com

In addition to checking that invariants hold for the specification, we can use the Alcoa analyser tool to produce an example or witness of the pipeline architectural style as shown in Figure 2. This Alcoa feature has proved invaluable in developing specifications as the examples usually indicate immediately that a specification is too weak. The condition `somePipeline` gives a hint to Alcoa on generating a sample architectural instance over a scope of three distinct components:

```
cond somePipeline {
   some com1, com2, com3:Comp | (com1 != com2) &&
   (com1 != com3) && (com2 != com3) }
```

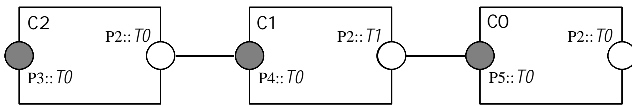

Figure 2 - Pipeline instance

Note that in Figure 2, `T0` is a subtype of `T1`; therefore the binding between `C1.P2` & `C0.P5` is valid.

In practise, pipelines are most commonly used as an adaptor between other structures and the edges of the pipeline are connected to components that don't belong to the pipeline chain. Additionally, components will have additional ports not concerned with pipeline communication, We can add constraints on the types of ports to reflect these situations. In addition, we can add constraints based on component application attributes to enforce ordering – a requirement for protocol stacks. The above constraints are the most general properties of a pipeline. We use application specific constraints in the file system example developed in [6].

In this section, we have used Alloy/Alcoa to show the feasibility of using an ADL with constraints as the basis of specifying, generating and analysing the architecture descriptions necessary for self-organising systems. In the next section, we explore the feasibility of an execution environment that supports self-organisation by satisfying architectural constraints.

3 RUNTIME ARCHITECTURE

Our work in self-organising architectures is firmly targeted at a distributed execution environment. The most difficult characteristic of this environment is arbitrary failure where components may fail suddenly without the opportunity to interact with the rest of the system. Previous work on change management [13] assumed firstly that the change manager did not fail and secondly that components had the opportunity to complete communication transactions before being removed. In this work, we do not make these assumptions and investigate the feasibility of a runtime architecture that has no centralised configuration management service. Our only assumption about the distributed execution environment is that network partitions do not occur. This assumption is necessary since as outlined in the following we use atomic broadcast to maintain a consistent view of architecture configuration. In the following, we first describe the elements of a prototype runtime component, next the supporting execution environment for components and finally illustrate how these combine to support self-organisation.

3.1 Runtime Components

The implementation of a component is packaged with a component manager and the configuration view to form a runtime component as shown in Figure 5.

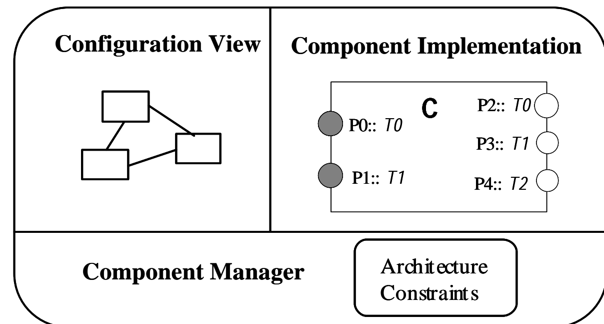

Figure 5 – Runtime Component

Component Implementation

In the current prototype implementation, components are implemented in Java and use Java RMI to access remote interfaces. Component implementations may be multi-threaded. Both the service interfaces implemented by a component (provided services), and those that are accessed remotely (required services) are managed by **`Port`** objects. Ports provide the management interface between the component application and the component manager. They control the binding of required interfaces and support event listener interfaces to allow the component implementation to react to changes in binding state. Events are generated by the component manager when a required port becomes bound to/ unbound from a remote interface. Similarly, events are generated when a remote interface is bound to/ unbound from a provided interface.

Although we have chosen not to focus on individual component participation in this paper, we make provision for a component to signal to the configuration manager to indicate a significant change of its internal state via **`Attribute`** objects. For example, the replicated file server uses an attribute object to signal changes in mastership to the configuration manager.

Configuration View

Included in each runtime component is a view of the system configuration state maintained by the component manager. The view consists of a descriptor for each component type currently included in the system and a directed graph of the configuration in which the nodes are component instances and the arcs are bindings from required to provided ports. The view is updated by the component manager in response to the binding and unbinding actions it performs and in response to external events indicating the arrival of new components, component removal/failure and binding/unbinding actions taken by other configuration managers. Maintaining a consistent view of the current configuration state is a critical property for the correctness of self-organisation.

Component Manager

Management of overall system configuration is achieved by the set of component managers. As described in the above, the component manager is responsible for maintaining the configuration view and for managing the component

المنارة للاستشارات
www.manaraa.com

implementation. Each component manager in a system is parameterised with the set of architectural constraints that describe the required architectural style as specified in section 2. When a component introduction/removal or attribute change event occurs it evaluates these constraints against the current configuration view to compute the required binding and unbinding actions needed to satisfy the constraints.

3.2 Runtime Support Environment

Component managers need to be apprised of the introduction of new components and removal/failure of existing components. In addition, each component manager must maintain a consistent view of the current configuration state. To maintain a consistent view, each manager must see the same set of events that change the configuration state. As depicted in Figure 6, we use a group membership service to detect component joining and leaving events and reliable broadcast to disseminate events between managers.

In addition to maintaining a consistent view of configuration state, a component manager must also be able to perform consistent modifications. To do this we use a totally ordered broadcast to implement a distributed locking scheme. To perform a configuration change, a manager must obtain the change lock before performing a change and release it afterwards. The broadcast system delivers join/leave events and broadcast messages using virtual synchrony [2]. The change lock and virtual synchrony ensure that a manager always performs a modification on a view of the system that has not been invalidated by some previous action.

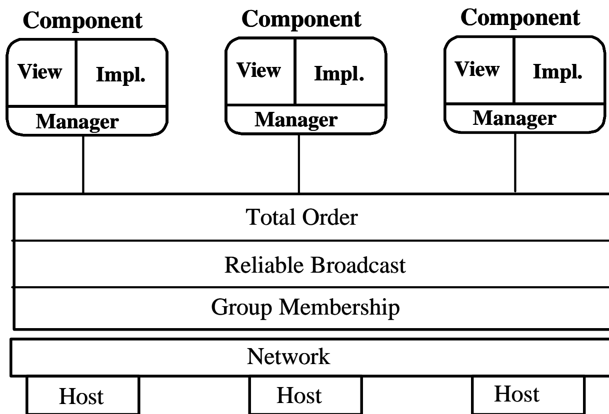

Figure 6 – Runtime Support Environment

We are currently using the OpenSource JavaGroups [1] implementation for our runtime support environment. It should be noted that while component managers communicate using JavaGroups, component application implementations can communicate via normal Java RMI or indeed use any available transport mechanism.

3.3 Self Organisation by Constraint Satisfaction

Changes to the configuration view held in a component occur as a result of component join/leave events signalled by the group membership service and binding/unbinding actions taken by component managers.

When a change to the current configuration view occurs, each configuration manager, for each required port of the component it manages, computes the binding needed to satisfy the architecture constraints. It does this by evaluating a set of configuration rules that conceptually take the form:

<required-port, selector, action-list>

although currently they are implemented as Java classes. The *required-port* identifies the port to which the rule applies, the *selector* function is evaluated with respect to the configuration view, to find a provided port and the *action-list* consists of a bind action for the *required-port* and in some cases unbind actions.

A major goal of our work is to derive these rules, which are loaded into a component manager, automatically from a declarative expression of constraints of the kind used in section 2. However, at the current stage of our research, the rule-based part of a component manager is designed manually, guided by the Alloy specification.

In summary, the goal of each component manager is to find a binding, for each of its required ports, that satisfies the architectural constraints. A configuration manager re-evaluates the selector function from each configuration rule every time the configuration view changes. The architecture stabilizes when all those required ports that can be bound are bound. Stability is guaranteed in the absence of continuing failure for those systems in which configuration rules guarantee monotonically increasing binding. This is true for both versions of the pipeline system since the rules either replace or add bindings – they do not remove bindings.

3.4 Implementation results

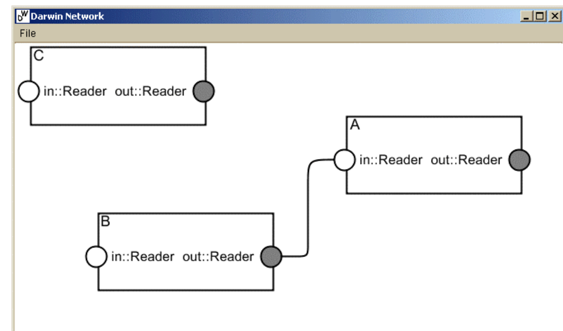

Figure 8 – C joins and C.out binds to B.in

As noted previously, components are implemented in Java and their execution is supported by JavaGroups. We have implemented a monitoring tool to view runtime structure. It joins the group channel, retrieves the configuration view from one of the components and then maintains the view when configuration update events are broadcast. The Diva graph visualization library from the Univ. of California at Berkeley is used to display the configuration graph. Figure 8 shows the monitoring tool display just after a component has been created and before it has been integrated into a pipeline architecture.

Figure 9 shows the time needed for a single component to integrate into an existing pipeline. The integration time is the time from a new component joining the group until the architecture stabilizes. The components were distributed among ten nodes (PCs) running on the same subnet. The times plotted in the graph

المنارة للاستشارات
www.manaraa.com

are the average over 20 integration executions with a fixed assignment of components to nodes. From the graph, it can be seen that the time needed for a component to integrate into a pipeline of length one is 170ms while the same time in a pipeline of length sixteen is 270ms. The increase of latency with pipeline length is mainly due to an increasing delay in message broadcasting with additional components in the group.

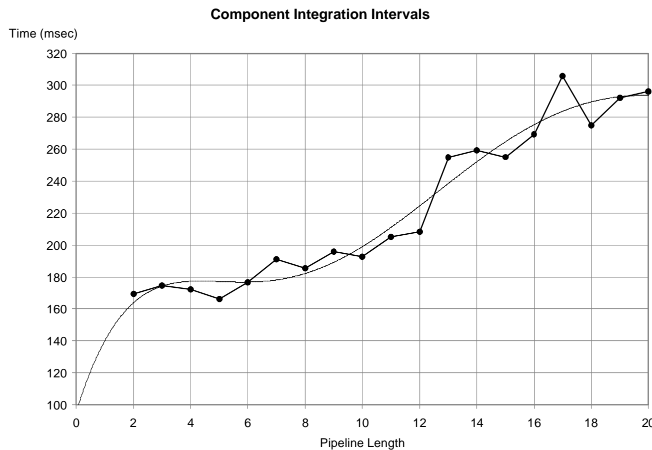

Figure 9 – Individual component integration time in a pipeline of 1 to 20 components

4 RELATED WORK

In specifying the architecture of self-organizing systems, we have used Alloy for the elegance and conciseness of the notation but more importantly for its associated automated analysis tools. Our specifications in Alloy are structural styles that architectural instances must conform to. Metayer [18] and subsequently Hirch [7] use graph grammars to express architectural styles and reason about conformance of change or evolution with respect to structural constraints. The Chemical Abstract Machine (CHAM) formalism has been used in the context of architecture description by Inverardi and Wolf [8]. Wermelinger [24] has employed the CHAM formalism to express architectural styles and has illustrated how CHAM execution can generate a sequence of actions to drive reconfiguration. Neither of these approaches has mechanical analysis support or deals with implementations.

In relation to runtime support required for reconfiguration Oreizy et al. [20] discuss an architectural approach to self-adaptive systems. They argue that architectural evolution needs support from a number of adaptation and evolution coordinators and monitoring tools. They have examples applied to the C2 architectural style. Our work has similar goals but has focused on structural re-organization and analysis.

Evaluation of constraints at runtime to control reconfiguration has been suggested by a number of other researchers. For example, the Raven configuration management system [3], uses constraints to recognize valid structures and to perform repairs. Minsky [19] suggests the use of "laws" to govern and constrain system re-configuration. However, rather than trying to deal with behavioural aspects and providing actions to maintain application state consistency as these system do, we have deliberately concentrated on structural organization. The issue of application response to re-organization is discussed below.

A recent paper [22] concerned with using architectural knowledge in self-repairing systems is closest in intent to what we are trying to achieve. The paper couples imperative directions for repairing a system with architectural constraints described in Armani.

5 RESEARCH ISSUES & AGENDA

The idea of basing self-organising software architectures on satisfying the constraints of a structural style was first proposed in a software architecture workshop position paper [17]. The research reported in this paper has gone some way towards making this idea concrete and testable. Firstly, we have shown that the required architectural styles can be expressed and subsequently analysed in a simple set based logical formalism. This provides a sound basis for specification and design although a special purpose constraint based ADL would undoubtedly be more accessible for practising architects. The replicated file system example[6] is a system of non-trivial complexity and shows how application state change can be incorporated in the architecture style. Although not discussed in this paper, components can conform to multiple styles. Secondly, we have designed and implemented a fully decentralised runtime system to support structural self-organisation. The decentralisation means that a system can re-organise in response to failure – a key goal of the approach. Finally, although our approach as presented expects that new components are always introduced or removed as a result of an action external to the system (i.e. by a user or by failure), in fact, there is no reason why the application cannot create new components or cause components to terminate. The way the system reorganizes to accommodate component addition/ removal remains the same.

Our investigation of self-organisation has raised a number of interesting issues that clearly require further research.

Selector function generation

Selector functions, as described in section 3, are evaluated when the configuration changes to find new bindings for required ports. Deriving these selector functions from the Alloy specification of constraints is currently a manual design step. The problem of mechanically deriving efficient selectors from constraints is something that we are actively investigating.

Application interaction with re-organisation

We have focused in this paper on structural re-organisation and explained that attributes allow component application state to influence component interconnection. We have mentioned that the application part of a component can be aware of re-configuration by listening on ports for binding/rebinding events. However, the paper has not addressed the question of how applications should be designed to take account of the possibility of dynamic rebinding. An approach to checking that a system preserves correctness properties with respect to application state in the presence of dynamic configuration change is described in [14].

Scalability & Efficiency

The use of reliable broadcast channels to co-ordinate component managers and maintain the replicated configuration view constrains the maximum size of systems. This is confirmed by the initial results reported in section 3. The time to re-organise a system increases with the number of components and this is likely to constrain maximum size to <100 components even with an efficient implementation of runtime support (which the current system is not). How do we build self-organising systems with

المنارة للاستشارات
www.manaraa.com

thousands of components? Can we relax the requirement for reliable broadcast? We are exploring two answers to these questions. Firstly, not all components need to have a complete and consistent configuration view. For example, the architecture for replicated servers described in [12] does not require clients to have any knowledge of the server structure. Secondly, we can use multiple broadcast groups and hierarchical structuring.

Architecture Evolution

How do we evolve the architecture itself to meet changing requirements? In the context of self-organising systems, this corresponds to changing the architectural constraints. Our current implementation supports the ability to dynamically load selector functions through Java's dynamic class loading capabilities. However, the interesting question is how to update selectors in a way that causes minimal disruption to the application and that results in a new stable system that conforms to the updated constraints.

References

[1] B. Ban, *JavaGroups User's Guide,* Cornell University, August 1999.

[2] K. Birman, A. Schiper and P. Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 271-314, 1991.

[3] T. Coatta and G. Neufeld, *Distributed Configuration Management using Composite Objects and Constraints,*, Distributed Systems Engineering Journal, Vol. 1, No. 5, pp. 294-303, 1994.

[4] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M.Sloman and K. Twidle, *Configuration Management forDistributed Systems*, Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (ISINM 95), Santa Barbara.

[5] D. Garlan, R. Allen and J. Ockerbloom, *Exploiting Style in Architectural Design Environments*, Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, pp. 175-188, December 1994.

[6] I. Georgiadis, *Self-organising Distributed Component Software Architectures*, Department of Computing, Imperial College, 2002.

[7] D. Hirch, P. Inverardi and U. Montanari, *Graph grammars and constraint solving for software architecture styles*, 3rd International Workshop on Software Architecture, pp. 69-72, 1998.

[8] P. Inverardi and A. L. Wolf, *Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 373-386, 95.

[9] D. Jackson, *Alloy: A Lightweight Object Modelling Notation,* MIT Lab for Computer Science, July 1999.

[10] D. Jackson, I. Schechter and I. Shlyakhter, *Alcoa: The Alloy Constraint Analyzer*, International Conference on Software Engineering, Limerick, Ireland, June 2000.

[11] D. Jackson and K. Sullivan, *COM Revisited: Tool-Assisted Modelling of an Architectural Framework*, ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering, San Diego, California, pp. 149-158.

[12] C. Karamanolis and J. Magee, *Client-Access Protocols for Replicated Services*, IEEE Transactions on Software Engineering, Vol. 25, No. 1, pp. , 1999.

[13] J. Kramer and J. Magee, *The Evolving Philosophers Problem: Dynamic Change Management*, IEEE Trans. on Software Engineering, Vol. 16, No. 11, pp. 1293-1306, 1990.

[14] J. Kramer and J. Magee, *Analysing Dynamic Change in Distributed Software Architectures*, IEE Proceedings - Software, Vol. 145, No. 5, pp. 146-154, 1998.

[15] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, 5th European Software Engineering Conference (ESEC'95), Sitges, Spain, 989, pp. 137-153, September 1995.

[16] J. Magee and J. Kramer, *Dynamic Structure in Software Architectures*, 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4), San Francisco, California, USA, 21, pp. 3-14, October 1996.

[17] J. Magee and J. Kramer, *Self-Organising Software Architectures*, 2nd International Software Architecture Workshop.

[18] D. L. Metayer, *Software Architecture Styles as Graph Grammars*, 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 15-23, November 1996.

[19] N. Minsky, *Building Reconfiguration Primitives into the Law of a System*, 3rd International Conference on Configurable Distributed Systems, Annapolis, pp. 89-97.

[20] P. Oreizy, M. M. Gorlick, R. N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems, Vol. 14, No. 3, pp. 54-62, 1999.

[21] D. E. Perry and A. L. Wolf, *Foundations for the Study of Software Architectures*, ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, 1992.

[22] B. Schmerl and D. Garlan, *Exploiting Architectural Design Knowledge to Support Self-repairing Systems*, 14th International Conf. on Software Engineering and Knowledge Engineering, Ischia, Italy, 2002.

[23] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 96.

[24] M. Wermelinger, *Towards a Chemical Model for Software Architecture Reconfiguration*, IEE Proceedings - Software, Vol. 145, No. 5, pp. 130-136, 1998.

المنارة للاستشارات

www.manaraa.com