Self-Managed Systems: an Architectural Challenge

Jeff Kramer and Jeff Magee

Jeff Kramer is Dean of the Faculty of Engineering at Imperial College London, and was Head of the Department of Computing from 1999-2004. His research interests include rigorous techniques for requirements engineering; software specification, design and analysis; and software architectures, particularly as applied to distributed and adaptive software systems. Jeff is the Editor-in-Chief of the IEEE Transactions on Software Engineering, and the co-recipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in Distributed Software Engineering. He is co-author of a recent book on Concurrency, co-author of a previous book on Distributed Systems and Computer Networks, and the author of over 200 journal and conference publications. He is a Chartered Engineer, Fellow of the IET, Fellow of the BCS and Fellow of the ACM.

Jeff Magee is Head of the Department of Computing at Imperial College London. His research is primarily concerned with the software engineering of distributed systems, including requirements, design methods, analysis techniques, operating systems, languages and program support environments for these systems. He is co-author of a recent book on concurrent programming entitled "Concurrency - State models and Java programs" and the author of too many journal and conference publications. He was co-editor of the IEE Proceedings on Software Engineering and is currently a TOSEM Associate Editor. He is the co-recipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in Distributed Software Engineering. He is a Chartered Engineer, Member of the IET and Fellow of the BCS.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Self-Managed Systems: an Architectural Challenge

Jeff Kramer and Jeff Magee
Department of Computing
Imperial College London
SW7 2AZ, UK

{j.kramer, j.magee}@ ic.ac.uk

Abstract

Self-management is put forward as one of the means by which we could provide systems that are scalable, support dynamic composition and rigorous analysis, and are flexible and robust in the presence of change. In this paper, we focus on architectural approaches to self-management, not because the language-level or network-level approaches are uninteresting or less promising, but because we believe that the architectural level seems to provide the required level of abstraction and generality to deal with the challenges posed. A self-managed software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system. 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 discusses some of the current promising work and presents an outline three-layer reference model as a context in which to articulate some of the main outstanding research challenges.

1 Introduction

As the size, complexity and adaptability required by applications increases, so does the need for software systems which are scalable, support dynamic composition and rigorous analysis, and are flexible and robust in the presence of change. Self-management is the rallying vision. What exactly are self-managed systems? The vision is of systems which are capable of self-configuration, self-adaptation and self-healing, self-monitoring and self-tuning, and so on, often under the flag of self-* or autonomic systems.

For instance, consider that you have a specification of the goals, properties and constraints that you expect your system to achieve and preserve. Consider further that you have a set of software components which implement the required functionality. The aim of self-configuration is that the components should either configure themselves such that they satisfy the specification or be capable of reporting that they cannot.

What if the system suffers from changes in its requirements specification [14] or operational environment such as changes in use, changes in resource availability or faults in the environment or in parts of the system itself? The aim of self-adaptation and self-healing is that the system should reconfigure itself so as to again either satisfy the changed specification and/or environment, or possibly degrade gracefully or report an exception. Change as evolution of the system tends to imply an off-line process in which the system evolves through a number of releases, where each release could employ self-configuration. However, dynamic change, which occurs while the system is operational, is far more demanding and requires that the system evolves dynamically, and that the adaptation occurs at run-time.

Finally, we should note that our required specifications include not only functional behaviour, but also those non-functional properties such as response time, performance, reliability, efficiency and security, and that satisfaction of a specification may well include optimisation.

Clearly this is a challenging vision, which includes almost every one of the research challenges identified in the first FOSE: Software Engineering: A Roadmap at ICSE 2000 [19], namely compositionality, change, NF properties, service-view, perspectives, architecture, configurability, and domain specificity.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

How can we approach this dream?

Different research communities are already engaged in relevant research, investigating and proposing approaches to various aspects of self-management for particular domains. For instance, in the networking, distributed systems and services community, there has been the Autonomic Computing conferences [2] and more recently, the SelfMan Workshop 2006 [1] to discuss and analyse the potential of self-* systems for managing and controlling networked systems and services. A special issue associated with SelfMan 2005 on Self-Managed Systems and Services [31] covers a diverse range of descriptions of work on aspects such as self-healing by dynamic fault awareness and self-adaptation/tuning for dealing with transient web overloads. Dobson et al. [17] provide a recent survey on autonomic communications, covering research work on context awareness, autonomic algorithms, trust and security and appropriate adaptation techniques. They propose an autonomic control loop (a phased approach) of actions *collect* (monitoring), *analyse*, *decide* and *act*, a cycle which naturally appears in many proposed approaches. In addition to those mentioned above, there is also the International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006) [3], and the International Conference on Autonomic and Autonomous Systems ICAS 2006 [6].

The proliferation of conferences and workshops mentioned above reflects the interest in the topic; and this is only from the networking, systems and services communities. Other research communities also interested and appropriate include the intelligent agent, machine learning and planning communities, and many others, adopting underlying models as diverse as those derived from biology and social interaction.

In the software engineering community there has been a series of workshops which started in the distributed systems community with the CDS (Configurable Distributed Systems) conferences [4, 9, 5] and more recently with WOSS (Workshop on Self-Healing and Self-Managed Systems) [8, 7] and SEAMS (Software Engineering for Adaptive and Self-Managing Systems) [3]. These conferences and workshops have provided excellent forums for discussing the software issues involved. However, although the work discussed over the years has provided much that is useful in contributing towards self-management, it has not yet resolved some of the general and fundamental issues in order to provide a comprehensive and integrated approach.

Why an architectural approach?

In this paper we focus on the use of an architecture-based approach, as we believe that it offers the following potential benefits:

- *Generality* – the underlying concepts and principles should be applicable to a wide range of application domains, each associated with appropriate software architectures.
- *Level of abstraction* – software architecture can provide an appropriate level of abstraction to describe dynamic change in a system, such as the use of components, bindings and composition, rather than at the algorithmic level.
- *Potential for scalability* – architectures generally support both hierarchical composition and other composition and hiding techniques which are useful for varying the level of description and the ability to build systems of systems, thereby facilitating their use in large-scale complex applications.
- *Builds on existing work* – there is a wealth of architecture description languages and notations which include some support for dynamic architectures and for formal architecture-based analysis and reasoning [12]. These provide a good basis for a rigorous approach which could support evaluation and reasoning, constraints and run-time checks.
- *Potential for an integrated approach* - many ADLs and approaches support software configuration, deployment and reconfiguration. In fact, as mentioned in an accompanying FOSE paper in this proceedings on Software Design and Architecture [37], "software architecture encompasses work in modelling and representation, design methods, analysis, visualization, supporting the realization of designs into code, experience capture and reuse, product lines, deployment and mobility, security, adaptation, and so on."

We are not alone in favouring a component-based architectural approach. Many others also advocate use of architectural principles in their work. For instance, Oreizy et al [34] provide a general outline of an architectural approach which includes adaptation and evolution management; Garlan and Schmerl [21] describe the use of architecture models to support self-healing; Dashofy, van der Hoek and Taylor propose the use of an architecture evolution manager to provide the infrastructure for run-time adaptation and self-

healing in ArchStudio [16]; Gomaa and Hussein [24] describe the use of dynamic software reconfiguration and reconfiguration patterns for software product families; Medvidovic, Rosenblum and Taylor present a language and associated environment for architecture-based development and component evolution [33]; Wang et al [39] describe an experiment to support component-based dynamic software evolution; Baresi et al. [11] suggest the use of contracts expressed as assertions to monitor and check dynamic service compositions in service-oriented systems; and Castaldi et al. [13] extend the concepts of network management to component-based, distributed software systems to propose an infrastructure for both component- and application-level reconfiguration using a hierarchy of managers. Our own work has concentrated on the use of ADLs for software design and implementation from components [29], including limited language support for dynamic change [30], a general model for dynamic change and evolution [28], associated analysis techniques [27] and initial steps towards self-management [23].

In order to try to draw all these threads together, we now propose an architectural reference model as a means of identifying more precisely the concerns and research issues that are needed in progressing towards self-management.

2 Towards an Architectural Model for Self-Management

In taking initial steps in the direction of an architecture model for self-management, we have sought inspiration from the large existing body of work on autonomous systems – namely robotics. The first architectures proposed for self–management correspond nearly exactly with the early sense-plan-act SPA architectures used in robots (cf. autonomic control loop [17] mentioned earlier). For example, Garlan's proposed adaptation framework for self-healing systems [21] consists of monitoring, analysis/resolution and adaptation. The monitoring of system operation corresponds to a robot sensing its environment, the analysis/resolution of faults corresponds to planning and adaptation or the execution of changes corresponds to action in the SPA framework. Indeed, Garlan's framework maintains an abstract model of a system in the same way as SPA robots try to maintain a symbolic model of their environment. It is not surprising that this correspondence exists since a self-managed system is clearly an autonomous system in exactly the same way as a robot is. Both are intended to achieve goals without human intervention. Since the SPA architectures of the early eighties, robot architectures have evolved considerably and now, since the mid-90's, nearly all conform to the three layer architecture described by Gat [22]. In Gat's paper's, the three layers are Control: reactive feedback control, Sequencing: reactive plan execution and Deliberation: planning. In the following, we attempt to interpret this three-level robotic architectural model for self-managed systems. Our goal is to exploit the considerable advances that modern robotic systems have in terms of flexibility and responsiveness over their SPA predecessors.

2.1 Component Control

The bottom layer of Gat's three layer architecture is the control layer. It consists of sensors, actuators and control loops. The bottom layer of a self-managed system consists of the set of interconnected components that accomplish the application function of the system. It must of course include facilities to report the current status of components to higher layers and also include the capability to support component creation, deletion and interconnection. In the same way that the control layer of a robot includes feedback loops to implement primitive behaviours such as wall following and moving to a destination, the bottom layer of a self-managed system will contain behaviours to adjust the operating parameters of components – for example the timeout values in a component implementing a TCP protocol. In summary, this layer of a self-managed system will include self-tuning algorithms, event and status reporting to higher levels and operations to support modification – component addition, deletion and interconnection. An important characteristic of this level, is that when a situation is met that the current configuration of components is not designed to deal with, this layer detects this failure and reports it to higher layers.

2.2 Change Management

The middle layer of Gat's three layer architecture is the sequencing layer which reacts to changes in state reported from the lower levels and executes plans that select new control behaviours and set new operating parameters for existing control layer behaviours. This is reactive plan execution. Given a new situation, this layer executes an action or sequence of actions to handle the new situation. For example, when the robot reaches a target location, this layer will determine what should be done next. In a self-managed system, this layer is responsible for effecting changes to the

underlying component architecture in response to new states reported by that layer or in response to new objectives required of the system introduced from the layer above. This layer can introduce new components; recreate failed components; change component interconnections and change component operating parameters. It consists of a set of plans which are activated in response to changes of the operating state of the underlying system. For example, when a component fails, change management can effect a repair either by changing component connections or by creating new components. In robotic systems, this layer has been implemented in a number of ways from conditional sequencing systems [10] to sets of state machines. Work in the network management area has produced languages such as Ponder [15] which perform a similar function to the planning languages in the context of systems. Ponder is essentially a language which execute actions in response to recognising (possible complex) events. The essential characteristic of this change management layer is that it consists of a set of pre-specified plans which are activated in response to state change from the system below. The layer can respond quickly to new situations by executing what are in essence pre-computed plans. If a situation is reported for which a plan does not exist then this layer must invoke the services of the higher planning layer. In addition, new goals for a system will involve new plans being introduced into this layer.

2.3 Goal Management

The uppermost layer of Gat's three layer architecture is the deliberation layer. This layer consists of time consuming computations such as planning which takes the current state and a specification of a high-level goal and attempts to produce a plan to achieve that goal. An example in robotics would be given the current position of a robot and a map of its environment produce a route plan for execution by the sequencing layer. Changes in the environment, such as obstacles that are not in the map, will involve re-planning. The role of the equivalent layer in a self-managed system is Goal Management. This layer produces change management plans in response to requests from the layer below and in response to the introduction of new goals. For example, if the goal in to maintain some architectural property such as triple redundancy for all servers, this layer could be responsible for finding the resources on which to create new components after failure and producing a plan as how to create and integrate these new components to the change management layer. It could be responsible for deciding the optimal placement of servers for load balancing purposes. As we will address further in the next section there are many research issues here as to how to represent high level system goals, how to synthesize change management plans from these goals and how general or domain specific this layer should be.

Figure 1 summarises our proposed three layer model for a self managed system following Gat's work on architectures for robotic systems. The principal criteria for placing function in different layers in Gat's architecture is one of time scale and this would seem to apply equally well to self managed systems. Immediate feedback actions are at the lowest level and the longest actions requiring deliberation are at the uppermost level. We would emphasize that we do not consider this an implementation architecture but rather a conceptual or reference architecture which identifies the necessary functionality for self management. We will use it in the next section to organise and focus discussion of the research challenges present by self management.

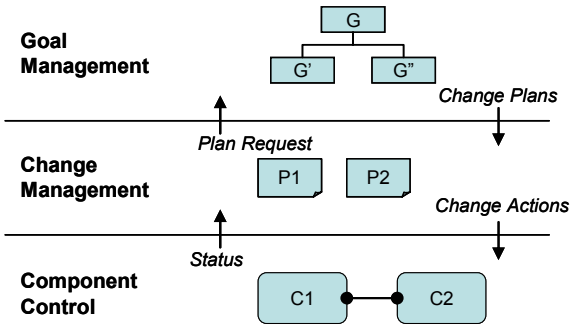

Figure 1 – Three Layer Architecture Model for Self-Management.

3 Research Issues

In the previous section we outlined a three layer architecture model which is intended as a form of reference model rather than as a guide to how self managed software should be implemented. In this section, we use the model to structure the presentation of the research issues we see presented by the challenge of implementing self-managed systems. To ground this discussion, we draw examples from the work with which we are most familiar – namely our own.

3.1 Component Control Layer

We are concerned with management at the architectural level where we consider a system to

consist of a set of interconnected components which may be co-located and/or distributed over a network of communicating computer nodes. Our model of a component is depicted in figure 2 below. A component implements the set of services that it provides and it may use another set of services, denoted the required services, in implementing these services. In addition, a component has an externally visible state which we term its *mode*. The mode is simply an abstracted view of the internal state of a component that is made visible for management purposes. For example, the mode may indicate whether the component is in an active or standby mode. It may in addition indicate non-functional aspects such as the current load on a server. Mode may therefore consist of more than a simple scalar datatype, although in our examples we use only a simple scalar mode. We have used the Darwin [29] format for diagrams updated with modes [25], however, we could equally have used UML 2.0 as the Darwin form of component can now be satisfactorily encoded in UML2.0 [32].

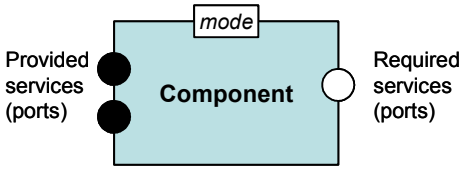

Figure 2 – Example component model

To initially construct and subsequently change systems, we need a set of operations on components. These are typically:

create C: T
– create component instance **C** from type **T**.
delete C
– delete component instance **C**.
bind C_1.r – C_2.p
– connect required port **r** of component **C_1** to provided port **p** of component **C_2**.
unbind C_1.r
– disconnect required port **r** of component **C_1**
set C_1.m to *val*
– set mode **m** of component **C_1** to *val*.

A system constructed in this way will have a configuration or management state consisting precisely of the set of components instances, the set of connections between components and the set component mode values – an example architecture for an autonomous underwater vehicle [20] operating in a mode in which the sonar is passive shown in Figure 3.

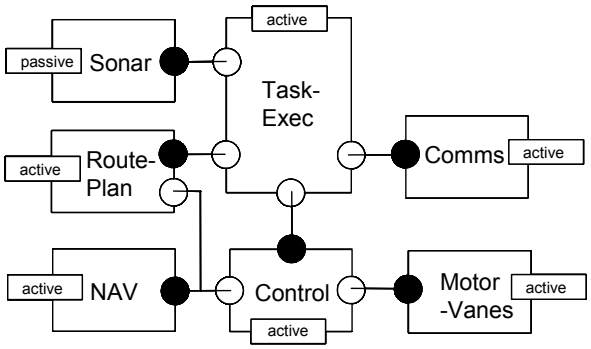

Figure 3 – Example component architecture

The research challenges at this level of a self-managed architecture are primarily concerned with preserving safe application operation during change. For example, the change of mode in a mechatronic system controlling a vehicle [36] can involve moving from one control algorithm to another. The implementation at this level must ensure that the mode change required to adapt to a change of the external operating environment does not generate undesirable transient behavior resulting in, for example, sharp accelerations or decelerations. In systems where the behavior is transactional rather than continuous, the challenge is to ensure that state information is not lost when the configuration is modified. The change management algorithm outlined in [28] tries to ensure stable conditions for change by ensuring that components are passive or quiescent before change. For example, a component can be safely removed from a system if it is isolated (no bindings to or from) and passive (cannot initiate transactions). The challenge is to find scalable algorithms that minimize disruption to the system during change and ensure that system safety properties are not violated. An associated challenge is to verify that safety properties are not violated during change [27], a problem addressed more promisingly by Zhang and Cheng [40].

We have looked at a system as a collection of components; however a component itself may consist of multiple interconnected components. To deal with complex systems in a scalable way, we must deal with hierarchical structure. This raises interesting issues with the respect to the type of a component. When we modify the internal structure of a component, we are

clearly creating a variant of its type. We may want to instantiate new instances of this variant. It is likely that self-management of system will require the online dynamic execution of operations that we currently consider to be offline maintenance or version control operations.

3.2 Change Management Layer

This layer as outlined in section 2 is responsible for executing changes in response either to changes in state reported from the lower layer are in response to goal changes. We think of this layer as a precompiled set of plans or tactics that respond to a predicted class of state change. For example, in a fault-tolerant system, failure of a component may cause a duplicate server to immediately switch from standby to active mode; however, the state change observed by change management should cause the system to create a new standby server. In a fault tolerant system, it is clear that server failure is a predicted state change and the change management layer will include a procedure for dealing with the change. Similarly, we would consider the example repair strategy outlined by Garlan and Schmerl [21] as a plan executed by change management.

One of the major research challenges at this level is dealing with distribution and decentralization. It is this issue of distribution or decentralization that appears to be the essential factor in distinguishing the problem of performing self-management of complex software systems from existing work on robotic systems. Distribution is the most general situation raising issues of latency, concurrency and partial failures, and is likely to be the case (at least for parts of the system) in large and complex applications. Coping with distribution and arbitrary failure leads to the need for some level of local autonomy while preserving global consistency. In essence, distribution contributes the problem of obtaining consistent views of system state on which to base change decisions and decentralization of control brings the problem of robust execution in a situation in which partial failure can occur. Our first attempt to deal with this resulted in a change architecture with completely decentralized change execution; this, however, required state change to be serialized to ensure termination of the configuration in a valid state [23]. This decentralized implementation architecture is shown in Figure 4.

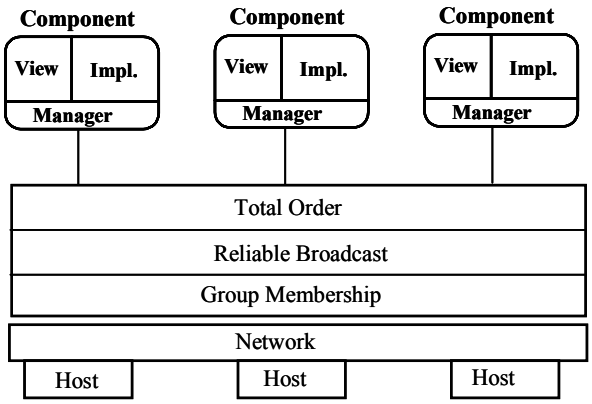

Figure 4 – Decentralized Change Management Implementation architecture

The system depicted in Figure 4, was an experiment to look at extreme distribution in which change management layer functionality was included with each component. Each component maintained a view of the overall system and executed local changes (include connection to other components) in response to state changes in the view. The problems with this system were 1) the view of the system has to be complete and 2) it requires a total order broadcast bus to keep views consistent. Consequently, this was a fully decentralized architecture that reliably executed change in the presence of arbitrary failure. However, it was not a scalable architecture. What we require are systems which can accommodate partial inconsistent views and as a consequence relax the need for totally ordered broadcast communication. The challenge is to find change management algorithms that can tolerate inconsistency and which eventually terminate in a system that satisfies constraints. It is also required that the system does not violate safety constraints while it is converging on a stable state. There are of course examples of self stabilising algorithms [18]; however, these are for specific configurations and applications.

One of the goals of the system described in Figure 4 was to preserve global structural constraints. It was primarily this requirement that dictated a consistent view of system structure. It may well be that taking a more behavioural view of system constraints will provide opportunities for relaxing the consistency requirement. For example, if we are not at all interested in structure, components can simply bind to any service that satisfies the local requirement. Failure of the remote service can trigger a search for a replacement service.

3.3 Goal Management Layer

The initial problem is to have a precise specification of the goals required of a system. These need to encompass both application goals and system goals concerned with self-management. It is likely that the refinement of very high-level goals to precisely specified goals that can be processed by machines will require human assistance as is current practice in goal oriented requirements engineering [38]. The challenge is to achieve goal specification such that it is both comprehensible by human users and machine readable.

If we ignore the problem of precisely specifying structure, behaviour and performance required of a system, the function of goal management can be succinctly described. It takes a declarative specification of system goals, a snapshot of the current state of the system and produces a change plan which moves the system from its current state to a state which satisfies the system goals. Regrettably, this is of course a computationally hard and sometimes intractable problem. Even when tractable, the time taken to generate a plan may not meet the response times required.

Solutions so far have focussed on dealing as far as is possible with planning by designing a set of plans (sometimes referred to as tactics) offline that can be shown either by construction or by a verification process to satisfy system constraints for a range of possible system states. For example, in our system described in [23], we specified system structural constraints in Alloy [26] and developed tactics that could be shown using the Alloy model checker to move a system into a state that satisfied constraints. In other words, the planning problem was done off-line and the problem reduced to one of verification. This approach is sufficient if it can be shown that the set of change plans are sufficient to deal with any possible system state. This is of course exactly what is done in some of the classic fault-tolerant architectures such as active-standby server pairs.

There has been promising work in the autonomous composition of Web services using a "planning as model-checking approach" [35]. In essence, this is in its present state an off-line planning approach. The more challenging problem in the spirit of autonomous self-management is to provide an on-line planner. This is invoked by the change management layer when it finds that none of its current plans apply to the observed system state. This is where the real research challenges in true self-management lie – in the automatic decomposition of goals and in the generation of operationalized plans from these goals. The usual strategy of constraining the problem domain will undoubtedly help.

4 Conclusion

In this paper, we have described our vision of self-management at the architectural level, where a self-managed software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system. We chose to concentrate on an architectural approach as we believe that this offers the required level of abstraction and generality to integrate some of the possible solutions to the challenges posed. We are biased towards a rigorous engineering approach in which low-level actions can be clearly and formally related to high-level goals that are precisely specified.

We have defined a three layer reference model – component control, change management and goal management – to provide a context for discussing the main research challenges which self-management poses. At the component layer, the main challenge is to provide change management which reconfigures the software components, ensures application consistency and avoids undesirable transient behaviour. At the change management layer, decentralized configuration management is required which can tolerate inconsistent views of the system state, but still converge to a satisfactory stable state. Finally, some form of on-line (perhaps constraint based) planning is required at the goal management layer.

To provide a self-managed system, solutions to these challenges need to be integrated to provide a comprehensive solution, supported by an appropriate infrastructure. In addition, the approach must be amenable to a rigorous software development approach and analysis, so as to ensure preservation of desirable properties and avoid undesirable emergent behaviour. A challenge indeed!

References

[1] *2nd IEEE Int. Workshop on Self-Managed Networks, Systems and Services (SelfMan 2006)*, IEEE, Dublin, 2006.

[2] *The 3rd IEEE International Conference on Autonomic Computing* IEEE, Dublin, 2006.

[3] *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)*, Erfurt, Germany, September 2006.

0-7695-2829-5/07 $20.00 © 2007 IEEE

[4] *Proceedings of IEE/IFIP 1st Int. Workshop on Configurable Distributed Systems (CDS 92)*, in J. Kramer, ed., London, May 1992.

[5] *Proceedings of IEEE 3rd International Conference on Configurable Distributed Systems (CDS 96)*, in J. Magee and K. Schwan, eds., May 1996.

[6] *Proceedings of International Conference on Autonomic and Autonomous Systems ICAS 2006*, Santa Clara, July 2006.

[7] *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, in D. Garlan, J. Kramer and A. Wolf, eds., ACM Press, Newport Beach, California, 2004, pp. 119.

[8] *Proceedings of the first workshop on Self-healing systems*, in D. Garlan, J. Kramer and A. Wolf, eds., ACM Press, Charleston, South Carolina, 2002, pp. 120.

[9] *Proceedings. of IEEE 2nd International. Conference on Configurable Distributed Systems, Pittsburgh, (CDS 94).* in J. Kramer and J. Purtilo, eds., Pittsburgh, May 1994

[10] C. Agre and D. Chapman, *What are Plans for?*, Robotics and Autonomous Systems, 6 (1990), pp. 17-34.

[11] L. Baresi, C. Ghezzi and S. Guinea, *Smart monitors for composed services*, *Proceedings of the 2nd international conference on Service oriented computing*, ACM Press, New York, NY, USA, 2004.

[12] J. S. Bradbury, J. R. Cordy, J. Dingel and M. Wermelinger, *A survey of self-management in dynamic software architecture specifications*, *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM Press, Newport Beach, California, 2004.

[13] M. Castaldi, A. Carzaniga, P. Inverardi and A. L. Wolf, *A light-weight infrastructure for reconfiguring applications*, *Proceedings of 11th Software Configuration Management Workshop (SCM03)*, LNCS, Portland, Oregon, 2003.

[14] B. H. C. Cheng and J. Atlee, M., *Research Directions in Requirements Engineering*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[15] N. Damianou, N. Dulay, E. Lupu and M. Sloman, *The Ponder Policy Specification Language*, *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, Springer-Verlag, 2001.

[16] E. M. Dashofy, A. van der Hoek and R. N. Taylor, *Towards architecture-based self-healing systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[17] S. Dobson, S. Denazis, Fernndez, Antonio, D. Gati, E. Gelenbe, Massacci, P. Nixon, F. Saffre, N. Schmidt and F. Zambonelli, *A survey of autonomic communications*, ACM Trans. Auton. Adapt. Syst., 1 (2006), pp. 223-259.

[18] S. Dolev, *Self-Stabilization*, MIT Press, 2000.

[19] A. Finkelstein and J. Kramer, *Software engineering: a roadmap*, *Proceedings of the Conference on The Future of Software Engineering*, ACM Press, Limerick, Ireland, 2000.

[20] H. Foster, J. Magee, S. Uchitel and J. Kramer, *Scenario-Based Software Synthesis for Adaptable Software Architectures of UAVs*, *Proceedings of First Annual SEAS DTC Conference*, www.seasdtc.com, Edinburgh, 2006.

[21] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[22] E. Gat, *Three-layer Architectures*, *Artificial Intelligence and Mobile Robots*, MIT/AAAI Press, 1997.

[23] I. Georgiadis, J. Magee and J. Kramer, *Self-organising software architectures for distributed systems*, *Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, 2002.

[24] H. Gomaa and M. Hussein, *Dynamic Software Reconfiguration in Software Product Families*, *5th International Workshop on Software Product-Family Engineering*, LNCS 3014, Springer 2004, 435-444., Siena, Italy, 2003.

[25] D. Hirsch, J. Kramer, J. Magee and S. Uchitel, *Modes for Software Architectures*, *Third European Workshop on Software Architecture (EWSA 2006)*, Springer, Nantes, France, Sept 2006.

[26] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.

[27] J. Kramer and J. Magee, *Analysing dynamic change in distributed software architectures*, Software, IEE Proceedings-, 145 (1998), pp. 146-154.

[28] J. Kramer and J. Magee, *The evolving philosophers problem: dynamic change management*, Software Engineering, IEEE Transactions on, 16 (1990), pp. 1293-1306.

[29] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, *5th European Software Engineering Conference (ESEC)*, Sitges, Spain, 1995.

[30] J. Magee and J. Kramer, *Dynamic structure in software architectures*, *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, San Francisco, California, United States, 1996.

[31] J.-P. Martin-Flatin, J. Sventek and K. Geihs, *Special Issue on Self-managed systems and services* Commun. ACM, 49 (2006), pp. 36-39.

[32] A. McVeigh, J. Kramer and J. Magee, *Using resemblance to support component reuse and evolution*, *Proceedings of the 2006 conference on Specification and verification of component-based systems*, ACM Press, Portland, Oregon, 2006.

[33] N. Medvidovic, D. S. Rosenblum and R. N. Taylor, *A language and environment for architecture-based software development and evolution*, *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Angeles, California, United States, 1999.

[34] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, *An architecture-based approach to self-adaptive software*, Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems], 14 (1999), pp. 54-62.

[35] M. Pistore, A. Marconi, P. Bertoli and P. Traverso, *Automated Composition of Web Services by Planning at the Knowledge Level*, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, Edinburgh, Scotland, 2005.

[36] W. Schaefer and H. Wehrheim, *The Challenges of Building Advanced Mechatronic Systems*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[37] R. N. Taylor and A. van der Hoek, *Software Design and Architecture: The Once and Future Focus of Software Engineering*, in L. Briand and A. L. Wolf, eds., *Future of Software Engineering 2007*, IEEE-CS Press, 2007.

[38] A. van Lamsweerde, *Goal-Oriented Requirements Engineering: A Guided Tour*, *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, 2001.

[39] Q. Wang, J. Shen, X. Wang and H. Mei, *A component-based approach to online software evolution: Research Articles*, J. Softw. Maint. Evol., 18 (2006), pp. 181-205.

[40] J. Zhang and B. H. C. Cheng, *Model-based development of dynamically adaptive software*, *Proceeding of the 28th international conference on Software engineering*, ACM Press, Shanghai, China, 2006.