Kramer J, Magee J. A rigorous architectural approach to adaptive software engineering. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(2): 183–188 Mar. 2009

A Rigorous Architectural Approach to Adaptive Software Engineering

Jeff Kramer, *Fellow*, *ACM*, and Jeff Magee

Department of Computing, Imperial College London, London SW7 2AZ, U.K.

E-mail: j.kramer@imperial.ac.uk; j.magee@imperial.ac.uk

Received April 22, 2008; revised November 28, 2008.

Abstract The engineering of distributed adaptive software is a complex task which requires a rigorous approach. Software architectural (structural) concepts and principles are highly beneficial in specifying, designing, analysing, constructing and evolving distributed software. A rigorous architectural approach dictates formalisms and techniques that are compositional, components that are context independent and systems that can be constructed and evolved incrementally. This paper overviews some of the underlying reasons for adopting an architectural approach, including a brief "rational history" of our research work, and indicates how an architectural model can potentially facilitate the provision of self-managed adaptive software system.

Keywords adaptive systems, self-managed systems, autonomic systems, software architecture

1 Software Architecture Approaches

Distributed processing offers the most general, flexible and promising approach for the provision of computing services. It offers advantages in its potential for improving availability and reliability through replication; performance through parallelism; and sharing and interoperability through interconnection.

Studies in software maintenance for distributed systems have indicated that the general move to distribution contributed to the simplification of the primitive software components used in distributed systems. However, this benefit is often overwhelmed by the increased complexity of the overall system. There is therefore a need to deal with issues such as component interaction and composition, design complexity, system organisation and reasoning. Rigorous use of a software architecture offers much potential benefit in providing a framework or skeleton with which to deal with these issues.

Software architecture descriptions aim to specify system structure at a sufficiently abstract level to deal with large and complex systems yet be sufficiently detailed to support reasoning about various aspects and properties. Architectures are generally defined hierarchically, as compositions of interconnected components. A component type is defined in a context-independent manner in terms of its communication interface: the services it *provides* to other components and the services it *requires* in order to perform its functionality.

Composite components are defined in terms of their constituent components (other primitive or composite components) and the bindings between these. Services provided internally are bound to an interface service provision so as to be available externally. Service requirements which cannot be satisfied internal to the composite component are made visible at its interface. Thus architectural descriptions support abstraction by hierarchical decomposition and encapsulation. The purpose of an Architectural Description Language (ADL) is to facilitate provision of precise software architecture descriptions, and to provide associated reasoning and/or software construction support.

A software architecture can be used as a model in much the same way as other engineers build models to check particular aspects of a system design. We believe that an ADL should be essentially structural and sufficiently abstract to support multiple views. These views can be presented as elaborations of the shared architectural structure. For instance, for behaviour modelling and reasoning, these elaborations add the particular component behaviour and interaction details of interest to the underlying structure. It is essential that the behaviour modeling is compositional so as to support analysis of the system behaviour and verification of its properties as the (parallel) composition of its component behaviours. For system construction, the architecture is elaborated with the necessary

Regular Paper

Much of the research has been supported by the Engineering and Physical Sciences Research Council and is currently partly supported by EPSRC Platform grant AEDUS 2 and a DTC grant.

implementation details. It can then be used to compose component implementations so as to construct and interconnect the particular distributed system. Thus the *same* instantiated architecture can be used for multiple aspects such as both behaviour modelling and system construction. Having a common architectural structure helps to preserve consistency between the various models and the system itself. Another important aspect of the ADL is the need to support variation in the form of system families. An architecture is a general description which, on instantiation, is tailored to produce a system instance which represents a member of the architectural family.

2 Rigorous Architectural Approach: A Brief History

As described in [1], our work can be divided into three overlapping phases. Firstly, the use of a declarative *explicit* architecture characterises our work on configuration programming. The prototype distributed system Conic[2,3] included the ability to specify, construct and dynamically evolve a distributed software system[2,4], using a configuration language to explicitly compose software components[5,6]. Work on the general purpose ADL, Darwin[7−9], and its industrial instantiation, Koala[10], followed, providing a sound structural language and facilities for variations respectively. The second phase focused on *modelling* in an architectural framework. The aim is to analyse systems as structural compositions of their constituent components' behaviour. This led to work with labelled transition systems (LTS), the process algebra, FSP (Finite State Processes)[11] and construction of the model checker, LTSA[12−15]. Model animation and model synthesis from scenarios[16] has enriched this vein of research.

It is our experience that software architecture descriptions at an appropriate level of abstraction seem to be crucial even during the requirements specification process. Requirements are often not fully elaborated or even understood before a (hypothetical) solution architecture is developed. The architecture often helps to raise new issues and requirements. It is important that the architecture should be stable, representing the essential core aspects of the system structure which do not change radically during software development. System evolution is then seen as a combination of minor changes to or replacement of primitive components, or major changes to composite components and hence the system structure. We believe that pure top-down design and refinement are essentially impractical except for very constrained well-understood parts of application domains. Design decomposition and compositional analysis and reasoning go hand-in-hand. They should be performed iteratively and incrementally, with automated compositional modelling techniques being used to provide the necessary feedback to designers to help correct errors and raise confidence in their designs. This experience has been gained over many years, working initially with industry such as British Coal and British Petroleum, and more recently with Philips and others.

However, some software systems are particularly difficult to construct, manage and analyse as they tend to be highly dynamic. Current ADL descriptions tend to be largely static and can describe only restricted forms of dynamically changing structures. In such circumstances, architectures should impose constraints on the kinds of components that can be integrated into the system and on the interactions that can take place. This is a difficult area that requires further research and experimentation, but is crucial if we are to be able to manage and reason about systems such as those of the scale, diversity, complexity and dynamism constructed from Web services. The goal is to provide support for dynamic, self-managing and adaptive systems that automatically reconfigure themselves to accommodate dynamically changing context and requirements without human intervention.

3 Self-Managed Adaptive Software

The challenge is to provide software systems in such a way that they are robust in the presence of major issues such as change and complexity.

Change is inherent, both in the changing needs of users and in the changes which take place in the operational environment of the system. Hence it is essential that our systems can adapt as necessary to continue to achieve their goals. Change is also induced by failures or the unavailability of parts of the system. It is therefore necessary to envisage dynamically changing configurations of software components so as to adapt to the current situation. Dynamic change, which occurs while the system is operational, requires that the system evolves dynamically, and that the adaptation occurs at run-time.

Complexity requires that we use rigorous techniques to design, build and analyse our software and thereby avoid unnecessary design flaws. This implies the need for analytical techniques which cope with changing goals and the changing compositions of adaptive software.

Since the complexity and response times required by the changes may not permit human intervention, we must plan for automated management of change.

The systems themselves must be capable of determining what system change is required, and in initiating and managing the change process wherever possible. This is the aim of self-managed systems.

Self-managed systems are those that are capable of adapting as required though self-configuration, self-healing, self-monitoring, self-tuning, and so on, which are also referred to as self-$*$ or autonomic systems.

The aim of self-configuration is that the components should either configure themselves such that they satisfy the specification of the goals, properties and constraints that you expect your system to achieve or be capable of reporting that they cannot. If the system suffers from changes in its requirements specification or operational environment such as changes in use, changes in resource availability or faults in the environment or in parts of the system itself, then 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. Note that the specifications should 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.

Different research communities are 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[17] and more recently, the SelfMan Workshop 2006[18] to discuss and analyse the potential of self-$*$ systems for managing and controlling networked systems and services. Dobson *et al.*[19] provide a recent survey on autonomic communications, and 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 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[20−22] and more recently with WOSS (Workshop on Self-Healing and Self-Managed Systems)[23,24] and SEAMS (Software Engineering for Adaptive and Self-Managing Systems)[25]. Other interested research communities include the intelligent agent, machine learning and planning communities. Huebscher and McCann[26] provide an excellent and comprehensive survey on autonomic computing.

However, although research has provided much that is useful in contributing towards self-management, the general and fundamental issues of providing a comprehensive and integrated approach remains.

Not surprisingly, we believe that an architecture-based approach offers potential benefits such as generality, where the underlying concepts and principles should be applicable to a wide range of application domains, each with its own peculiar architecture. Architectures are also designed to handle issues of scalability and complexity — by supporting abstraction and separation of concerns.

Many others also advocate the use of a component-based architectural approach. For instance, Oreizy *et al.*[27] provide a general outline of an architectural approach which includes adaptation and evolution management; Garlan and Schmerl[28] 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 runtime adaptation and self-healing in ArchStudio[29]; and Castaldi *et al.*[30] 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.

As mentioned, our own work has concentrated on the use of ADLs for software design and implementation from components[9], including limited language support for dynamic change[10], a general model for dynamic change and evolution[4], associated analysis techniques[31] and initial steps towards self-management[32].

4 An Architectural Model for Self-Management

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. More details of our approach can be found in [33, 34]; we provide a summary below.

Based on the work by Gat[35] in robotics, we propose the use of a three-layer reference architecture (see Fig.1). This provides the necessary separation of concerns for a rigorous engineering approach in which low-level actions can be clearly and formally related to high-level goals that are precisely specified. In addition to the separation of concerns, one of the criteria for placing functionality in different layers in our self-managed systems architecture is that of time. Immediate feedback actions are at the lowest level and the longest actions requiring deliberation are at the uppermost level.

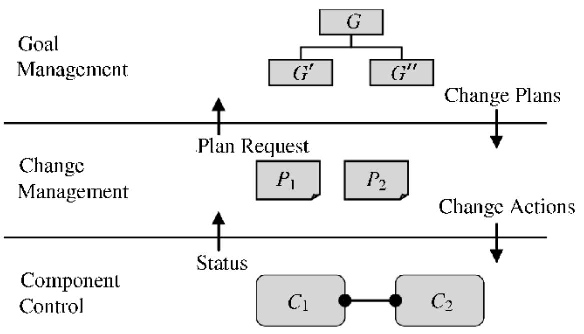

Fig.1. Three-layer architecture model for self-management.

4.1 Component Control

The bottom layer is *Component Control*, which consists of the set of interconnected components that accomplish the application function of the system. It includes facilities to report the current status of components to higher layers, to adjust the operating parameters of components and for modification by component creation, deletion and interconnection. An important characteristic of this layer is that, when a situation arises that the current configuration of components is not designed to deal with, this layer detects this failure and reports it to higher layers.

The main research challenge at this level of a self-managed architecture is concerned with preserving safe application operation during change. The algorithm outlined in [4], which includes the component responsibilities of change management, 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[31], a problem promisingly addressed by Zhang and Cheng[36].

4.2 Change Management

The middle layer is *Change Management* which 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. It consists of a set of reactive plans, each giving an action or sequence of actions to handle the new situation. It changes the component configuration from one architecture to another by recognizing which components are needed to accomplish the current plan of actions, where action(s) indicate which component is appropriate and required (i.e., provides a mapping from plan actions to components). This layer can introduce new components, recreate failed components, change component interconnections and change component operating parameters. 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.

One of the major research challenges at this level is dealing with distribution and decentralization. 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.

4.3 Goal Management

The uppermost layer is *Goal Management* which is responsible for change planning. This takes the current state and a specification of a high-level goal and attempts to produce a plan to achieve that goal. This layer produces change management plans in response to requests from the layer below and in response to the introduction of new goals.

There are many research issues at this layer, such as how to provide a precise domain model in which formulate plans, 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. The initial problem is to have a precise specification of the domain which is sufficiently general yet provides the constraints introduced by the environment. The goals required of the system 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[37]. The challenge is to achieve goal specification such that it is both comprehensible by human users and machine readable. The likelihood is that the more domain specific this layer is, the more automated plan synthesis can be, even if this is at the expense of generality and flexibility.

5 Conclusion

We have presented our background experience and views as to why architectural concepts seem to provide a promising basis for an approach to the provision of self-managed adaptive software systems. We have also presented an outline of our three-layer architecture model. 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. It also provides a context for discussing some of the main research challenges which self-management poses. There has been much progress towards providing self-managed systems, but much remains to be done to provide an integrated and comprehensive solution, supported by an appropriate infrastructure.

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. Furthermore, we have not discussed non-functional properties such as performance modelling. It would seem that, in order to perform realistic performance modelling, there needs to be sufficient detail as to the actual performance of implemented components, their allocation, resource conflicts and interaction properties and delays. This may well mean that anything other than crude response estimates and performance analysis is just not possible at an abstract architectural level. However even such crude indications of feasibility are useful. We are currently interested in extending our architectural behaviour models to handle probabilistic models.

Nevertheless, we believe that a layered architectural approach offers a suitable separation of concerns and a promising framework for working on each of the relevant research issues.

Acknowledgements We would like to acknowledge the contribution of our co-workers in the Distributed Software Engineering group at Imperial College who over the years have contributed hugely to the work outlined and referenced in this paper.

References

[1] Kramer J, Magee J. Engineering distributed software: A structural discipline. In *Proc. the 10th European Software Engineering Conference* held jointly with *the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal, Sept. 2005, pp.283–285.

[2] Kramer J. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 1985, 11(4): 424.

[3] Magee J. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 1989, 15(6): 663.

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

[5] Kramer J. Configuration programming — A framework for the development of distributable systems. In *Proc. International Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, May 1990, pp.374–384.

[6] Kramer J, Magee J, Ng K. Graphical support for configuration programming. In *Proc. International Conference on System Sciences*, Hawaii, USA, Jan. 1989, pp.860–870.

[7] Magee J, Dulay N, Kramer J. Regis: A constructive development environment for parallel and distributed programs. *Distributed Systems Engineering Journal, Special Issue on Configurable Distributed Systems*, 1994, 1(5): 304–312.

[8] Magee J *et al.* Specifying distributed software architectures. In *Proc. the 5th European Software Engineering Conference* (*ESEC'95*), Sitges, Spain, Sept. 25–28, 1995, pp.137–153.

[9] Magee J, Kramer J. Dynamic structure in software architectures. In *Proc. the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE 4*), San Francisco, California, USA: ACM Press, 1996, pp.3–14.

[10] van Ommering R. The Koala component model for consumer electronics software. *Computer*, 2000, 33(3): 78.

[11] Magee J, Kramer J. Concurrency: State Models & Java Programs. Chichester: Wiley, xviii, 2006, p.413.

[12] Magee J, Kramer J, Giannakopoulou D. Behaviour analysis of software architectures. In *Proc. the 1st Working IFIP Conference on Software Architecture* (*WICSA1*), San Antonio, TX, USA, Feb. 1999.

[13] Cheung S C, Kramer J. Checking subsystem safety properties in compositional reachability analysis. In *Proc. the 18th International Conference on Software Engineering* (*ICSE'18*), Berlin, Germany: IEEE Computer Society Press, March 25–26, 1996.

[14] Giannakopoulou D, Magee J, Kramer J. Checking progress with action priority: Is it fair? In *Proc. the 7th European Software Engineering Conference* held jointly with *the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE'99*), Toulouse, France: Springer, Sept. 6–10, 1999, pp.511–527.

[15] Giannakopoulou D, Magee J. Fluent model checking for event-based systems. In *Proc. ESEC/SIGSOFT FSE*, Helsinki, ACM, Finland, Sept. 1–5, 2003, pp.257–266.

[16] Uchitel S. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 2004, 13(1): 37.

[17] *Proc. the 3rd IEEE International Conference on Autonomic Computing*. Dublin: IEEE, Nov. 29, 2006.

[18] *Proc. the 2nd IEEE Int. Workshop on Self-Managed Networks, Systems and Services* (*SelfMan 2006*). Dublin: IEEE, June 16, 2006.

[19] Dobson S *et al.* A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 2006, 1(2): 223–259.

[20] *Proc. IEE/IFIP 1st Int. Workshop on Configurable Distributed Systems* (*CDS 92*). London, UK, May 1992.

[21] *Proc. IEEE 2nd International Conference on Configurable Distributed Systems* (*CDS 94*). Pittsburgh, USA, May 1994.

[22] *Proc. IEEE 3rd International Conference on Configurable Distributed Systems* (*CDS 96*). Annapolis, USA, May 1996.

[23] *Proc. the First Workshop on Self-Healing Systems*. Charleston, South Carolina: ACM Press, 2002.

[24] *Proc. the 1st ACM SIGSOFT Workshop on Self-Managed Systems*. Newport Beach, California: ACM Press, 2004.

[25] *Proc. 2nd IEEE Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems* (*SEAMS 2007*), ICSE, Minneapolis, 2007.

[26] Huebscher M C, McCann J. A survey of autonomic computing — Degrees, models and applications. *ACM Computing Surveys*, 2008, 40(3): 1–28.

[27] Oreizy P *et al.* An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE* [see also *IEEE Intelligent Systems*], 1999, 14(3): 54–62.

[28] Garlan D, Schmerl B. Model-based adaptation for self-healing systems. In *Proc. the First Workshop on Self-Healing Systems*, ACM Press: Charleston, South Carolina, USA, 2002, pp.27–32.

[29] Dashofy E M, A van der Hoek, Taylor R N. Towards architecture-based self-healing systems. In *Proc. First Workshop on Self-Healing Systems*, ACM Press: Charleston, South Carolina, 2002, pp.21–26.

[30] Castaldi M *et al.* A light-weight infrastructure for reconfiguring applications. In *Proc. the 11th Software Configuration Management Workshop* (*SCM03*), Portland, Oregon, *LNCS*, 2003, pp.231–244.

[31] Kramer J, Magee J. Analysing dynamic change in distributed software architectures. In *IEE Proc. Software*, 1998, 145(5): 146–154.

[32] Kramer J, Magee J. Self-Managed Systems: An Architectural Challenge. *IEE Proc. Software*, 145(5): 146–154.

[33] Georgiadis I, Magee J, Kramer J. Self-organising software architectures for distributed systems. In *Proc. the First Workshop on Self-Healing Systems*, ACM Press: Charleston, South Carolina, 2002.

[34] Sykes D *et al.* Plan-directed architectural change for autonomous systems. In *Proc. the Sixth International ACM Workshop on Specification and Verification of Component-Based Systems* (*SAVCBS'07*), Dubrovnik, Croatia, Sept. 3–4, 2007, pp.15–21.

[35] Gat E. Three-Layer Architectures. Artificial Intelligence and Mobile Robots, MIT/AAAI Press, 1997.

[36] Zhang J, Cheng B H C. Model-based development of dynamically adaptive software. In *Proc. the 28th International Conference on Software Engineering*, ACM Press, Shanghai, China, 2006.

[37] van Lamsweerde A. Goal-oriented requirements engineering: A guided tour. In *Proc. the 5th IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, 2001.

Jeff Kramer is a professor of computing and dean of the Faculty of Engineering, Imperial College London. His research interests include requirements engineering, design and analysis methods, software architectures and software development environments, particularly as applied to concurrent and distributed software. He was principal investigator in various research projects that led to the development of the CONIC and DARWIN environments for distributed programming and the associated research into software architectures and their analysis. The work on the Darwin Software Architecture led to its commercial use by Phillips in their next generation of consumer television products. 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 in recognition of his research contribution in Software Architecture and 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 fellow of the Royal Academy of Engineering, fellow of the IET, fellow of the BCS, fellow of the City and Guilds of London and fellow of the ACM.

Jeff Magee is a professor of computing and is currently both head of the Department of Computing and deputy principal of the Faculty of Engineering at Imperial College London. His research is primarily concerned with the software engineering of distributed systems, including design methods, analysis techniques, operating systems, languages and program support environments for these systems. His work on software architecture led to the commercial use by Phillips of an architecture description language based on Darwin in their current generation of consumer television products. He is the author of over 100 refereed publications and has co-authored a book on concurrent programming entitled "Concurrency — State Models and Java Programs". He was co-editor of the IEE Proceedings on Software Engineering and a TOSEM associate editor. He was program co-chair of the 24th International Conference on Software Engineering and chaired the ICSE Steering Committee from 2002∼2004. He was a member-at-large of the ACM SIGSOFT committee from 2002∼2005. He was awarded the BCS 1999 Brendan Murphy prize for the best paper in distributed systems and the IEE Informatics Premium prize for 1998/99 for a paper jointly authored with Professor Kramer on software architecture. He is the co-recipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in distributed software engineering.