Distributed Software Engineering: a Rigorous Architectural Approach

Jeff Kramer
Department of Computing, Imperial College London
j.kramer@imperial.ac.uk

Abstract

The engineering of distributed 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 extended abstract overviews some of the underlying reasons for adopting this architectural approach and provides a brief "rational history" [1] of our research work, together with some selected references.

1. Why do we need architecture descriptions?

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 has 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.

2. How can architecture descriptions help?

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.

3. What form should an ADL take?

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 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. For system construction, the architecture is elaborated with the necessary 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 aspects such as 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.

4. A brief history of our approach

As described in [1], our work could be roughly divided into three overlapping phases. Firstly, the use of a declarative *explicit* architecture characterises our work on configuration programming. The prototype distributed system Conic [2,3,4] included the ability to specify, construct and dynamically evolve a distributed software system [2,5], using a configuration language to explicitly compose software components [6,7]. Work on the general purpose ADL, Darwin [8,9,10], and its industrial instantiation, Koala [11], 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) [13] and construction of the model checker, LTSA [14,15,16,17]. Model animation and model synthesis from scenarios [18] has enriched this vein of research. Our current work, is concerned with *implicit* structural specifications. The aim is to generate and check structures which satisfy constraints that can be imposed both statically and dynamically. We believe that this is needed in realising self-organising systems that both automatically configure themselves and subsequently reconfigure themselves to accommodate dynamically changing context and requirements without human intervention [19,20].

5. What was our general experience?

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.

6. What are some of the outstanding problems?

Some distributed software environments 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. As mentioned, the goal is to provide support for self-organising systems.

Another aspect which seems to be particularly difficult at the architectural level is the association of 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.

7. Acknowledgements

I would like acknowledge the contribution of my 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 extended abstract. In particular, I have worked closely with Jeff Magee for over 25 years. Much of the research has been supported by the Engineering and Physical Sciences Research Council and is currently partly supported by EPSRC grant READS GR/S03270/01.

8. Selected References to our work

[1] Kramer, J. and Magee J., Engineering Distributed Software: a Structural Discipline. ESEC/FSE '05 (10th European Software Engineering Conference (ESEC) / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)), Lisbon, Portugal: 283-285.

[2] Kramer J. and Magee J., Dynamic Configuration for Distributed Systems, IEEE Trans. on Software Eng., SE-11 (4), (1985), 424-436.

[3] Magee J., Kramer J., and Sloman M.S., Constructing Distributed Systems in Conic, IEEE Trans. on Software Eng., SE-15 (6), (1989), 663-675.

[4] Kramer J., Magee J. and Finkelstein A., A Constructive Approach to the Design of Distributed Systems, (10th IEEE Int. Conf on Distributed Computing Systems) Paris, (1990), 580-587

[5] Kramer J. and Magee J., The Evolving Philosophers Problem: Dynamic Change Management, IEEE Trans. on Software Eng., SE-16 (11), (1990), 1293-1306.

[6] Kramer J., Magee J. and Ng K., Graphical Configuration Programming, IEEE Computer, 22 (10), (1989), 53-65.

[7 Kramer J., Configuration Programming - A Framework for the Development of Distributable Systems, (IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90)), Tel-Aviv, Israel, (1990), 374-384.

[8] Magee J., Dulay N. and Kramer J., Regis: A Constructive Development Environment for Distributed Programs, Distributed Systems Engineering Journal, 1 (5), Special Issue on Configurable Distributed Systems, (1994), 304-312.

[9] Magee J., Dulay N., Eisenbach S., Kramer J., Specifying Distributed Software Architectures, (5th European Software Engineering Conference (ESEC '95), Sitges, September 1995), LNCS 989, (Springer-Verlag), 1995, 137-153.

[10] Magee J. and Kramer J., Dynamic Structure in Software Architectures, (4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4)), San Francisco, (October 1996), SEN, Vol.21, No.6, November 1996, 3-14.

[11] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. Computer 33, 3 (2000), 33-85.

[12] Magee J. and Kramer J., Composing Distributed Objects in CORBA, in Information Systems Interoperability, Kramer B., Papazoglou M. and Schmidt H., Research Studies Press / John Wiley & Sons Inc., England, 1998.

[13] Magee J. and Kramer J., Concurrency: State Models and Java Programs, Wiley 1999; 2006 2nd Edition.

[14] Magee J., Kramer J. and Giannakopoulou D., Behaviour Analysis of Software Architectures, First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, 22-24 February 1999, pages 35 –50.

[15] Cheung S.C. and Kramer J., Checking Subsystem Safety Properties in Compositional Reachability Analysis, (18th IEEE Int. Conf. on Software Engineering (ICSE-18), Berlin, 1996), 144-154.

[16] Giannakopoulou D., Magee J. and Kramer J. Checking progress with Action Priority: Is it Fair?, ESEC / SIGSOFT FSE 1999, LNCS 1687, p511-527

[17] Giannakopoulou D. and Magee J., Fluent model checking for event-based systems. ESEC / SIGSOFT FSE 2003: 257-266.

[18] Uchitel S., Kramer J. and Magee J., Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM Trans. Softw. Eng. Methodol. TOSEM 13(1): 37-85 (2004).

[19] Georgiadis I., Magee J. and Kramer J.: Self-organising software architectures for distributed systems. ACM WOSS 2002, p 33-38.

[20] Chatley R., Eisenbach S., Kramer J., Magee J., Uchitel S., Predictable Dynamic Plugin Systems. FASE 2004, p129-143.