Engineering Distributed Software: a Structural Discipline

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

ABSTRACT
The role of structure in specifying, designing, analysing, constructing and evolving software has been the central theme of our research in Distributed Software Engineering. This structural discipline 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 our development of a structural approach to engineering distributed software and gives indications of our future work which moves from explicit to implicit structural specification. With the benefit of hindsight we attempt to give a "rational history" to our research.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques
D.2.2 [**Software Engineering**]: Software/Program Verification
D.2.11 [**Software Engineering**]: Software Architectures

General Terms
Design, Verification.

Keywords
Distributed software engineering, software architecture, structure, software components, configuration programming, dynamic configuration, evolution.

1. INTRODUCTION
Separation of structural concerns has its origins in the seminal 1975 paper by deRemer and Kron [1] which made the distinction between *programming-in-the-small* and *programming-in-the large* and advocated the need for a separate module interconnection language. The structural view, we advocate, puts structure at the heart of software architecture. Structure, as a separable perspective, relies on sound techniques for composition — whether of software components or specifications of the behaviour of these components. These complementary concerns of structure and composition are the themes we address here.

With hindsight, our work can be divided into three overlapping phases. Firstly, explicit structure characterises our work on configuration programming. The prototype distributed system Conic included the ability to specify, construct and dynamically evolve a distributed software system, using a configuration language to explicitly compose software components. Work on the general purpose ADL Darwin and its industrial instantiation, Koala, followed. The second phase, focused on modelling in a structural framework. The aim was to analyse systems as structural compositions of their components' behaviour. This led to work with labelled transition systems (LTS), the process algebra, FSP (Finite State Processes) and construction of the model checker, LTSA. Model animation and model synthesis from scenarios 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.

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

2. EXPLICIT STRUCTURE
The Conic System [2,3] exhibited a separate structural language that we referred to as a configuration language. This explicitly described the structure of the system to be constructed as a set of component types, the instances of these types and the interconnections between instances. Conic provided a test-bed for exploring ideas on system design [4], system construction and evolution, including dynamic configuration [2,5] which exploited an explicit structural description. For example, dynamic configuration was performed by editing operations on the elaborated system description which directed changes to the running system.

The principles underlying the approach which we termed Configuration Programming [6,7] are outlined below:

- The *configuration language* used to describe structure is separate from the language used to program basic components.
- Components are defined as *context independent* types with well-defined interfaces that define both services provided and services required.
- Complex components are described as a *composition* of instances of component types using the configuration language.
- *Change* is expressed at the configuration level, as changes to component instances and/or their interconnections.

The Conic configuration language retained a number of dependencies on the programming language (Pascal) used to programming components. In particular, it assumed interfaces typed in Pascal that used a fixed set of interaction primitives. A deliberate effort was made to remove these language dependencies in the configuration language we used in our next prototype distributed systems — Rex followed by Regis [8]. These design efforts eventually resulted in the final form of the Darwin language which was parameterised by the interface type system and, as a result, represented a purely structural description. It was at this point that we realised that the objectives of Configuration Programming given its focus on structure were very much in line with those of Software Architecture [9,10] and Darwin came to be known as an Architecture Description Language ADL[11,12]. Through a very fruitful collaboration with Philips, Darwin gave birth to Koala [13], an ADL used successfully to construct the software for a line of television products.

3. MODELLING

Darwin was initially used to describe and generate distributed software systems, with components providing and requiring services at their interfaces and with implementation elaborations for the primitive components. Service were typically accessed by a Remote Method call mechanism such as that provided by CORBA[14]. However, Darwin proved to be sufficiently abstract to support multiple views (cf. Kruchten [15]). Two important views are the service view (for construction) and the behavioural view (for behaviour analysis) (see Figure 1). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation.

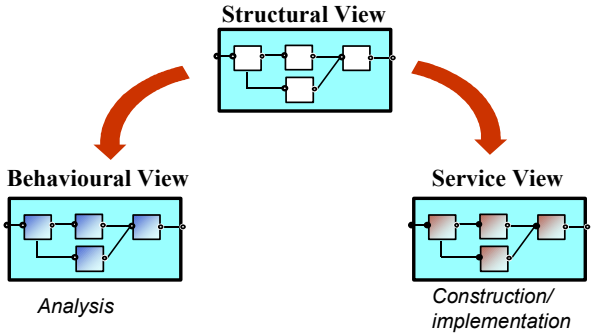

Figure 1. Common Structural View with Service and Behavioural Views

We can easily restate the four principles outlined before, in the context of description and construction, in the specification and analysis context as follows:

- The *configuration language* used to specify structure is separate from the language used to specify basic component behaviour.
- Component specifications define the behaviour *visible* at the component interface.
- Complex components are described as a *composition* of more basic component behaviours.
- *Change* is expressed at the configuration level, as changes to component specification interconnection.

We use Labelled Transitions Systems (LTS) to specify and (predominately) Compositional Reachability Analysis to analyse the behaviour of systems. These LTSs are generated from specifications in the process algebra FSP [16]. Basic components are described using the dynamic combinators – action prefix, choice and recursion while compositions of basic components are modelled using the static combinators – parallel composition and relabelling. This rather elegant separation found in process algebras allows Darwin structural descriptions to be directly translated into FSP composition specifications using parallel composition for component instantiation and relabelling for component interconnection/binding [17].

We have focused on making modelling and analysis accessible to practising software architects. This has led to work on how to specify and check properties [18,19,20] and more recently on generating initial behavioural specifications from Message Sequence Charts (MSC) gathered during requirements capture [21].

4. IMPLICIT STRUCTURE

Our initial work permitted the specification of 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 [2]. A change to the software architecture can 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 can be performed by a configuration manager 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 is however too restrictive for current dynamic, open systems.

In our current work, 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 a structural approach? In addition to the autonomy inherent in self-organizing systems, we wish to retain the benefits of an overall structural 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 an explicit description of component instances and their interconnection but rather a set of constraints on the way components may be composed – an implicit structural specification. This implicit specification can be used to generate and/or check a specific architectural instance for conformance.

Furthermore, if a disturbance occurs, correcting changes can be generated.

Currently, we have built some decentralised [22] and centralised systems [23] that utilise the idea of implicit structural spepcification and have looked at analysing strucural constraints. However, there are many open questions that remain in building large and scalable systems and in specifying and analysing the behaviour of such systems.

5. ACKNOWLEDGMENTS

The authors would like acknowledge their co-workers in the Distributed Software Engineering group at Imperial College who over the years have contributed hugely to the work we have outlined and referenced in this extended abstract. 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.

6. REFERENCES

[1] DeRemer F.and HH Kron, *Programming-in-the-large versus Programming-in-the-small*, IEEE Trans. on Software Engineering, 2(2): 80-86, June 1976.

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] Kruchten, P.: *The 4+1 view model of architecture*. IEEE Software 12(6), IEEE Computer Society Press (1995) 42—50.

[16] Magee J. and Kramer J., *Concurrency: State Models and Java Programs*, Wiley 1999.

[17] 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.

[18] 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.

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

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

[21] 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)

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

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