EXOSKELETAL SOFTWARE: MAKING STRUCTURE EXPLICIT

J Kramer

Rapporteurs: Rogério de Lemos and Cecília M.F. Rubira

EXOSKELETAL SOFTWARE - making structure explicit

Jeff Kramer

**Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.
(jk@doc.ic.ac.uk)**

Hypothesis: *Consider the humble crab. Ignoring its culinary appeal, the crab provides us with an example of a working system which explicitly flaunts its structure, both in terms of components and interconnections. One can readily comprehend its operation from its exoskeletal architecture. Each part provides at its interface a clear indication of the allowable interactions with other parts yet hides its internal workings. Like the crab, it is possible for many software systems to benefit from an explicit and visible architecture, not only during design and analysis but also in the constructed system.*

1. The general area: Distributed Software Engineering

Distributed processing provides the most general, flexible and promising approach for the provision of computer processing. Interconnected workstations are widely used for local processing, to support user communication for interaction and cooperation, and to provide access to shared facilities and information. Conventional and special purpose processors are interconnected to support a wide range of applications from chemical plants to cars, from stock market trading to campus meal reservations.

Why are distributed systems so attractive? The answers are as multifarious as the applications. The users of computers, the information they require and provide and the applications themselves are often physically distributed. To match these needs, both the hardware and software can be designed and constructed in a flexible modular fashion - as interconnected and interacting components. Particular resources, services and information can be accessed across the network and shared among the system users. Some seek to exploit the potential for improved availability by the use of replication and the removal of single failure points. Others seek performance gains by improving the response time through local processing or the throughput by the use of parallel processing. Thus distributed computing offers advantages in its potential for improving availability and reliability through replication; performance through parallelism; sharing and interoperability through interconnection, and flexibility, incremental expansion and scalability through modularity.

However, to gain these benefits, we must cope with the issues that distributed computing raises. The interactions between the concurrent components give rise to issues of non-determinism, contention and synchronisation. Component separation and autonomy gives rise to issues of partial information and partial failure. These issues demand that we adopt effective engineering methods and tools. Our techniques must avoid constraining the resultant software unnecessarily by the use of conventional sequential or centralised designs but take cognizance of and exploit the component-based nature of these systems. Software engineering itself must be extended and adapted to address these distribution issues: hence Distributed Software Engineering (DSE).

Construction

The need to construct these systems as interacting components can be considered a blessing which forces software engineers towards compositional techniques which offer the best hope for constructing scalable and evolvable systems in an incremental manner. There is some consensus on the mechanisms which should be used to support component interaction. Among the mechanisms agreed as useful are: communication by remote procedure call, atomic group cast communication between server replicas to support

availability and atomic actions to preserve data consistency in the presence of failures. This consensus is reflected in the inclusion of most of the above in commercially available distributed applications platforms such as ANSA and OSF/DCE. However, none of the above mechanisms help in the design process of decomposing an overall application into a set of components nor in its subsequent construction. They are rather the "glue" used to compose components such that they may communicate and interact. What is missing is any notion of *structure*. Support is missing for the design and construction of applications exhibiting a structure which is any more complex than a simple client-server arrangement.

Software Architecture
Software systems should be designed using a combination of models, such as structural models to convey the overall architecture as interacting components, computational models to describe component behaviour, and interaction models to describe the forms of communication and synchronisation. Hierarchical composition is used to form subsystems and finally the system itself. Model precision varies greatly from the informal, such as data flow diagrams, to more recent formalisms, such as the π-calculus, which can be used for expressing and reasoning about particular architectures. The key ingredient here is composition: the ability to check consistency of interconnection and interaction and the ability to infer and analyse composite behaviour. Composition of behaviours is the means by which we can gain greater confidence in the adequacy of our designs in meeting their system specifications.

Furthermore, the specification of system structure or architecture can be used to generate and manage the system itself. Hence, system structure (architecture), separately and explicitly described, should be recognised as the unifying framework upon which to hang the specification, design, construction and evolution of systems. Such architectural models are more abstract in the upstream activities such as requirements specification, but become more specific and concrete downstream in design and coding. Unfortunately, in practice, the description and retention of the architectural information becomes more and more *implicit* as we move from design through coding into execution and maintenance. For instance, components (such as objects in OOP) generally make direct calls to others thereby obscuring their interactions. Component instantiation is often embedded in the code of various other components.

Why tolerate this loss of architectural information? One would expect that, since this information is so useful to system integrity, comprehension, construction and management, it should be explicit and visible, being retained in the design, code and possibly even in the running system. Our aim is to make exoskeletal software so that its architecture is as obvious as that of the crab.

2. The particular approach: a Configuration Language (Darwin)

The premise of our approach is that a separate, explicit structural (configuration) description is essential for all phases in the software development process for distributed systems, from system specification as a configuration of component specifications to evolution as changes to a system configuration. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify and design systems, and can be used directly by construction tools to generate the system itself. In many cases - particularly embedded applications - it is the structure of the application itself which is used to dictate the structure of the resultant system. This approach, initially under the guise of simple "module interconnection languages" (MIL) and subsequently as "configuration languages" provides generalised to support for a wide variety of component and interaction types.

We use the neutral term "component" to mean a software entity which encapsulates some resources and provides a well defined interface in terms of the operations it *provides* to

access the resources and the operations it *requires* to implement its functionality. Further, we require our components to be "context independent" in that they use only local names to communicate with their environment, thereby allowing them to be developed independently of the context in which they execute. Context independence makes it easy to plug a component into different programs since it specifies both the communication objects required as well as those provided.

Design

In contrast to the "specification driven" approach, our approach to distributed systems design is "constructive". The "specification driven" approach attempts to formalise the decomposition process based only on the system specification. We believe that this process of component identification remains informal as it requires design information not usually included in the system specification. Decomposition is best dealt with through design heuristics. Emphasis should rather be placed on the validation process using analysis through composition of component behaviours analogous to "construction" of the system from components. This constructive approach is the means by which we gain confidence that our design is satisfactory. Our design approach is thus not restricted to a specific design method or technique such as SASD or OOD in the sense of enforcing those specific rules and method steps. Rather it supports a general approach to design consisting of the following general design activities:

- *Structure and Component Identification:* Initial design aims to identify the main processing components and produce a structural description indicating the main data flows.
- *Interface Specification:* This aims at introducing control (synchronisation) between components and refining the configuration, component interface specifications (intercommunication) and component descriptions accordingly.
- *Component Elaboration* consists of elaboration of the component types, either by hierarchical decomposition of composite component types into a configuration of subcomponents, or by functional description of behaviour (formal or informal specifications). Primitive components must also be provided with implementation (code) descriptions. As before, the identification of common component types is emphasised.

Although this description emphasises the top-down approach, bottom-up (constructive) composition and component reuse can be used at any stage.

Construction

The design activity culminates in a structural description of the desired system and a set of primitive component types described in a programming language. From these, the executing distributed system can be constructed by invoking the appropriate compilation, linking and loading tools. Central to the construction activity is the structural description. It can be annotated with non-functional information such as location, availability and resource requirements during the design process to direct the construction phase.

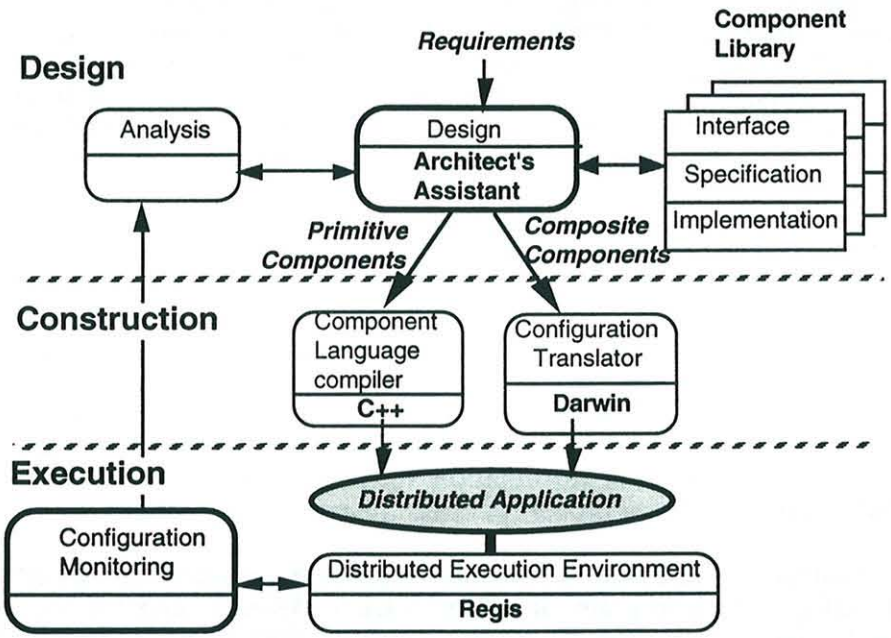
For a number of years, the Conic environment supported the use of an explicit configuration language ("configuration programming") for distributed system construction. A number of other research projects make use of a separate configuration language for distributed systems construction (Durra, Lady, Polylith) but few are as widely distributed and used, and as simple yet versatile as the Conic configuration language which included facilities for hierarchic definition of composite components, for parametrisation of components, for replication of both component instances and interface ports, for conditional configurations with evaluation of guards at component instantiation, and even for recursive definition of components. This work has been extended to include dynamic and generic structures in our new configuration language, Darwin. Darwin descriptions compile to C++ procedures which elaborate the required structures at system generation/instantiation time. Darwin permits the definition of both static and dynamically changing structures. Darwin is neutral with respect to the form of inter-component communication or interface specification used although it will allow these interfaces to be checked for compatibility. Constructed distributed systems run using our in-house platform for distributed computing, Regis.

Analysis

In order to exploit the central architectural view, we adopt compositional techniques analogous to system construction except that the entities being manipulated are not software components but associated attributes giving specifications or models of their functional or timing behaviour. The objective is to ensure that the structure of the system and of its specification is the same. System architects should be free to select those specification techniques that are appropriate to the application. Currently we have focused on the use of Milner's work on the π-calculus as a means for defining the semantics of Darwin. More recently, we have been working on the use of labelled transition systems (similar to state transition systems) to describe component behaviour. Reachability analysis is then provided using an approximate but tractable technique for flow analysis of distributed programs and an exhaustive compositional technique for reachability analysis. Both techniques are supported by automated software tools.

Tool Support

The architectural methodology described is supported by a graphical environment, the *System Architect's Assistant*, for the design and engineering of distributed systems (see figure). The SAA has the capability to manipulate structural (Darwin) descriptions in both graphical and textual forms and is intended to transform one to the other (currently only graphical to textual). Its functionality can be split into three complementary and interacting areas. These are support for design, analysis, and construction. The SAA is intended to act as a front end for specifying the structure and those component specification attributes selected by the designer. It should then be possible to invoke the relevant compositional analysis or verification tools associated with any or all of the provided attributes.



Integration using the System Architect's Assistant

3. Conclusions

Our approach is to adopt an architectural design methodology in which distributed systems are described, modelled and constructed in terms of their software structure. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify, design and analyse systems, and can be used directly by construction tools to generate the system itself. Darwin, Regis and the Architect's Assistant

are intended to provide an environment for supporting this methodology.

The specification of system *requirements* can be construed as a set of tests which the design and subsequent implementation are expected to satisfy. A *design* is a model of the system on which these tests (often specified as scenarios with specific required results) can be conducted. Test satisfaction strengthens confidence in the adequacy of the design, and test failure indicates particular inadequacies. It is thus essential that designs are testable. The level of the design model(s) is selected so as to support such testing and analysis yet also provide the basis for (efficient) implementation.

Design initially concentrates on the description of an *instance* of the architecture. This can be generalised to provide generic architectural types with the inclusion of component parameters, conditional instantiation and binding, iteration and recursion, and generic structures with component type parameters. The designer is obliged to ensure that the particular required instance can be generated from the general architectural description. This process of moving between the specific (usually described graphically) and the general (usually described textually) requires further work in both the methodology and in the support offered by the Assistant.

Could (generic) component types be analogous to having a generic architecture for the crab family (genus)? If, say, one of the parameters was the size of one claw, we could make it larger and stronger than the other and thus instantiate a fiddler crab!

Bibliography (available by anonymous ftp from dse.doc.ic.ac.uk)

Distributed Software Engineering:
Jeff Kramer, "Distributed Software Engineering", Proc. of 16th IEEE Int. Conf. on Software Engineering (ICSE-16), Sorrento, May 1994, 253-263.

Darwin/Regis:
Jeff Magee, Naranker. Dulay and Jeff Kramer, "Structuring Parallel and Distributed Programs", Proceedings of IEE 1st International Workshop on Configurable Distributed Systems, London, March 1992, 102-116.

Jeff Magee, Naranker Dulay, Jeff Kramer, "A Constructive Development Environment for Parallel and Distributed Programs", Proceedings of IEEE 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, March 1994, 4-14.

Architect's Assistant:
Jeff Kramer, Jeff Magee, Keng Ng and Morris Sloman, "The System Architect's Assistant for Design and Construction of Distributed Systems", Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, Sept. 1993, 284-290.

Analysis:
Shing Chi Cheung, Jeff Kramer, "Enhancing Compositional Reachability Analysis with Context Constraints", Proceedings of ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering, Los Angeles, California, Dec 1993, 115-125.

Shing Chi Cheung, Jeff Kramer, "An Integrated Method for Effective Behaviour Analysis of Distributed Systems", Proc. of 16th IEEE Int. Conf. on Software Engineering (ICSE-16), Sorrento, May 1994, 253-263.

Air Accident

1 ***Issue*** : **What was the cause of the accident?**
What were the factors which caused the accident to occur?
(Demo User)

1.1 ***Position*** (Responds To Issue **1**): **Operational**
Effect of an operational action of lack of an operational action.
(Demo User)

1.1.1 ***Argument*** (Supports Position **1.1**): **Aircrew not qualified**
Required qualifications not held
(Demo User)

1.1.2 ***Argument*** (Supports Position **1.1**): **Flight not Authorised**
Authority for the particular aspect of the flight had not been granted
(Demo User)

1.1.3 ***Argument*** (Supports Position **1.1**): **Aircraft operating outside limits**
Limits of CAA release being exceeded.
(Demo User)

1.2 ***Position*** (Responds To Issue **1**): **Engineering**
Engineering cause
(Demo User)

1.2.1 ***Argument*** (Supports Position **1.2**): **Equipment failure**
Failure of equipment
(Demo User)

1.2.2 ***Argument*** (Supports Position **1.2**): **Servicing Error**
Incorrect servicing procedures
(Demo User)

1.3 ***Position*** (Responds To Issue **1**): **Air Traffic Control**
Action or lack of action by Air Traffic Control
(Demo User)

1.3.1 ***Argument*** (Supports Position **1.3**): **Incorrect procedures**
Procedures as laid down in CAA Regulations had not been followed
(Demo User)

1.3.2 ***Argument*** (Supports Position **1.3**): **Procedures not clear**
Procedures in the CAA Regulations are ambiguous
(Demo User)

1.4 ***Position*** (Responds To Issue **1**): **Weather**
Effect of weather
(Demo User)

1.4.1 ***Argument*** (Supports Position **1.4**): **Outside operating limits for this aircraft**
Aircraft was being operated outside limits laid down in CAA Regulations
(Demo User)

1.4.2 ***Note*** (About Position **1.4**): **Copy of Meteorological Information**
Details of the weather at the time of the accident.
(Demo User)

2 ***Issue*** (Expands On Argument 1.2.1): **Was initial design correct?**
Was there an error in the original design which could have contributed to the failure?
(Demo User)

DISCUSSION

Rapporteurs: Rogério de Lemos and Cecíia M.F. Rubira

Lecture One

Professor Rechting raised the issue whether the "context independent" property of a component was an unnecessary constraint, since general purpose systems are usually less efficient than specific purpose systems. Dr Kramer agreed with the comment, and emphasized that in software terms, to be context independent is a nice property fundamentally in the initial stages of architecting the software. He continued by saying that this property provides flexibility by allowing to delay to the later stages of software development issues concerning, for instance, reliability and efficiency.

Dr Sventek commented that it was nice to have the notion of plugging in and plugging out components to a system, which is basically a context dependent issue. Dr Kramer replied that in his approach the situation is handled by adopting an hierarchical model in which there is a distinction between requirements at the same level of abstraction and requirements at different levels of abstraction.

On the topic of what level should we make the system architecture, Dr Aho made an analogy with urban planning (with which a basic infrastructure is usually associated) to ask to what degree should we leave the forces of the society to exploit such infrastructures. Dr Kramer answered that his aim was to provide a way of describing software architectures, which are just pieces of software that interact. In addition to that, one also has to consider what is visible of that architecture in order to allow users, or software engineers, to interact and manipulate with such architecture. The problem was how to place constraints in order to avoid an arbitrary intervention where everything could fall over.

Dr Aho continued by remarking that in his opinion Internet planning should follow the same principles of urban planning because of the unforeseen problems that might arise in the future. Dr Kramer replied that it was his impression that the Internet, mainly the World Wide Web, represents the opposite case, where the open planning did not established any controls over a structure. He continued by saying that an interesting question that arises is how such a system should evolve; only from experience can the right decisions be taken.

Still concerning the question made by Dr Aho, Professor Rechtin said that the concept of "unboundness", that Dr Aho was referring to, is one of the sophisticated concepts associated with architecting. He continued by saying that in general terms all systems consist of subsystems and therefore all systems are part of still larger systems, it does not matter the architecture being considered because there is always a bigger one and there are also smaller ones which leads to an hierarchy of architectures. This notion of system architecting is drastically different from the scientific approach which deals only with bounded systems where the system boundaries are very clearly delineated. Professor E Rechtin concluded by saying that he originally thought that there was only one architect, but now his impression was that an architect is a member of a family of architects.

On the possible meanings of the word architecture, Professor Randell made a comment that the meaning of the word that Dr Kramer was using, was different from the one of Professor Rechtin. He continued by saying that in his opinion he tends to think of an architecture not so much as being an architecture of a system but an architecture of a class of systems, and as such an architecture tells you the sort of things you are not allowed to do.

Lecture Two

During the talk, Dr Aho asked Dr Kramer who, in his understanding, was the client of a software architecture. Dr Kramer replied that in his terms it is not the end user who will produce the software architecture, but a software architect who produces it as a way of arguing that the system is capable of achieving what the end user wants. He continued by saying that in the kind of architecture that he was talking about the software architecture was for designing, quite far down towards the implementation end, and it was used as a way of discussing what the system is supposed to do. Rather than talking about general properties, the software architecture was a way of assigning responsibilities to a system performing different functions.

Dr Aho enquired at which stage in the software life cycle one obtains a software architecture. Dr Kramer answered that for the type of systems that Professor Rechtin was talking about an architecture is produced at the very earlier stages. Dr Kramer continued by saying that he was concerned with software architectures very late in the life cycle, those architectures which are used to construct the actual software; they are produced by the designer rather than the requirements engineer.

Professor Randell asked what was the difference between the term software architecture that Dr Kramer was using and the term programming-in-the-large. Dr Kramer answered that what software architecture does is programming-in-the-large, and what actually he was doing is configuration programming, which deals with slightly different kinds of entities, that is, components that are to be put together. This was not like writing individual sequential statements in a third generation language.

On the same topic Professor Randell asked why should software architecture be assumed to be a one level notion instead of a multiple level notion, in the sense that we can establish other levels in which we have to program in the large. Dr Kramer replied that he agrees with the notion of different levels of software architectures, however what he was showing was a suitable way to configure a particular level. He continued by saying that his approach might be suitable for some other level, but he was not claiming by any means that his approach was adequate to all different levels.

Dr Aho made a comment that Donald Knuth in his book "Literate Programming" mentions that it was very hard to write about software so that others could understand. Then he asked why in the approach presented, some kind of the description of the software was not included in the exoskeleton in order to make it much easier to understand the software so that it could automatically be transformed into its executable code. Dr Kramer replied that the key point for understanding software was its architecture because it was through the architecture that one is able to walk through the code of somebody else. He concluded by saying that it is a good thing to do to keep an explicit architecture description of the software.

Professor Randell asked whether there exists a language that documents what the system does' besides describing its structure. Dr Kramer replied that what the system is expected to do is just an attribute of the program and not part of the language, and it exists only for documentation purposes.

Dr Aho commented that in the approach presented it was clear how a system is put together, but nothing was mentioned how to capture its semantics in order to modify and maintain the system. Dr Kramer replied that presently he relies on individuals to check whether the semantics are correct, mainly because the approach is not widespread. At the moment, the documentation provided is very informal in order to check, at the design level, whether a particular combination of components satisfies the requirements imposed on the system. He also emphasised that without adequate tools nobody will use the exoskeleton.

Professor Tienari asked how scaleable was the approach. Dr Kramer replied that currently nothing exists in terms of Darwin and he could not affirm whether Darwin was the right language to write very large systems of the scale of million lines of code, perhaps it could be used for systems of a hundred thousand of lines.