Understanding Software Architecture: A Semantic and Cognitive Approach

Stuart Anderson and Corin Gurr
Division of Informatics, University of Edinburgh
James Clerk Maxwell Building
The Kings Buildings
Edinburgh EH9 3JZ

Fax: +44 131 667 7209
Email: soa@dcs.ed.ac.uk, corin@cogsci.ed.ac.uk

Abstract

We emphasise the role of Software Architecture as a medium for communicating aspects of a software system to the diverse specialist and non-specialist groups involved in the creation, implementation and deployment of the system. This approach forces us to take careful account of issues such as the domain of application, the task the architecture is supporting, and the representation used for the architecture. We present some preliminary results of this approach drawing on both semantic and cognitive analyses of Software Architecture and outline some empirical work on Software Architecture in practice based on the approach presented here.

Keywords

semantics of Software Architecture, representing Software Architecture, cognitive aspects of Software Architecture, domain specificity, reasoning about Software Architecture

1 INTRODUCTION

Much of the work on Software Architecture has taken software design as the main activity software architecture is intended to support. Here we take the view that, although support for design is important, the role of software architecture is much broader and that this broader role requires a different emphasis in its study.

Our interest in software architecture began in the study of safety-critical software. The main standard for safety-critical process-control software is IEC 61508 (IEC 1995). This states that:

> *From a safety viewpoint, the Software Architecture is where the basic safety strategy is developed for the software.*

Thus the software architecture of a software system is the description of the system that would be used by the team responsible for the safety of the system as the primary representation of the system structure. That team would include software, hardware, domain, and safety experts.

This view of architecture as a *lingua franca* for the high-level analysis is common in many areas of engineering. For example, the notions connector and component originate in Fault-tree analysis (Vesley 1981). Similarly, Hazard and Operability Analysis (HAZOP) (Kletz 1986) takes the piping and instrumentation diagram of a plant as the architectural description that represents the shared view of the plant used by the multi-disciplinary team carrying out HAZOP analysis to determine the absence of serious hazards.

This change from a view centred on the design team to one where software architecture is the representation of shared knowledge between a diverse team of experts suggests the a new working definition of software architecture that we use in this paper:

> *Software Architecture is a suitable representation of some aspects of a software system used by a potentially diverse group of technical specialists to reach reliable agreement on a shared task.*

This new definition suggests some areas of study for software architecture:

Aspects of systems: this involves looking at the domain of the system, different kinds of structure in systems and at good representations for different aspects of systems.

Representations: looking at the software architecture as a communication device brings representation into focus. We need to study errors of interpretation, the immediacy of representations, how well they support particular tasks, how well they represent particular aspects of systems.

Agreement: we are concerned that the architectural representation contains enough information that it is possible to reach agreement and demonstrate that agreement is justified.

Task: different tasks may require quite different representations of different aspects of the system. It may be that systems have many different representations. The consistency requirements between such representations are quite weak because we only need consistency of information associate with the intended task.

Our characterisation of software architecture as a communication device suggests that we need to study it in at least three ways: the logical analysis of software architecture, cognitive aspects to take account of human interpreters and social aspects to take account of the way the individuals embed in the surrounding organisations. This

paper is our first attempt at synthesising work reported in a series of papers of the *Understanding Software Architecture* project*. The paper presents three strands of research: formal (semantic) models of languages expressing architectural elements of specific domains; cognitive-based theories of representations; empirical studies of software architectures as used in industrial applications. The results reported here form the foundations for each of these three strands of research. Work currently in progress is taking a more direct approach to combining and synthesising these three strands into a coherent, holistic approach to the understanding of software architectures.

Currently, these three strands intersect in the study of the design of industrial embedded controllers. This commonly occurring class of systems spans many different domains (e.g. automotive, process control, ASIC design, mobile telephony) and is a very common component of critical systems. The approach to design is quite stable, emphasising a clear distinction between data and control flow, but suffers from very fragmented use of notations and from languages tied to specific manufacturers (as with, for example, Programmable Logic Controller (PLC) languages (IEC 1993)). Data relating to controllers is particularly interesting because: (i) a disciplined design/review process gives a detailed view of the evolution of designs; (ii) it is common to develop families of controllers for slightly different circumstances. In such circumstances the pragmatic benefits of architectural reuse are evident to designers; (iii) embedded controllers involve diverse technical specialities in their construction (thus giving an insight into our view of the use of architecture as a *lingua franca* for the design team); (iv) one aspect of our investigation is in the choice of representations for designs (see (Gurr 1997) for a discussion of this). Embedded controllers involve diverse forms of diagrammatic and textual representation to capture control flow, data flow and timing aspects of systems.

2 SEMANTIC VIEW

We have taken the PLC programming language IEC 1131-3 (IEC 1993) as a starting point for our investigation into the semantics of Software Architecture. Although IEC 1131-3 is primarily intended to support programming it does have many features in common with Software Architecture notations. We consider it to be a “naturally occurring” example of a language intended to encourage an architectural approach to system design. For us, its important features include: addressing a particular application domain, the extensive use of diagrammatic notations and a stratified approach to design at three different levels (these correspond roughly to *views* in (Perry and Wolf 1992).) These levels are: function blocks (FB), corresponding to a data flow view; sequential function charts (SFC), corresponding to a control view; and configurations, that correspond to a resource use view.

*We acknowledge the support of the UK Engineering and Physical Research Council, grant number: GR/L37953.

So far we have two main results that we believe are useful in the context of IEC 1131-3 and point to a class of results that are important for Software Architecture:

Faithfulness of notation: In (Anderson and Tourlas 1997) we explore the dataflow representation in IEC 1131-3 and characterise the requirement placed on the semantics of the programming system by the use of the diagrammatic notation. The characterisation consists of a set of equations that must be satisfied by the semantics. This captures the constraints placed on the semantics by the use of diagrams. We can see this as a minimum level of agreement expected of users of the notation.

Diagrammatic support for proof tasks: In (Anderson and Tourlas 1998) we explore the control representation of IEC 1131-3. The aim of this work is to support diagrammatically driven proofs of a simple class of properties that are designed to be relevant to the class of applications designed in the system. Here we are aiming at combining domain dependency with the limited nature of the representation to assist with a particular task which is difficult in general but is considerably simplified by restricting the class of properties and choosing an appropriate representation.

These two results take the representation as a critical element in supporting the analysis of system designs. One result looks down by capturing the requirements the representation of the system places on the semantics of the system while the other looks upwards towards the support of complex tasks based on the representation. Both of these results are interesting instances of more general cognitive work on the use of graphical representations.

3 COGNITIVE VIEW

The semantic view of Software Architecture provides some control over the safety of representations and the correctness of tasks supported by a particular representation. To assess how well a particular representation meets the needs of it's users we need finer tools that consider features of representations (e.g. complexity, clarity, ...) that are not considered in the semantic view.

From a cognitive point of view to match a representation to a task (and domain), we must consider the following three issues:

Matching: What properties do diagrammatic representations possess, so that they might be matched to properties in the task and domain at hand? Some of our work (Gurr 1998) develops a general framework based on the extent to which the representation is isomorphic to the represented situation. In our previous studies we have looked at representing simple syllogisms and have observed that close matching of representations aids understanding. In the case of Software Architecture we might hope to observe that certain structural relations are preserved in the

diagrammatic representations. We have achieved this in the representation used in our representation of a restricted class of IEC 1131-3 Sequential Function Charts. Further work is needed to see the extent to which this is helpful.

Complexity: What effect does representation have on the complexity/salience of a task? In earlier work (Stenning and Oberlander 1995) we have observed that the limited expressiveness of diagrammatic representations seems to help aid some reasoning tasks. The choice of an appropriately matching representation with expressiveness closely matched to the reasoning task seems to provide appropriate support for humans undertaking the task. This work helped motivate the design of our simple proof system for Sequential Function Charts. The result has been a system for proving safety properties of systems where the independence of concurrent components is directly expressed in the graphical notation. We believe this greatly simplifies the task of reasoning about the system and it restricts concurrency in SFC systems to a case that is easily understood by non-programmers.

Human variability: the human element: what more is there to consider, beyond an analysis of the logical properties of diagrammatic representations? In recent work (Stenning and Yule 1997, Stenning, Cox and Oberlander 1995) we have observed significant variation in the extent to which a particular representation aids reasoning tasks. These variations arise from variations in the subjects capacity to utilise the representation effectively. It seems that for some people graphical representations can be a very poor choice to support some tasks. We believe this work is important in the evaluation of graphical representations of Software architecture because this variability may mask the discovery of representations that very effective for some classes of user. We anticipate observing similar effects for Software Architecture representations.

We believe all three of these cognitive issues can be used to motivate the design of Software Architecture representations that are finely tuned to support specific tasks effectively. These combined with the semantic approach discussed in the previous section offer good methods to analyse architectural representations. In the next section we consider some preliminary empirical investigations that are motivated by our semantic and cognitive work on representations.

4 EMPIRICAL STUDIES

Having outlined our *theoretical* approaches, both in the formal modelling of properties software architectures and in the cognitively-situated models of (diagrammatic) design notations, we now turn to the third strand of our research: that of our more pragmatic and practically based empirical work.

The aim of our empirical work is to develop a sharp characterisation of the role of software architecture in practice and to gather concrete examples of its use. We report here the initial findings of a study of the development of designs for software-based, automotive engine management systems (this data has been anonymised to protect the confidentiality of the supplier). The analysis of these designs is particularly in-

teresting in the light of our work of Section 2, as again here we are considering the class of industrial embedded controllers. Indeed, the structure and notations used in these designs are quite close in concept to the PLC notations discussed in Section 2.

4.1 Case Study: Architecture in Engine Management Systems

The data we are analysing consists of sequences of design review reports, which detail the stepwise development of designs for engine management systems. Any given project (a single engine management system) consists of a number of modules, typically between 8 and 12 modules per project, with a separate design for each module. A module contains components (any number from 1 to over 160 per module) which are of one of three types: 'control flow diagram' (cf); 'data flow diagram' (dfd); and 'code' (cd). Control flow diagrams are effectively finite state automata which indicate the flow of control through the module. The code defines functions computed over variables and the dfd's specify how these variables are shared between components.

The development of designs follows a rigorous design/review cycle. A design (for some module) is produced by a member of the design team and then reviewed by two other members of the team, who record their findings in a review report. The reviewers rate each component and, give the module an overall rating.

We have collected the entire sequence of design review reports for four separate projects.In total, over the four projects there are 39 modules, consisting of 852 components. Overall, these were subjected to a total of 169 reviews. This represents a substantial and rich data corpus, presenting us with a highly detailed view of the evolution of these designs.

4.2 Hypotheses and Analysis of Data

Initial examination of the data indicates that, as with PLC languages, the developers used design descriptions in which control flow dominates. From this we hypothesised architectural structure will predominately reside in the control flow descriptions, suggesting that: (i) parts of the cf (control flow) components would exhibit significantly greater stability over the design/review process than would other components; (ii) similar control patterns recur across different projects; (iii) design notations emphasise the stable aspects of control flow.

In our initial analysis of the data we have tested the first of these hypotheses by determining the average number of changes made to any one component during a review, and sorting the results by type of component (i.e. cf, dfd and cd). Figure 1 presents these results summarised over the four projects. This table indicates confirmation of our initial hypothesis, with dfd components being changed on average 0.705 times per review as compared with the 0.493 average for cf components. This does indeed confirm that data flow components within a design are revised signific-

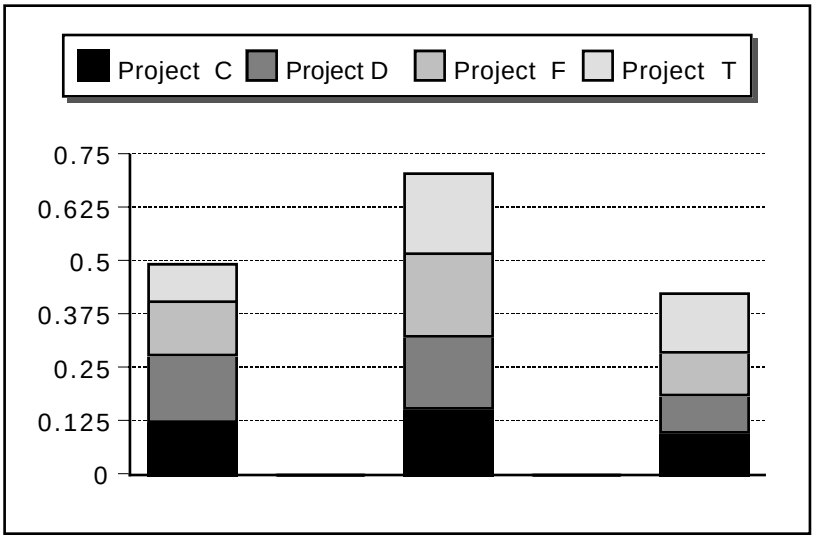

Figure 1 Average changes/review of (from left to right) *cf*, *dfd* and *cd* components.

antly more often than are control flow components. Naturally, these results hide a great deal of detail – detail which requires substantial further analysis.

5 CONCLUSIONS

The concept of Software Architecture, we argue, requires understanding in a context which is wider than simply 'design'. Our belief is that it should be analysed as a communication method. To analyse software architectures with this view requires an understanding grounded in theories of formal semantics; of the effect of representation on human reasoning; and in an empirically validated understanding of the use of software architecture in practice. In particular, empirical work is important as a check on hypotheses and on the applicability of the work undertaken in the other two research strands, and in their synthesis.

It is our view that one route to providing a stable basis for software engineering is to draw practice in software engineering closer to that of conventional engineering. Our research into high-integrity systems (MacKenzie 1996, MacKenzie 1995) indicates that safety arguments in conventional engineering are predicated upon the *continuity* of the concrete artifact being designed. Engineers reuse generic architectures and components because their properties are understood and agreed upon by all the different technical specialities involved in the design and operation of the artifact. The architecture is the stable structure which characterises the artifact during detailed design and carries its main characteristics from version to version of the

product. We believe that the work outlined in this paper offers some progress in this direction.

REFERENCES

Anderson, S. and Tourlas, K.: 1997, Diagrams and programming languages for programmable controllers, *Proceedings Formal Methods Europe '97*, Vol. 1313 of *LNCS*, Springer, pp. 1–??

Anderson, S. and Tourlas, K.: 1998, Design for proof: An approach to the design of domain specific languages, *Third FMICS Workshop*. To appear in Formal Aspects of Computer Science.

Gurr, C.: 1998, On the isomorphism, or lack of it, of representations, *in* K. Marriot and B. Meyer (eds), *Theory of Visual Languages*, Springer. In press.

Gurr, C. A.: 1997, Knowledge engineering in the communication of information for safety critical systems, *The Knowledge Engineering Review* . Cambridge University Press. *In Press*.

IEC: 1993, *IEC 1131-3: Programmable Controllers – Part3: Programming Languages*, IEC 1131-3: 1993(E) edn, International Electrotechnical Commission.

IEC: 1995, *Draft IEC 1508 – Functional safety: safety related systems*, International Electrotechnical Commission.

Kletz, T. A.: 1986, *HAZOP and HAZAN: notes on the identification and assessment of hazards*, Institution of Chemical Engineers, Rugby, UK.

MacKenzie, D.: 1995, A worm in the bud?: computers, systems and the safety-case problem, *in* T. Hughes and A. Hughes (eds), *The Spread of the Systems Approach*, Chicago University Press. To appear.

MacKenzie, D.: 1996, How do we know the properties of artifacts?: applying the sociology of knowledge to technology, *in* R. Fox (ed.), *Technological Change*, Harwood, London, pp. 247–263.

Perry, D. E. and Wolf, A. L.: 1992, Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes* **17**(4), 40–52. *http://www.bell-labs.com/user/dep/work/papers/swa-sen.ps

Stenning, K., Cox, R. and Oberlander, J.: 1995, Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences, *Language and Cognitive Processes* **10**.

Stenning, K. and Oberlander, J.: 1995, A cognitive theory of graphical and linguistic reasoning: logic and implementation, *Cognitive Science* **19**, 97–140.

Stenning, K. and Yule, P.: 1997, Image and language in human reasoning: a syllogistic illustration, *COgnitive Psychology* **34**(2), 109–159.

Vesley, W.: 1981, Fault tree handbook, *Technical Report NUREG 0492*, US Nuclear Regulatory Commission.