Software Architectures in Industrial Systems: An Empirical Study

C A Gurr

Human Communication Research Centre
University of Edinburgh
Edinburgh EH8 9LW, UK
email: C.Gurr@ed.ac.uk
Phone: +44 131 650 4448
Fax: +44 131 650 4587

Abstract

“Understanding Software Architectures” is an ongoing project at the University of Edinburgh, investigating the extent to which software architecture can be provided with a secure foundation. This investigation involves empirical, theoretical and pragmatic study of the use of architectures in real industrial systems. We present here an overview of the aims of this project, and report preliminary results in the empirical analysis of the characteristics of architectures used in the development of automotive programmable controllers.

Further information regarding this project may be found at: http://www.hcrc.ed.ac.uk/Site/UNDERSTA.html

1 Introduction: Understanding Software Architectures

The term “Software Architecture” is in common use in the computing industry and is claimed to provide a basis for properties such as safety and security. However, as yet it has no clear characterisation in the literature which could provide a sound basis for, for example, the safety strategy of a safety-critical system, or for the security strategy of a security-critical system. For example, the recently published generic safety critical standard IEC 1508 [4] states that “From a safety viewpoint, the Software Architecture is where the basic safety strategy is developed for the software”, yet this same standard contains no definition of software architecture.

“Understanding Software Architectures” is a project at the University of Edinburgh which is currently investigating the extent to which software architecture can be provided with a secure foundation. The project hypothesises that software architectures are useful because at a sensible level they impose modularity upon systems under development, making explicit the distinction between “structure” and “detail”. This modularity clarifies the interfaces between the various technical specialists involved in the design and provides the means to “factor” the arguments developers use to motivate and verify designs. The project is investigating the earlier quotation from IEC 1508 by empirical, theoretical and pragmatic study of the use of architectures in real industrial systems.

One route to providing a stable basis for software engineering is to draw its practice closer to that of conventional engineering. Research into high-integrity systems [5] 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 between versions of the product. Software systems typically lack, in any clearly expressed form, this continuity of design and this structure for carrying knowledge forward between projects. Software architecture may be a surrogate for this, and we believe that this is precisely what is meant by the earlier quotation from IEC 1508. Our hypothesis is that software architectures allow the design team to arrive at agreement by providing a structure within which inconsistencies concerning components and connectors, and their inherent properties, can be resolved in an application specific manner. A formal foundation for software architecture will assist in providing an explanation of the link between best practice and the reliability of systems, clarifying the contribution of software architecture to system reliability. It is strongly to be desired that any such foundation should clearly account for current engineering practice. Our work aims to be closely linked to and motivated by empirical observation of the use of software architecture in diverse real-world environments.

Here we consider 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 [3]). Data relating to controllers is particularly interesting because: (i) a disciplined design/development 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) one aspect of our investigation is the use of...
architecture as a *lingua franca* for the design team. Embedded controllers involve diverse technical specialities in their construction; (iv) one aspect of our investigation is in the choice of representations for designs (see [2] 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 Case Study: Architecture in Engine Management Systems

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 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. They may also provide comments which are similarly recorded in the review report. If a module fails review then it is subsequently modified by the original author and submitted for further review (not necessarily by the original reviewers). Typically any given design will require between 3 and 6 reviews to pass, although some designs pass on first review, and at least one underwent 27 reviews before passing.

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.

3 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 significantly more often than are control flow components.

Naturally, these results hide a great deal of detail – detail which requires substantial further analysis. One initial point we may note is that, while for each of the four projects the overall ratio of changes to cf components versus changes to dfd components did not deviate significantly from that suggested by Figure 1 (a ratio of approximately 0.493/0.705 = 0.7), there was substantial variation in this ratio within individual projects. For example, Figure 2 presents this ratio (average changes to cf components over average changes to dfd components) for each module within two of the projects (projects ‘D’ and ‘F’). For project F the overall ratio was roughly in line with that for the total figures of Figure 1, while for project D the ratios were closer to one-to-one. However, what is of interest in the tables of Figure 2 is not the overall ratios, but the degree of variability in this ratio between individual modules. For project D the ratios are relatively consistent, whereas for project F the ratios vary substantially between differing modules. The significance of this variation between modules in project F, and the characteristics of variability between projects, is currently under investigation.

Further investigation is also required to both confirm and refine our initial hypothesis (that it is a control flow view of the designs which determines the character of their architecture). For example, building upon this initial hypothesis, we would further suggest that where a module’s design has been subject to a number of reviews, then the cf components would not only be revised less often than the dfd components, but also that the revisions to the cf components would be less substantive – consisting primarily of ‘fine tuning’ of these components at the start and finish of the review process. By comparison we would suspect that the dfd components would be subject to more substantive revision, and that these revisions would take place throughout the review process (although perhaps less often at the start). These hypotheses are similarly under current investigation, and we expect the results to help illuminate the developers understanding of the nature of the architecture of these systems.

4 Conclusions and Future Research

Embedded controllers are an economically and technically significant class of systems. Their relative simplicity, proximity to conventional engineering components and relatively carefully controlled development process suit them to empirical study. Our belief is that in the class of controllers we consider, certain aspects of the control flow descriptions are “architectural” and hence are subject to less change during development than other system components. On a preliminary analysis, this seems to be supported by our data. We believe we can refine our analysis further by classifying some aspects of the control flow as “architectural” and by considering structural comparisons of control flow across different projects.

We believe our initial hypothesis can be considerably refined, as we have a rich data set to work from and will have access to similar data from another domain this year. Future
Figure 1: Average changes/review of (from left to right) $cf$, $dfd$ and $cd$ components.

Figure 2: Ratio of average changes of $cf$ versus $dfd$ components, by individual module.
work will include: (i) augmenting our data with interview
data gathered from designers; (ii) looking more closely at the
tasks involving design notations. Some recent work [1] on
the semantics of PLC languages indicates that the utility of
particular representations are closely linked to the complex-
ity of tasks. It may be that notations, which aid reaching
agreement for some tasks, fail when they are transferred to
a new context; (iii) experimenting with representations of
computational systems to see how they help or hinder the
completion of varying tasks.

Underlying our project is the idea that Software Archi-
tecture is the focus for achieving agreement between diverse
members of a project team. This involves studying the pro-
cess of reaching agreement at a semantic, cognitive and so-
cial level. The Software Architecture should carry enough
information to promote disagreement where that is neces-
sary for the success of the project but should remain silent on
detail that is inessential ensuring the success of the project.
Achieving an understanding of this role requires empirical
study of real projects, analysis of the social context of use,
experiments in the cognitive process surrounding software
design and semantic analysis of the structure of Software
architecture.

References

[1] S. Anderson and K. Tourlas. Diagrams and program-
ming languages for programmable controllers. In Formal

[2] C Gurr. Knowledge engineering in the communication of
information for safety critical systems. The Knowledge

[3] International Electrotechnical Commission. IEC 1131-3:
Programmable Controllers – Part3: Programming Lan-

1508 – Functional safety: safety related systems, June
1995.