An Empirical Investigation of Natural Architectural Constraints

T.R. Addis
University of Portsmouth
Tom.Addis@port.ac.uk

B. Visscher
University of Portsmouth
Bart-Floris.Visscher@port.ac.uk

Abstract

An extended functional dependency analysis was developed in order to investigate the natural structures of programs (if it exists). This initial study was conducted with the eventual purpose of exploring the effects of programming language and development methods on program architecture. The analysis was designed to produce a uniform representation across different programming languages. Further, this representation was chosen so that it merges the classical distinction between data and procedure into a single functional form by using an extension of the normal functional dependency, as used in functional analysis, to include functional behaviour. Using this extended functional dependency analysis, several programs were transformed from two language sources and some natural bounds were discovered that seem to be independent of both language and method. Based on these natural bounds, two additional analyses were developed. Both these additional analyses, the function role analysis and keystone analysis, exposed further constraints that could also not be explained by language, method, architectural or problem domain constraints alone.

1 Introduction

The AMUSE project [1], which stands for Architectural Modelling to Understand System Evolution, deals with finding an isomorphic relation between a problem domain and the architecture of integrated software within that problem domain. One of the main triggers that started this project was the observation that some changes can be done easily while others require complete restructuring of the integrated software. Since program evolution means that the software will evolve, we have to deal with how changes in the problem domain impact on the artefact in terms of its structure and behaviour. This study of effect, in turn, will deal with notions like flexibility, robustness, reliability, efficiency, manageability and complexity.

We will focus on analysing the program structure and which constraints need to be satisfied in order for the artefact to remain manageable from a complexity point of view. We explored the design process and searched for architectural structures through a set of exemplar programs. We took five well established small/medium-scale industrial programs written in Fortran and C and subjected them to two types of analyses, the function role analysis and the keystone analysis. Both analyses use a functional dependency model to represent the program while remaining independent of its implementation language. This conversion of a program to its functional form was done automatically through specially written translation programs. We will not be describing this translation process in this paper.

2 The Functional Dependency Model

We standardised all programs so that language form and type are not issues. To achieve this we considered all programs in their functional form. This form is extended to explicitly give the program structure as a set of functional dependencies between functions as well as data.

2.1 Internal Architecture Of A Function

The theory of denotational semantics shows that all procedural languages and OO languages can be converted into a functional format, albeit with many practical difficulties. This makes the functional representation, in principle, a unifying description of all styles of programming languages. Since a function is a mapping, it means that procedures, arrays[1],

[1] An array is modelled as a mapping from the array index to the value of the array element.

constants[2] can all be modelled as functions. By ignoring the internal architecture, the description of how the mapping is implemented, and merely seeing this as a mapping, we can concentrate on the external program architecture without loss of behaviour.

2.2 Functional Dependencies; The External Architecture

A functional dependency is the relation between how the mapping of one function influences the mapping of another. When function A relies upon function B for its own functionality, A is then said to be functionally dependent upon B. An example of a common functional dependency is where a function A 'calls' a function B as part of its processing. This will be depicted in Figure 3 and Figure 4 as:

A ® B

There can be many functions that are dependent upon B and A may rely upon many different functions. However, the functionality (the operational behaviour) of A will imply the functionality of B but not the other way around; the existence of A depends upon the existence of B. We will call function A the *dependent* and function B the *enabler* because function B enables function A to have the desired range of behaviour. The terms 'enabler' and 'dependent' replaces the parent and child relationship because there is some potential confusion of meaning[3].

2.3 Relation Between Functional Dependencies , Data- and Control-Flow

There is a view that a distinction should be made between data- and control-flow in an analysis of the program. This would imply that in a correct and complete representation of the program structure, both have to be present and distinguishable. On the other hand, there is a view that control-flow is merely a synchronization message between the different processing nodes and therefore control-flow forms a subset within the data-flow paradigm. Taking this view, the focus of a program analysis must be with data-flow and not the control-flow.

Since the control-flow guides the data-flow in imperative languages, the control-flow can be used to represent parts of the data-flow but can in no way ever show the complete data-flow or distinguish between different data-flows over a single control-flow. However, if data flows through memory are inserted into the control-flows, we have the complete data-flow structure in which each link represents one or more data-flows. To make a distinction with data-flows and the links, data-flows are not used explicitly but are described as functional dependencies. This allows many data-flows to be combined into a single functional dependency, which is what happens when one subroutine calls another with many parameters resulting in a single functional dependency where each parameter is represented by a data-flow.

Since a functional dependency is a more abstract, and therefore more simple way of looking at the program structure than either control- or data-flow, it is not possible to extract the exact control- or data-flow from functional dependencies but we know that:

- A functional dependency contains one or more data-flows
- After removing all functional dependencies to and from memory, all remaining dependencies contain one or more control-flows.

2.4 An Indicator For Functional Dependencies

It is not possible to extract the exact dependency model of a program, for this requires an analysis of the code that is equal in complexity to the halting problem[4]. However, there is an effective indicator, which is the apparent communication between two functions given in the program. Although it is possible that certain pieces of code never actually communicate due to a conditional branch for example, it is assumed that communication is intended when

---

[2] A constant is modelled as a function without a variable.

[3] This is because during the activation of a function there is the converse interpretation that the existence of A depends upon the existence of B. This interpretation would imply the parent is the child and the child is the parent. The terms 'enabler' and 'dependent' replaces the parent and child relationship because there is some potential confusion of meaning. This is because during the activation of a function there is the converse interpretation that the existence of A depends upon the existence of B. This interpretation would imply the parent is the child and the child is the parent

[4] Having a general mechanism that would allow you to determine if a call is being made between two functions would allow you to solve the halting problem. This can be done by adding a function-call to the problem (of which you are trying to determine if it halts or not) and determine if the last function-call is made.

global variables or subroutine or function calls are found in the code.

2.5 Extracting Functional Dependencies from Source-Code

We expect to find problems with complexity in large-scale industrial programs that have been developed by more that one designer over a long period of time. Table 1 shows the test set program that we used and that are considered complex.

Table 1. Overview of projects

Project	Date	Problem Domain	Func-tions	P(Fun. Dep)
COJ	1987	Crude Oil Jetty (Shore installation)	193	0.041
FPS	1993	Floating Platform Smallwels	274	0.035
SUEZ	1997	SUEZ tankers (modern double hull)	363	0.034
VLC	1986	Very Large Crude Oil	322	0.031
CLARITY	2002	Schematic Functional Language PDE	1589	0.003

Two frequency distributions of the connectivity arities per function can be taken. The ranked order frequency distributions are shown in Figure 1. Similar pairs of distributions are observable for all the other example programs and this graph shows the concatenation of all the data for the test set. One of these distributions shows the rank order frequency of dependents while the other distribution shows the rank order frequency of enablers. The graph uses a log-log scale. The two important log-log distributions are Zipf's law, which according to Scarrott's conjecture [2,3,4 pp. 249] is associated with a recursive structure and the Whitworth distribution, which is associated with an arbitrary division of a fixed set of objects into subsets [3]. A more familiar example of these two distributions are words and letters respectively in the English language in which words follow Zipf's law while the letters behave as a Whitworth distribution [5].

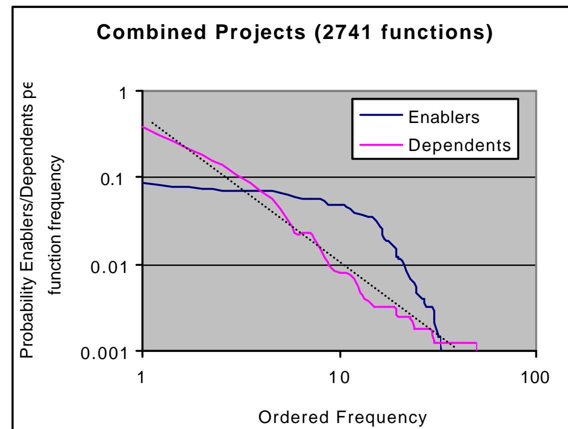

Figure 1. Rank order frequency of enablers/dependents

2.6 Selection Mechanism Of Enablers And Dependents

It can be seen in Figure 1 that the dependent distribution is a relatively straight line with respect to the enabler distribution on a log-log scale. Since a straight line on a log-log scale is a Zipf distribution there is a clear indication from Scarrott's conjecture that the creation of dependents is a structured process that is likely to be recursive. On the other hand, the enablers of a function follow a curve that appears to be based upon a random division of fixed elements (Whitworth distribution) – namely the fixed available functions at any level. This clearly shows that the mechanism, with which enablers or dependants are chosen, differs in terms of selection from either an infinite or finite set.

We can consider software design as the task of the designer to create new functions that extend the programming language. He can only do this by designing functions using the functions already provided. This suggests that the function when constructed out of other functions behave like a sentence in that the possible behaviour of a function is virtually unlimited like the number of sentences but constructed using a limited set (the words with which to construct a sentence, functions that can be used). This suggests that when the functions are constructed there is an open-ended set of possible behaviours it can choose from. On the other hand, when functions are deployed to construct a concept, they are chosen in the same way as characters are chosen to construct words; they are drawn from a fixed set of characters, the alphabet. In the case of functions, they are drawn from the current pool of constructed/built-in/library

functions, which is fixed at the time when any particular function is implemented.

3 Complexity Reduction

We assume that designers are managing the complexity of creating large software systems. We further assume they employ some strategy that reduces the complexity for the designer. As a result of this, we expect to find evidence in the program structure of the application of these strategies. Such evidence would also imply architectural constraints. These constraints would in turn determine an upper limit to the complexity of a manageable system. We would expect there to be a measure that could predict when systems become unmanageable and when reorganisation is required.

One of the conventional strategies proposed is to build modules with low coupling between them but high cohesion within [e.g. 6]. This will certainly reduce the complexity if we can presume that a module is viewed as a single concept where the internal workings of the module can remain hidden without loss of overall understanding. This notion of hiding as much information (connectivity and functions) as possible can be employed by the designer in creating and structuring code at every stage of design. Focussing on function dependencies, one of the units in the information, we note from the dependent/enabler graph, Figure 1, that about 50% of the functions only have a single dependent.

3.1 Function Role Analysis

Figure 1 shows that there is a very high chance of a function being called only once. This can be described by the role a function has as a results of a process of complexity management. We define a function as having one of the following two roles:

- Decomposing: The function has one dependent. The enabler function must therefore be a unit of decomposition for the dependent function into something more specific. Since decomposing is not arbitrary, we refer to this as the process of modularisation.
- Mediating: The function has more that one dependent. Because the function is used by several different dependents, it must have a common behaviour that is shared and useful for its dependents. We refer to the creation of these mediating functions as the process of abstraction (i.e. concept formation).

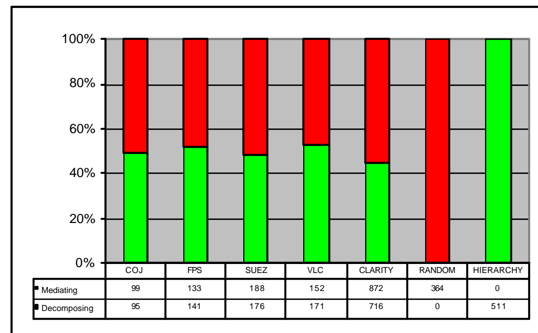

Figure 2. Results of Function Role Analysis

For comparison, Figure 2 shows the function role analysis of a random graph with same parameters as SUEZ project in terms of number of functions and probability of two functions having a dependency using a uniform distribution. In the generated graph, no decomposing functions were found. On the other hand, the analysis of a hierarchy shows the other extreme where all functions are decomposing (which by definition they are). This shows that making a model of a program as a uniformly distributed heterarchy or hierarchy will be inconsistent with these results. The actual structure of a program seems to be best represented by a mixed structure somewhere between the hierarchy and heterarchy.

Table 2. Statistical significance of function role analysis

	P(Decomposing)	*P(Mediating)*
Set size	*5*	*5*
Mean	*.4934*	*.5080*
Std. Dev.	*.0306*	*.0303*
Std. Error	*.0137*	*.0136*
T-Test 95%	*.4555*	*.4703*
Confidence	*.5314*	*.5457*

What can be observed from Figure 2 and Table 2 is that all the projects have approximately 50% decomposing and 50% mediating functions (+- 5% with 95% confidence interval). To find out if this is a typical value irrespective of the language or design method, more projects need to be analysed.

3.2 Program Sub-Structures

As suggested by the interpretation of Figure 1 we can expect design to be a recursive process. Thus structures that are created will also have a similar (recursive) structure. Because the designer is building a program, we would expect the sub-components of

this program to have a similar structure to that of a program. We would also expect similar statistics for both the program and any of its sub-programs. We thus define a program (and a sub-program) as a set of functions with a single entry point using a set of predefined functions. These predefined functions for the 'main' program, the program called from the operating system, are drawn from: the library functions, operators and other built-in functions of the language. For any of the sub-programs, they will also include user-defined functions of the program.

We will define a *keystone analysis*, which will assume that a program is a collection of sub-programs with functional dependencies between them. These sub-programs will all have a single *keystone-function* entry-point that is used to interface with the program and provide its entire functionality of the program (e.g. 'main' in C, un-headed function in Fortran-77). The size of each of the sub-programs may vary from only a single function, the entry point function, to sub-programs that contain a large number of functions, which could be entry points to other sub-programs. We thus give a definition of keystone function groups. We define:

- A *keystone function* is the entry point to a program. The keystone function is the interface between the outside world and the internal working of the program and is the only function from the program that has dependents outside the program.
- A *keystone group* is the set of functions that form a program to which the keystone function forms the interface.

A keystone group is identified through its characteristic in that all functions within the group are enablers of the keystone function or enablers of functions in the keystone group. At the same time, these enablers may only have dependents within that keystone group. The keystone group forms a program or sub-program whose entire functionality is encapsulated by the keystone function. By definition, it is impossible for any function outside a keystone group to be functionally dependent on a function inside the keystone group. However, it is allowed that a function inside a group to be functionally dependent on a function outside the group as illustrated in Figure 3.

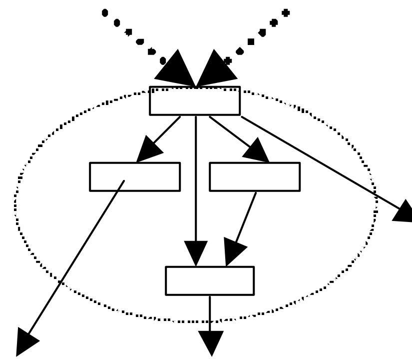

Figure 3. Keystone group with the keystone function at the top

3.3 The Keystone Analysis

All programs have been standardised into a functional form. The keystone analyses searches for the keystone groups in the functional dependency model. We begin with a function that calculates the keystone group for a single function. To analyse a complete program, this function is applied to every function in the program. This will result in a list of functions for each function; a list that is the keystone group for that function. Figure 4 shows an example of the keystone analyses applied recursively on the left structure and displayed on the right. The result is a nested group of functions.

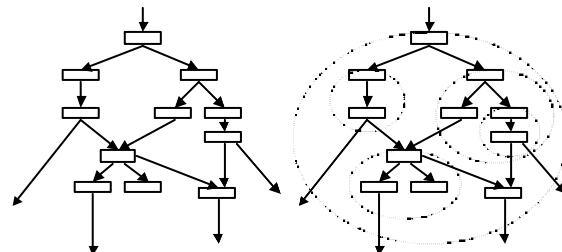

Figure 4. Example of a functional dependency model and keystone analyses

The question must arise as to whether the keystone analyses, if applied to a randomly generated graph, as done with the function role analysis, would still find structures of this kind. If the keystone analysis yields many groupings in randomly generated graphs then the structuring effect of the keystone groupings would be insignificant. An experiment was carried out on randomly generated graphs that uses the same number of functions and probability that one function is functionally dependent on another as was found in the SUEZ project (see Table 1). Six random graphs were generated and the number of non-empty

keystone groups counted. In a total of 6 trials using randomly generated graphs with the same parameters as the SUEZ project, only a single non-empty keystone group was found. This indicates that when keystone groupings are found, they are not random occurrences but an intended program structure. The result of applying the keystone analyses on the programs and counting the number of non-empty keystone groups is shown in Figure 5.

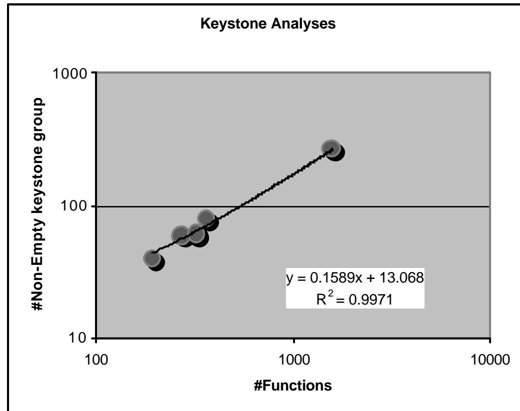

Figure 5. Graph of number of functions in relation to number of non-empty keystone groups

Figure 5 shows that there is a strong linear relationship between the number of functions in a program and the number of non-empty keystone groups. This can be approximately represented by:

$$NumberofNonEmptyKeystoneGroups \approx \frac{NumberOfFunctions + 78}{6}$$

Equation 1. Functions in relation to non-empty keystone groups

Given that the requirements by designers to cope with complexity through the principles of abstraction and decomposition are similar then this suggests that the capacities of Fortran and C to naturally provide simplicity of representation are also similar since their representation of abstraction and decomposition uses the same format (i.e. subroutines, procedures, functions).

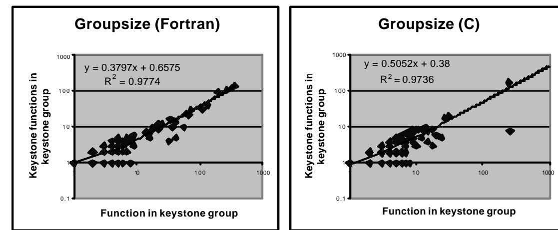

Figure 6. Function v.s. Keystone functions for every keystone group in the combined Fortran projects and C project.

To determine the effectiveness of abstraction, we measure the number of functions that need to be considered with and without abstraction. For these measurements, the data from the Fortran projects was combined into a single data set, which should not make any difference to the results, because every keystone group is considered as a program, but does give more samples. It also allows us to compare C and Fortran distributions.

Figure 6 shows the effect on the number of functions that need to be considered for a program without keystone groups (x–axis) and for the same program with keystone groups (y-axis). It is clear that the projects have a substantial number of non-empty keystone groups resulting in a reduction of concepts (functions) that need to be considered to fully comprehend the (sub-) program. This relationship for both C and Fortran can be approximately represented by the following equation:

$$NumberofKeystoneFunctions \approx \frac{NumberOfFunctions + 1}{2}$$

Equation 2. Relation between keystone functions and functions

3.4 Inferred Design Mechanism:

The main results of the analyses can best be explained by at least two separate mechanisms at work to reduce the complexity of the design. The result of the function role analysis show that for programs in the test-set, with a very different range of functions, about half of the functions are decomposing. We suggest that this is a result of the act of *decomposition*, which comes into force when functions become unwieldy. Under these conditions, the functions are decomposed into simpler sub-functions. Because of the consistency of the result, irrespective of the number of functions in the program, we can expect a similar ratio in keystone groups.

On the other hand, from the keystone analysis we can observe a second mechanism that deals with the hiding of information. We suggest that hiding information is achieved by the act of *abstraction*, which merges functions into a single concept that is represented by the keystone function. This allows the designer to merely take note of the keystone function while having the whole functionality of the entire keystone group available. This reduces the number of functions and functional dependencies that need to be considered.

Although literature might frequently refer back to these mechanisms, their existence has always been assumed. The functional dependency analysis combined with the function role and keystone analyses demonstrate empirically their use in program design.

4 Conclusion

The decomposing / mediating ratio which was a result of the function role analysis, clearly distinguishes the program structure for hierarchies and uniformly distributed heterarchies. The keystone group analysis also showed that programs are not structured in a uniformly random way. Because of the regularity of the decomposing / mediating ratio and the regularity in which keystone groups occur in relation the number of functions strongly suggest that the formation of keystone groups through the strategy of abstraction and the strategy to decompose functions, are actively pursued strategies for decreasing the 'complexity' of a program.

Understanding the principles that govern the construction of software artefacts, gives a basis on which we can investigate the impact of change on the structure. Because of the nature of functional dependencies and the indicators that are used for them, we can guarantee where change will not affect the program behaviour. This limits the potential impact of change down from the whole program to a more manageable size. At the same time we observed very strong architectural constraints in the usage of the different programming languages and possibly design mechanisms. However, due to the recursive nature of the process, there is as of yet no evidence to suggest there is an upper limit to the size of programs that can successfully be managed. Because the architecture does have constraints, there will be program structures that cannot be successfully managed. Where exactly this boundary is between manageable and unmanageable projects is yet to be established.

5 Acknowledgement

We would like to thank the EPSRC (Grant GR/R11919/01, GR/R12152/01) for their support of the AMUSE project. We have also had considerable help from Ship Analytics who provided the Fortran programs. Thanks must also be given to Clarity Support Ltd for providing the C code. This work would not have been possible without the support of other AMUSE team members, N. Clark, G. Gallal-Edeen (Principle Investigator at London Metropolitan University now HoD Faculty of Computers & Information, Cairo University), A. Gegov (Co-Investigator at the University of Portsmouth), and J.J. Townsend-Addis who developed the conversion software and did much of the initial functional analysis.

6 References

[1] The AMUSE project: http://www.tech.port.ac.uk/staffweb/addist/amuse.html

[2]. Scarrott, G.G., 1981:'*Some consequences of recursion in human affairs*', IEE proceedings 129A(1), January, pp. 66-75

[3]. Bennett, J.M., 1975:'*Storage design for information retrieval: Scarrott's conjecture and Zipf's law.*', International Computing Symposium pp. 233-237, North-Holland, Amsterdam / New York.

[4]. Addis T.R., 1985: *'Designing Knowledge_Based Systems'.*, Kogan Page. published October. Hardback ISBN 0 85038 859 7. Soft back ISBN 1 85091 251 3.

[5]. Whitworth, W.A., 1901:'*Choice and Chance*' 5th edition, pp 567-580 (reprinted by Stechert 1942).

[6]. *Concept Analysis for Module Restructuring pp351-363 April 2001 Vol. 27 No. 4,* IEEE Transactions on software Engineering