ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

Formally Assessing Software Modifiability

C.R. Roast and J.I. Siddiqi

Published in Collaboration with the
British Computer Society

ISBN 3-540-76105-5

Formally Assessing Software Modifiability

Chris Roast and Jawed Siddiqi
Computing Research Centre, Sheffield Hallam University
Sheffield, S11 8HD, UK.

Abstract

An analytic framework termed cognitive dimensions is developed to provide formal definitions of dimensions for assessing the suitability of interactive systems for particular tasks. Cognitive dimensions is a psychological framework that provides broadbrush characterisations of interactive behaviours that are of particular relevance to ease of use. The framework also provides an effective terminology to support a wide range of assessments including interface evaluation, and assessing the resistance of languages to program modification. We propose that software design can benefit from interpreting cognitive dimensions as tools for assessing software characteristics such as usability and modifiability. Our interpretation of these dimensions has the benefits of being formal and at the same time yielding practical measures and guidelines for assessment. In particular such a formalisation emphasises the degree to which cognitive dimensions can serve as constructive expressions of non-functional requirements in general. This work builds upon a growing body of work concerned with formally characterising interactive properties that are significant to successful use. In particular it examines the dimensions associated with the notion of viscosity — resistance to local change and demonstrates their relevance in the context of program modification.

Keywords: Cognitive Dimensions, Formal Modelling, Software Modification.

1 Introduction

In the area of software requirements engineering the accommodation of non-functional requirements into system development still continues to receive little attention. Initially, two particular non-functional requirements or system attributes are of interest to us, namely, usability and modifiability. Usability has been the focus of a variety of research activities both in formal methods and human computer interaction. The overall goals of these investigations have focused upon bridging the gap between usability and system specification [1, 2, 8, 14]. We have extended this work to investigates possible correspondences between a broadbrush cognitive perspective on interaction and generic interactive system properties. The current work employs techniques drawn from our existing research into formal modelling of usability requirements [16, 10] to identify the extent to which the informal notion of cognitive dimensions can serve as a mechanism for system specification of non-functional requirements. An analytical framework incorporating formal definitions of several cognitive dimensions including secondary notation, hidden dependencies and viscosity has been developed and applied to usability requirements [17].

Given the overwhelming evidence that the majority of systems built undergo extensive changes motivates us also to turn our consideration to modifiability. Of course any treatment of this attribute should at the outset realise that there is a multiplicity of factors that can ease or hinder it. For example, it is widely accepted that the choice of a particular programming language can significantly influence the success and ease with which particular software engineering tasks, such as construction and modification, can be achieved. However, the assessment and comparison of programming languages in terms of the medium by which systems are defined and modified has been given scant attention. Similarly, design quality, or the degree of modularity that a system possesses, is considered to be a highly significant factor in facilitating system modification. Though software metrics have been proposed ranging from simple line counts to sophisticated measures, such as cohesion and coupling, they appear to have paid little regard to the psychological reality of problem solving strategies employed.

Our starting point, therefore, is to take the psychologically rich informal notion of cognitive dimensions and to provide a system oriented semantic under-pinning. This is then applied in an illustrative case study involving two program

modification tasks. One involving different levels of design quality and the other involving two different paradigms. Our example indicates that the notion of modifiability considered is not necessarily clear cut, and in general, its effective assessment can benefit from formal analysis.

2 Formal Framework for Interaction

2.1 Cognitive Dimensions

Cognitive dimensions [4] were originally proposed as psychologically motivated metrics or as general characterisations of interaction that focus upon factors central to successful use. As such, these dimensions provide effective (and efficient) support for interface evaluation within a variety of contexts [7]. In contrast to other approaches to interface evaluation, dimensions provide a rich framework for examining the benefits and limitations of particular systems, and can help direct design improvements. However the existing treatments of cognitive dimensions have largely been informal and anecdotal. Moreover, the development of a definitive list of dimensions and their accurate definition is the subject of on-going research [6]. As a consequence, at present the effective use of cognitive dimensions is often speculative and largely evaluative.

The present study focuses on the cognitive dimension known as "viscosity" because as the metaphor suggests it concerns the ease with which change can be achieved while using a particular interactive system or medium. Intuitively, high viscosity means that a high degree of effort is required in order to achieve change because the medium is highly resistant to change, and by contrast low viscosity means that some changes can be achieved with low resistance from the system, and hence little effort. A simple example of high viscosity within word processing would be if a document has a numbered reference list, and a new reference is introduced early in the list. In the absence of a specific tool to help, the user would have to engage in a complex and lengthy activity of updating all references. From this example, it can seen that viscosity is a product of both the notation being manipulated (in this case a word processor document) and the tools which enables the manipulation (the word processor). Green distinguishes two types of viscosity: repetitive and knock-on.

Repetitive viscosity refers to the 'resistance' to change encountered in circumstances where a modification involves the user in effort intensive, and often repetitive, inputs. Hence, a conceptually simple modification becomes a complex activity. Examples of repetitive viscosity can be found within a variety of notations. For instance in a programming language that did not support the declaration of constants one might have written the specific value of some constant throughout the program, should this value subsequently change one would have to repeatedly identify and change each occurrence individually.

Knock-on viscosity refers to the situations in which a user, having made a change, finds that they have to make a number of additional 'corrections' in order to preserve the desired state. The first change effectively violates some intended invariant and therefore the user has to engage in a task of 'putting things right'. Knock-on viscosity concerns the manner in which a system environment can often limit or restrict how a goal is reached. A simple illustration is if a procedure definition is found to require an additional parameter, then introducing the new parameter is easily managed, however the addition has a knock-on effect on any applied occurrences of the procedure. Hence, what can be viewed as a simple modification could result in a chain of alterations.

Repetitive and knock-on viscosity are cognitive phenomena and as such behavioural characterisations of them are inherently limited. It may be the case that certain instances of both repetitive and knock-on viscosity can be attributed to what may be a lack of foresight, or experience, on the part of the user. However, in general the particular medium being manipulated will determine the extent of the foresight/experience demanded of the user.

2.2 Modelling Interaction

Adopting the familiar concept of 'dimension' suggests that it should be possible to locate individual artifacts within a cognitive space so that their (relative) locations inform us about their (relative) merits. However, appropriate dimension definitions have not been developed. In order to provide a coherent framework in which to interpret the above dimensions, we propose a basic vocabulary of concepts intrinsic to their description and characterise the vocabulary in terms of an existing interactive state based system model involving:

- the goals (and sub-goals) which users may achieve,
- the inputs which have to be performed to achieve goals,
- goal objects central to achieving user goals.

Drawing on our work [15, 16] and others (see [11]) on developing formal models of interaction, a user goal corresponds to some system property which represents the set of system states which coincide with the satisfaction of the goal. This notion is analogous to the representation of post-conditions to 'user objectives' as proposed in [8, 3]. Below we consider examples of some of these system oriented abstractions within the context of programming.

2.3 Programming Goals and Goal Objects

There are a variety of alternative views on goals and objects used in programming based upon both computational and cognitive perspectives. The computational perspective upon program design and comprehension can draw upon a variety of paradigms, including:

- the flow of data between functional units,
- the flow of control between operational units,
- the manipulation of objects and data stored within them
- the maintenance of specific relations between data elements
- the maintenance of specific relations between successive program states

We do not propose that any one of these views is in some way the 'best', nor do we consider that they exclude one another. For us a goal has a very general interpretation which is compatible with the above computational views and accords with the notion in cognition. It is simply a condition to be achieved or the end point of a series of sub-goals.

Objects in programming from a computational perspective are concerned with notational objects which form a program within some language. From a language theory standpoint, programming objects can be derived from the grammar of its language. However, empirical studies within the psychology of programming suggest that other less tangible language objects are employed in program design and comprehension. While developing programs, the cognitive perspective asserts that users employ 'plans' for solving component problems, such plans can be composed of non-contiguous segments of code which will fit within an overall program structure.

In examining the ease with which particular programming languages can be used it is necessary to consider the tools available to the programmer which may help the process of development. In the case of programming two levels of tool support can be considered, first there are mechanisms which enable the programmer to generate code in a target language, at the simplest level this may be a text editor while more advanced editors may support particular language features. Second, the availability of powerful language primitives, extensive procedure libraries, or object classes, can be interpreted as providing support.

2.4 Notation

In order to formalise the concepts introduced we employ an action logic notation, this provides a relatively concise mechanism for characterising dimensions and supports formal reasoning. From our system perspective goal properties are system states and therefore can be taken to correspond to logical values which satisfy a goal. Hence, given a goal properties p and q, we can develop logical expressions such as their conjunction $(p \wedge q)$ and disjunction $(p \vee q)$ correspond to more specific and more general goals respectively. In addition, we use of the following ACTL constructs [9]. For any proposition p:

- $[op]p$ is true iff following the input op, p is true. Hence, op leads to a state in which p is true.

- $\mathsf{EF}\ p$ is true iff there is some sequence of operations that may make p true. Alternatively read as: 'it is possible that p will be true'.

- $\mathsf{AG}\ p$ is true iff following any sequence of operations p is true. Alternatively read as 'in all future states p is true'.

Although the entire repertoire of the actions logic is not exploited here, the logic seems to provide a suitably rich framework for the current and future examination of cognitive dimensions.

2.5 Repetitive Viscosity

Repetitive viscosity is characterised by the complexity/repetitiveness of inputs necessary to achieve some goal. This attribute can be assessed by examining the 'language' of input sequences that are capable of achieving a goal (from a state in which the goal does not hold). Taking that viscosity refers to the ease with which change can take place, we consider repetitive viscosity in terms of (i) the nature of the change which takes place, and (ii) the inputs which can achieve that change. Thus, for a given change we can identify a language of user inputs which can implement that change.

The changes to the system state of interest to the users are expressed in terms of the goal properties they alter. Following Duke and Harrison's characterisation of task, a change is treated as a period of system usage which under some initial condition (pre) achieves some goal ($post$):

pre-condition	T	?	?	?
post-condition	F	. . .	F	T

as input proceeds $\longrightarrow$

Initially, the pre-condition is true and the post condition false, following an appropriate input sequence the change is achieved once the post condition is true. In general, we write $pre\ \mathsf{REP}\ post$ to denote the language of input sequences:

Definition *The input sequence $i_1, ..., i_n$ is in $pre\ \mathsf{REP}\ post$ provided:*

$$\mathsf{EF}\ pre \wedge \neg post \wedge [i_1](\neg post \wedge [i_2](\ldots \neg post \wedge [i_{n-1}](\neg post \wedge [i_n] post)\ldots))$$

In words: *From a state in which pre is true, the input sequence is able to achieve $post$.*

We propose that repetitive viscosity can be assessed in terms of the complexity of the language $pre\ \mathsf{REP}\ post$. A variety of 'measurements' of language complexity may be considered ranging from simple measures such as length of language elements to more "sophisticated" such as those that relate cognitive effort to the complexity of the grammar rules [13, 12].

Of course in general, these languages will include limitless sequences of inputs which eventually achieve the post condition. An immediate significant benefit of our formalisation is that we can propose ease of use with respect to repetitive viscosity to be the shortest sequences included in the language — the minimal language: The minimal language for precondition pre and post condition $post$ will be written: $minimal(pre\ \mathsf{REP}\ post)$, where: $l \in minimal(L) \Leftrightarrow l \in L \wedge \forall m \in L : |m| \leq |l|$.

Our proposal that the minimal language for a defined modification is indicative of repetitive viscosity, implies that particular attributes of a minimal language may serve as 'measures' of viscosity. It is not the purpose of this initial study to identify such factors exhaustively, however assessments of repetitive viscosity may be based upon simple attributes, such as the size of the minimal language or the length of its elements. Here, we take the length of the elements within the minimal language to be indicative of repetitive viscosity. Hence, the shorter the elements within a minimal language, the easier would be to achieve the associated modification and the operation can be treated as less viscous.

Under this interpretation of repetitive viscosity, generalising the pre condition, will low the associated viscosity — since, the initial condition for elements of REP will include a larger set of states.

2.6 Knock-on Viscosity

Knock-on viscosity concerns the manner in which a system environment can often limit or restrict how a goal is reached. With examples of knock-on viscosity in achieving a primary goal, the system negates other properties still required by the user.

Assuming the user wishes to achieve the conjunctive goal $p \wedge q$, then knock-on viscosity can arise from the system resisting the achievement of both conjuncts. We characterise knock-on viscosity in terms of how achieving some goals can interfere with others. The manner in which a goal p can interfere with another q is characterised by a relation disrupts — p DIS q as achieving p negates q. The disruption of q by p complicates achieving their conjunction. In general viscosity of this sort is particular to a users means of achieving p, hence it is defined with respect to the input operation which achieves p.

Definition *The formal definition of disrupts relates two goals, and an input — p DIS q for the input in iff*

$$\mathsf{AG}\ (\neg p \wedge [in](p) \Rightarrow [in]\neg q)$$

To ensure that this behaviour is not necessitated by the domain model, we also require that p and q are not mutually exclusive: AG EF $(p \wedge q)$.

In words: *For all future states whenever the input in achieves p it negates q.*

The significance of the relation disrupts for knock-on viscosity can be assessed in a variety of ways. The simplest interpretation of disrupts is that it can indicate an inappropriate order for achieving particular goals. Thus, if q holds and the user wishes to ensure $p \wedge q$, then achieving p would have the knock-on effect of negating q, hence, it may be more appropriate to achieve p first, then q. However, since DIS can be symmetric (and dependent upon particular inputs) the relation cannot provide an optimal order in which to achieve to goals. For instance, if p DIS q and q DIS p, then the interference between p and q cannot be avoided, for the inputs concerned.

The significance of an instance of disrupts will largely be determined by the generality of the goals it involves. Assuming p DIS q: the more general the conjunction $p \wedge q$ the higher the possibility of knock-on viscosity being encountered.

3 Program modification case study: average to filtered average

Our formal accounts of viscosity can be illustrated by comparing it with an informal example taken from [5]. This example from the psychology of programming focuses upon an individual programming task and solution using specific programs. Such examples are used to characterise general features of a chosen language. Here we take a similar approach, our analysis of the examples serves as an illustration of the notions of modifiability captured by our formal treatment of viscosity.

Green illustrates differences in the viscosity of two programming languages by considering the complexity of introducing a change in each language. The particular modification considered is that of changing a program which computes the average of a series of integers to one that also computes the 'filtered average' of the same series. The 'filtered average' is the series average ignoring all zero elements. Here we extend and re-work Green's example relating it to the general definitions introduced above. This enables a more reliable assessment of the particular languages considered and also demonstrate the potential applicability of formalised cognitive dimensions.

The languages for which this modification is examined are a structured BASIC and the declarative language PROLOG. The change to the BASIC program (from **P1** to **P2**, figure 1) involves the insertion of three lines, where as the change to the PROLOG (from **DA1** to **DA2**, figure 2) involves many more modifications and, hence, is intuitively more viscous. We can apply our formalisation of repetitive viscosity as minimal language for these examples.

3.1 Analysis of repetitive viscosity

The repetitive viscosity of achieving the example modification may be expressed as: $minimal(G1\ \mathsf{REP}\ G1 \wedge G2)$, where:

```
1:  Count = 0
2:  Sum = 0
3:  READ Item
4:  WHILE Item <> 9999
5:     sum = Sum + Item
6:     Count = Count + 1
7:     READ Item
8:  WEND
9:  PRINT Sum/Count
```

P1

```
1:  Count = 0
2:  Sum = 0
3:  FCount = 0
4:  READ Item
5:  WHILE Item <> 9999
6:     Sum = Sum + Item
7:     Count = Count + 1
8:     IF Item > 0 THEN FCount = FCount + 1
9:     READ Item
10:  WEND
11:  PRINT Sum/Count
12:  PRINT Sum/FCount
```

P2

Figure 1: Introducing a filtered average in structured BASIC. Additions and insertions are underlined.

```
average1(L,Result) :-
   sumAndCount(L,Sum,LL),
   Result is Sum div LL.
sumAndCount([],0,0).
sumAndCount([H|T],Sum,L) :-
   sumAndCount(T,SubSum,SubL),
   Sum is SubSum + H,
   L is SubL + 1.
```

DA1

```
average2(L,Result,FResult) :-
   sumAndCountF(L,Sum,LL,FLL),
   Result is Sum div LL,
   FResult is Sum div FLL.
sumAndCountF([],0,0,0).
sumAndCountF([0|T],Sum,L,FL) :-
   sumAndCountF(T,Sum,SubL,FL),
   Sum is SubSum + H,
   L is SubL + 1.
sumAndCountF([H|T],Sum,L,FL) :-
   H > 0,
   sumAndCountF(T,SubSum,SubL,SFL),
   Sum is SubSum + H,
   L is SubL + 1,
   FL is SubFL + 1.
```

DA2

Figure 2: Introducing a filtered average in PROLOG. Additions and insertions are underlined.

$G1$ = "The program computes the average of a series"
$G2$ = "The program computes the filtered average of a series"

In order to realistically apply our formalisation to the examples (**P1** to **P2** and **DA1** to **DA2**), it is necessary to ensure that the goals used adequately characterise the qualities of the specific programs. Hence, it will be assumed that the goals $G1$ and $G2$ demand good programming practice and style throughout, such as reasonably efficient code, the use of "meaningful" identifier names, etc. For the purposes of this example such factors are treated as implicit, though a more thorough analysis of any notation may demand that such issues are addressed explicitly.

We are unable to apply our definition as it stands within the context of these examples, since it would require extensive analysis of the goals $G1$ and $G2$. However, the indicators for measuring complexity of modification depend upon the sequences of operations necessary to achieve the modifications. The necessary operations can be measured for the examples. Taking individual operations to correspond to the insertion (or replacement) of lines within the code, the following table is obtained:

P1 to **P2**	**DA1** to **DA2**
insert count initialisation (line 3) insert conditional increment (line 8) insert output line (line 12)	6 lines, additional `sumAndCountF` clause 5 in-line modifications, `sumAndCountF` 1 in-line modification, call to `sumAndCountF` 1 line to compute filtered average
Total: 3	Total: 13

Applying this measure of complexity, it can be seen that the modification within the procedural language is more complex than in the declarative language, this concurs with Green's informal analysis, the BASIC program modification is less viscous that of the PROLOG program.

We can examine the same measure of complexity for a program modification in which the program is designed to be highly modular. This example is **DB1** to **DB2** (see figure 3), the programs here avoid the state-based computation of averages and exploit code re-use.

The same measure applied to this higher quality solution totals 9 operations, this suggests that the improved solution has higher modifiability. In addition, this measure of **DB1** to **DB2** shows that the first PROLOG solution is clearly not minimal, hence the complexity of **DB1** to **DB2** is more representative of PROLOG's repetitive viscosity than is the complexity of **DA1** to **DA2**.

Above we have demonstrated the application of our definition of repetitive language as an indicator of repetitive viscosity. This application illustrates that our intuition regarding viscosity is reflected in its formal treatment, for a specific example. In general, the value of formal measure of repetitive viscosity is more extensive than that demonstrated. By developing a generic definition the potential exists for alternative notations to be precisely assessed and compared in terms of repetitive viscosity without reliance upon specific (and possibly un-representative) samples.

3.2 Disruptions in average to filtered average

Within the context of program modification, examples of knock-on viscosity can be found when a program is changed in some locality and alterations outside the locality become necessary. We illustrate this within the context of the previous examples of program modification. In Green's consideration of the examples the alternative programming languages are not assessed in terms of knock-on viscosity. However, our analysis illustrates that knock-on viscosity can be assessed for the given examples.

For this illustration, we employ particular programmer goals motivated by a conventional strategy for computing averages:

$G3$ = "a filtered count is accumulated"
$G4$ = "all counts are initialised"
$G5$ = "all counts are used"

The goal $G3$ is clearly concerned with satisfying the overall goal of computing the filtered average ($G2$), whereas $G4$ and $G5$ are more generic and concern good programming conventions, this is evident in their universal nature. Using

```
average1(L,Result) :-
   sum(L,Sum),
   length(L,LL),
   Result is Sum div LL.
sum([],0).
sum([H|T],Sum) :-
   sum(T,SubSum),
   Sum is H + SubSum.
```

DB1

```
average1(L,Result) :-
   sum(L,Sum),
   length(L,LL),
   Result is Sum div LL.
sum([],0).
sum([H|T],Sum) :-
   sum(T,SubSum),
   Sum is H + SubSum.
average2(L,Result,FResult) :-
   average1(L,Result),
   filter(L,FL),
   average1(FL,FResult).
filter([],[]).
filter([H|T],[H|FT]) :-
   H > 0,
   filter(T,FT).
filter([0|T],FT) :- filter(T,FT).
```

DB2

Figure 3: Introducing a filtered average in PROLOG exploiting re-use and modularity. Additions and insertions are underlined.

goals $G3$, $G4$ and $G5$, we examine approaches to making the modifications in our example programs, and identify instances of disruption between the goals. These instances of disrupts demonstrate scenarios within the examples where knock-on viscosity may be encountered.

We examine the disruption that can result from the programmer first ensuring that a filtered count is accumulated (that is achieving $G3$):

- For the structured BASIC, (**P1** to **P2**) the filtered count is accumulated by the operation introducing the conditional at line 8: `IF Item > 0 THEN FCount = FCount + 1`. Inserting this line introduces a counter: $\neg G3 \wedge [insertLine]G3$. However the counter is not initialised and not used, hence both goal $G4$ and $G5$ are false following the insertion $[insertLine](\neg G4 \wedge \neg \mathrm{G5})$. As a consequence, additional effort is required to make $G4$ and $G5$ true (in this case lines 3 and 12 have to be introduced).

- For the first PROLOG example (**DA1** to **DA2**) the filtered count is accumulated by extending and modifying the predicate `sumAndCountF` and the call to it: $\neg G3 \wedge [insertBaseCase]G3$. The predicate uses two counters both of which are implicitly initialised by the base case within the definition: `sumAndCountF([],0,0,0)` — for an empty series of numbers both counters are zero. Since, introducing the filtered count does not necessitate additional work initialising the count, $G4$ is not disrupted: $\neg[insertBaseCase]\neg G4$. However, the new counter is not used until the final pair of conjunctions in `average2`, $[insertBaseCase]\neg G5$. Therefore the introduction of the new counter leads to $G5$ being false, and necessitates the additional work of modifying `average2`.

- For the second PROLOG example (**DB1** to **DB2**) the program **DB2** re-uses code present in **DB1**. Because of the re-use of code we find that computing the filtered count, initialising and using the new counter is embodied in one call to the existing predicate `average1`. Hence, $G3$ is achieved by introducing a second call to `average1`: $\neg G3 \wedge [insertBaseCase]G3$, $G4$ and $G5$ are unaltered $\neg[insertBaseCase](\neg G4 \wedge \neg G5)$. In this example there is no case of knock-on viscosity between the goals considered.

In themselves these examples do not serve as instances of knock-on viscosity since they are based on individual scenarios. However, the general nature of the goals considered and language characterstics indicate likely disrupts

relations. The following table summarizes the examination of the influence of achieving $G3$ upon the goals $G4$ and $G5$, in each of the examples:

	P1 to **P2**	**DA1** to **DA2**	**DB1** to **DB2**
$G3$ DIS $G4$	YES	NO	NO
$G3$ DIS $G5$	YES	YES	NO

The specific examples do not reliably indicate the quality of the modifications in the alternative languages and solutions, however they suggest that the modification to BASIC is higher in knock-on viscosity than for the PROLOG cases, and the PROLOG modification involving re-use appears to be less viscous still. Hence, we may speculate that both the declarative language and program re-use have the potential to lessen knock-on viscosity.

It is interesting to note that for these examples knock-on viscosity do not correlate with repetitive viscosity. The modification to BASIC was low in repetitive viscosity while being comparatively high in knock-on viscosity, whereas the converse was true for the PROLOG examples. It can be seen that Green's intuitive concept of "resistance to change" involves a variety of competing factors.

In general, formalising knock-on viscosity enables more than specific examples to be analysed. By developing a valid generic definition, alternative notations are open analysis without reference to specific examples and uses, in addition the formal framework supports analysis of dependencies between dimensions such as repetitive and knock-on viscosity.

This not only benefits our understanding of particular uses of notations, but enables the analysis of the competing demands made by any notation (and interface to it) upon the user.

4 Concluding Discussion

This paper has described an initial investigation into the formal characterisation of cognitive dimensions. Two aspects: repetitive and knock-on of the dimension viscosity has been given a system based interpretation using a system model of interaction. In general terms the work has illustrated how a strongly cognitive view of interaction can be related to formal system properties which are suitable for reasoning about interaction, and for our particular examples the interaction involved in the programming task of modification.

Programs are structures which are manipulated in tasks such as prototyping, implementation, and modification, therefore, the ease with which particular manipulations can be achieved is clearly of significance in the context of program development. Similarly assessing the relative ease of manipulations in different programming languages is also of concern for programming practice. The case study has clearly demonstrated the relevance of viscosity to program modification and hence to the development activity. Thus, having examined and developed a better understanding of the relationship between program modification and the resistance of particular programming languages, we must also pay attention to how the nature of this relationship is significantly influenced by the tools used to achieve the manipulation. Similarly, the relationship between program modification and design quality or modularity needs to take cognisance of not only the notation used but also the abstractions the notation promotes or supports.

This paper has addressed a major weakness of cognitive dimensions by proposing system oriented formal definitions within a common framework that not only contributes to a clearer understanding of these dimensions but also enables their potential impact upon design to be considered. Our conjecture that if these dimensions are to be effective and efficient in supporting design, system oriented formal characterisations of them are necessary has been demonstrated. Moreover, such characterisations not only provide precise expositions of the dimensions but also enable trade-offs and comparisons between dimensions to be made. Focusing upon system based expressions of cognitive dimensions means that we have developed psychologically grounded interpretations of non-functional requirements such as usability and modifiability. Such interpretations also offer the potential to treat usability requirements in a manner akin to functional ones in that they can be incorporated into functional requirements, as illustrated in [16]. Alternatively as shown here, these interpretations inform our understanding of the relationships between design quality or medium used and modifiability.

Interpreting cognitive dimensions within a well defined framework provides a 'sounding board' for refining the concepts underlying cognitive dimensions. Based on our analysis, misconceptions and ambiguities within dimensions

can be clarified within the limits of the system perspective offered. In addition to this a long term goal of this work is to develop a well defined vocabulary of cognitive dimensional primitives to underpin reasoning about dimensions and enhance their coverage of human factors. In turn, such development would enhance the scope of human factors issues that can be analysed and applied early in interface development.

Acknowledgments

The work described has been motivated and encouraged by discussions with the originator of cognitive dimensions Thomas Green of Applied Psychology Unit, Cambridge.

References

[1] Barnard, P. J. and Harrison, M. D. Towards a framework for modelling human computer interactions. In Gornostaev, J. (ed), Proceedings International Conference on HCI, EWHCI'92, pages 189–196. Moscow:ICSTI, 1992.

[2] Dix, A. J. Formal Methods for Interactive Systems. Academic Press, 1991.

[3] Duke, D. J. and Harrison, M. D. Mapping user requirements to implementations. Software Engineering Jounal, 10(1):13–20, 1995.

[4] Green, T. R. G. Cognitive dimensions of notations. In Sutcliffe, A. and Macaulay (Eds), People and Computers V, pages 443–460. Cambridge University Press, 1989.

[5] Green, T. R. G. The cognitive dimensions of viscosity: a sticky problem for HCI. In Diaper, D., et al. (Eds), Human-Computer Interaction — INTERACT'90, pages 79–86. Elsevier Sciences (North-Holland), 1990.

[6] Green, T. R. G. and Benyon, D. The skull beneath the skin: Entity-Relationship Models of Information Artifacts. 1995. In preparation.

[7] Green, T. and Petre, M. Usability analysis of visual porgramming environments: a 'cognitive dimensions' framework. The Journal of Visual Languages and Computing, 7(2):131–174, 1996. In press.

[8] Harrison, M. D., Blandford, A. E., and Barnard, P. J. The software engineering of user freedom. Technical Report Amodeus 2 Document, University of York, November 1993.

[9] Nicola, R. D., Fantechi, A., Gnesi, S., and Ristori, G. An action based framework for verifying logical and behavioural properties of concurrent systems. In Proceedings of 3rd Workshop on Computer Aided Verification, 1991.

[10] Parker, H., Roast, C., and Siddiqi, J. Towards a framework for investigating temporal properties in interaction. Technical report, Jan' 1996. Paper presented at User-Centred Requirements Engineering Workshop, The University of York.

[11] Paternò, F. (ed). Proceedings, EUROGRAPHICS Workshop on the Design, Specification, Verification of Interactive Systems, Bocca di Magra, Italy, Eurographics Seminar Series. Springer-Verlag, 1995. ISBN 3-540-59450-9.

[12] Payne, S. J. and Green, T. R. G. Task-action grammars: a model of mental representation of task languages. Human-Computer Interaction, 2(2):93–133, 1986.

[13] Reisner, P. Formal grammar as a tool for analysing ease of use: some fundamental concepts. In Thomas, J. C. and Schneider, M. L. (Eds), Human Factors in Computer Systems, pages 53–78. Ablex, 1983.

[14] Roast, C. R. Executing Models in Human Computer Interaction. PhD thesis, Department of Computer Science, University of York, 1993.

[15] Roast, C. R. Modelling interaction using template abstractions. In Cockton, G., Draper, S., and Weir, G. (Eds), People and Computers IX, pages 273–284. Cambridge University Press, 1994.

[16] Roast, C. R. and Siddiqi, J. I. A formal analysis of an interface specification using the template model. Technical Report in preparation, 1995.

[17] Roast, C. R. and Siddiqi, J. I. The formal examination of cognitive dimensions. In Blandford, A. and Thimbleby, H. (Eds), HCI96 Industry Day & Adjunct Proceedings, pages 150–156, 1996. ISBN 1 85924 119 0.