Cognitive Dimensions and Multiple Notations

Tim Wright and Andy Cockburn

September 1, 2005

Contents

Abstract

Cognitive Dimensions is widely used framework to evaluate visual notations. Our research indicates that many programming environments let people program using multiple notations. Unfortunately, Cognitive Dimensions does not provide mechanisms to assess the relationship between the notations. In this paper we introduce our view of how notations are used in programming environments and we perform a preliminary extention of cognitive dimensions so it applies over multiple dimensions.

Much of this work is taken from Tim Wright's PhD thesis [21].

1 Classification Scheme

Central to our understanding of how notations are used in programming environments are two concepts: the activities people perform while programming and a method of specifying how the activities are supported by notations in a programming environment. We determined the fundamental activities by performing an activity-based decomposition of programming, and identified three fundamental activities: reading programs, writing programs, and watching programs run. We call our method of specifying how the activities are supported in a programming environment a Language Signature. This section introduces our three programming activities and Language Signatures. To aid the explanation of the activities and gulfs, we begin by introducing three programming environments that will be used as examples throughout this chapter: Logo, StageCast, and OpenOffice, and we examine how notations are used in these environments.

1.1 Sample Programming Environments

This section describes three sample programming environments and examines how notations are used in these environments. From these environments, we learn two things. First, that programming environments do use multiple notations and that using different notations could cause problems for users of the environments). Second, these notations are used for three separate programming activities: reading, writing, and watching. These three activities are the foundation of the framework for understanding how notations are used programming environments and are discussed further in the following section.

1.1.1 Logo

Logo, developed by Papert, is an environment where users edit text to control how a turtle moves

```
sub transpose
rem ----------------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------------------------
dim args1(1) as new com.sun.star.beans.PropertyValue
args1(0).Name = "Count"
args1(0).Value = 1
args1(1).Name = "Select"
args1(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoLeft", "", 0, args1())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Cut", "", 0, Array())

rem ----------------------------------------------------------------------
dim args3(1) as new com.sun.star.beans.PropertyValue
args3(0).Name = "Count"
args3(0).Value = 1
args3(1).Name = "Select"
The largest number of end-user programming environments use two notations.
args3(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoRight", "", 0, args3())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Paste", "", 0, Array())


end sub
```

Figure 1: Automatically generated OpenOffice.org macro to transpose two adjacent characters.

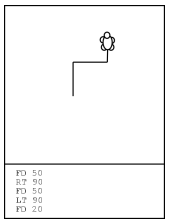

Figure 2: A mockup of the Logo Programming Environment. Users read and write programs using text, but see a turtle move around drawing lines when their program is executed.

around a screen and draws lines [12]. For example, a user might type:

```
REPEAT 4 [    Do the following 4 times
  FD 50       Move ForwarD 50 units
  RT 90       Turn RighT 90 degrees
]
```

and see a picture of a square. A mock-up Logo environment is shown in Figure 2.

The Logo environment contains two notations. The first is the textual description of what the turtle will do when the program is executed and is used when a user reads or writes a program. The second notation is the collection of lines that are drawn as the turtle that moves around the screen as well as the turtle itself drawing the lines. This notation is used for watching a program run

We postulate that the difference in notations between the reading/writing and watching activities could cause problems for a user as they must reason about one notation using another notation. For example, a user might ask themselves "*why did the turtle turn right instead of left*" and have to reason about their program by asking "*Is my mistake typing **LT** instead of **RT** or **FD** instead of **BW**?*". Later in this chapter we examine this type of usability problem, caused when users must interact with different notations for different programming activities.

1.1.2 StageCast

StageCast (formally called KidSim or Cocoa) is a programming environment for 2D visual simulations [17]. 2D simulations are programs that run in a two dimensional area of a screen where agents can move around and interact with other. An example simulation is shown in Figure 3. StageCast users write *before-after* rules to define behaviour

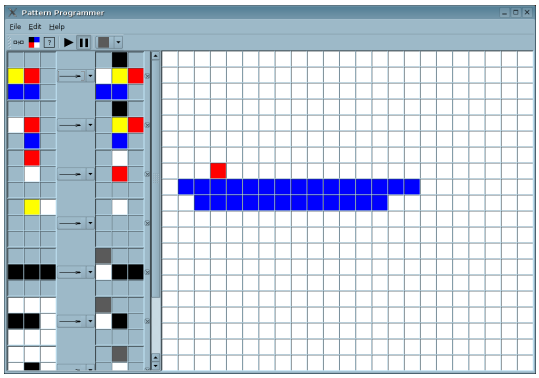

Figure 3: A 2D simulation. Many programming environments, including StageCast [17], AgentSheets [16], and Playground [4], use this domain for programming. The environment pictured (PatternProgrammer) was developed by Wright [20].

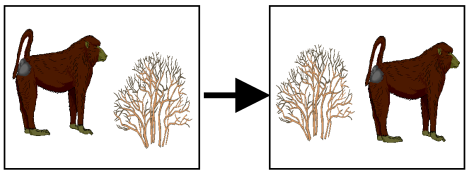

Figure 4: A before/after rule similar to those used in StageCast [17]. A user is writing a program to move a baboon past a bush. When StageCast executes, it will search the simulation for arrangements of agents matching the left-hand side of the rule and replace the agents with the arrangement of agents on the right-hand side of the rule.

in the simulation. A before after rule has two parts: an arrangement of agents to search for in the simulation, and an arrangement of agents with which to replace the found agents. An example rule is shown in Figure 4 (on page 3).

StageCast avoids problems of different notations by using the same notation for reading, writing, and watching programs: users write before-after rules by manipulating visual representations of agents and when users run their program they see the same set of agents move around and interact with each other.

Unfortunately, StageCast is not perfect. Usability studies of StageCast found that children have trouble predicting what StageCast will do when their program is executing [13]. The problem was caused by a difference in StageCast's rule scheduling algorithm and users' expectations of rule scheduling. This type of problem, where the way a program is executed is different from user's expectations of how it will be executed, is investigated in this chapter.

1.1.3 OpenOffice.org

OpenOffice.org is a free[1] word processing program with capabilities similar to Microsoft Word. In particular, OpenOffice.org provides functionality so users can program macros to manipulate their documents. Users program using the symbols provided by the environment: for example, a user wishing to program a macro to transpose two adjacent letters might turn on macro recording, select and cut the letter to the right of the mouse cursor, then move the cursor left and paste the letter. Users can then execute the macro and watch letters in their document change places. Unfortunately, if users discover a problem with their program they must either re-demonstrate the program from scratch or edit their program using a complex textual language: OpenOffice Basic (an example of OpenOffice Basic can be found in Figure 1 on page 2).

We postulate that this difference, between the notation used for writing and watching programs and the notation used for reading and writing the program, could cause usability problems for a user of OpenOffice. Again, this chapter examines this type of usability problem, caused by users interacting with different notations for different programming activities.

[1] OpenOffice.org is licensed under the GNU GPL.

1.2 Three Fundamental Activities

The previous section describes three programming environments: Logo, StageCast, and OpenOffice, and describes several ways programming environments can use notations. The three environments described use notations in very different ways: Logo uses one notation for editing programs and another notation for watching programs; StageCast uses the same notation for editing and watching programs; and OpenOffice uses one notation for writing and watching programs and another notation for editing programs. By decompose the editing task into two sub-tasks, reading and writing, we uncover three fundamental programming activities: reading, writing, and watching. These three activities are both fundamental to programming and fundamental to our framework for understanding how notations are used in programming environments. We now describe each activity in depth.

Reading is the act of viewing a notation describing program behaviour, writing is the process of using a notation to describe program behaviour, and watching is the act of viewing program behaviour, either viewing an animation of the notation or viewing the behaviour specified by the notation. The activities are shown in Figure 7. Also, we use the term the "program representation" to refer to any notation used for reading and we use the term the "program visualisation" to refer to any notation used for watching: program visualisations can range from animations of the code to agents interacting in a 2D simulation.

To use an example (described in the previous section and also shown in Figure 7), consider a user of a word processing program who wants to write a program to transpose two characters. First, using their word processor's programming by demonstration mechanism, they record themselves transposing two characters. This recording of their behaviour is the writing activity, and in this example they are writing using the icons and behaviour provided by their word processor. Next, they execute their macro a couple of times to transpose various characters that were out of order. This execution of their program is the watching activity. Again, in this example, they are watching their program using the icons and symbols provided by the word processor. After they have executed their macro several times, they discover that there is a bug in their macro: the macro transposes the two characters to the left of the cursor instead of the intended behaviour of the two characters

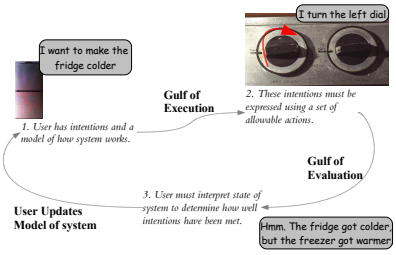

Figure 5: Donald Norman's gulfs of Execution and Evaluation. The gulf of execution is "the difference between user intentions and allowable actions" and the gulf of evaluation is "the amount of effort the the person must exert to interpret the physical state of the system and to determine how well the expectations and intentions have been met" [10]

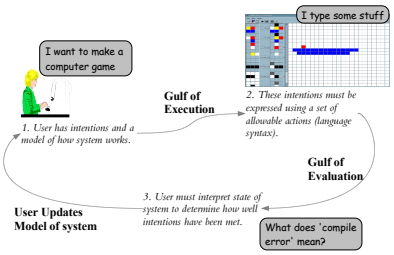

Figure 6: Donald Norman's gulfs of Execution and Evaluation in a programming context [10]. The gulf of execution is the difference between a user's model of desired program behaviour and how they must express their program to the computer while the gulf of evaluation is how hard it is for the user to figure out if they have expressed their program correctly.

surrounding the cursor. To fix their program, the user opens their macro in the macro editor to first read and understand it. This is the reading activity. In many current word processors, they will see their code in a textual form close to a conventional programming language. In this example, the program representation is the textual form they view to edit their macro while the program visualisation refers to the dynamic effects caused by the user executing their program.

1.3 Three Cognitive Gulfs

In 1988, Norman examined the real world for usability problems [10]. He identified two cognitive gulfs: the gulfs of execution and evaluation. He describes the gulf of execution as "the difference between user intentions and allowable actions" and the gulf of evaluation as reflecting "the amount of effort the the person must exert to interpret the physical state of the system and to determine how well the expectations and intentions have been met". For example, imagine someone trying to decrease the temperature of a fridge (Figure 5). First, they examine the dials on the fridge and note two dials. Assuming that one dial is for the fridge and the other dial for the freezer, they turn the left hand dial to the right. This difference between the user's intentions (make the fridge colder) and the allowable actions (two dials that can be turned left or right) is the gulf of execution. After letting the fridge's temperature stabilise, the user notices that while the fridge has become colder, the freezer has also become warmer. The effort the user must expend interpreting the system and understanding why their actions had a different effect than they expected is Norman's gulf of evaluation. Norman's gulfs are shown in Figure 5.

Figure 6 contextualizes Norman's gulfs in a programming context. While the gulf of execution is still applicable (users must translate their mental model into the symbols of a programming language), the gulf of evaluation has become more complex: a program has both a static representation and a dynamic visualisation that users must interpret to determine how well their intentions have been met. To distinguish the different gulfs of interpreting the representation of visualisation, we decompose Norman's gulf of evaluation into two gulfs: the gulf of representation and the gulf of visualisation.

So, in a programming context, Norman's two gulfs become three (and are shown in Figure 7). A gulf of expression is created when the user's mental model of desired program behaviour differs to how the user must express the program for reading. A gulf of representation is created when the user's mental model of program behaviour and the program representation. A gulf of visualisation is created when the user's mental model of program behaviour differs from the program visualisation.

As an example, consider a user writing a program to transpose two letters. If the user is using a word processor with a macro recorder, they can write the program using the icons and behaviour

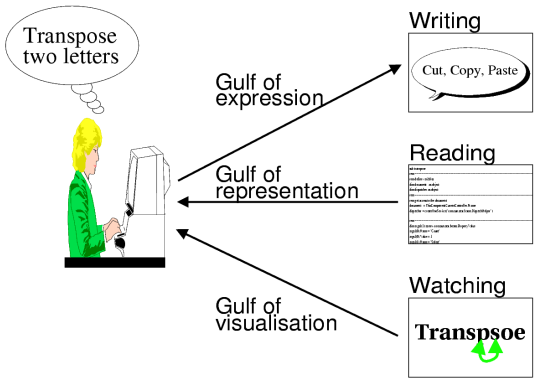

Figure 7: The gulfs of expression, representation, and visualisation and their associated activities. In this example a user is writing a program using programming by demonstration mechanisms. The user is reading an computer generated representation of their program. When they run their program they see letters in their document being transposed.

of their word processor. As they are likely familiar with the icons and behaviour of their word processor, we argue they have a low gulf of expression. However, imagine that their word processor had no macro recorder and the user had to write their program using a conventional textual language (like Visual Basic). Unless the user is experienced with this conventional textual notation, they will be unfamiliar with the notation and we argue the notation creates a high gulf of expression.

We define the gulf of visualisation as the cognitive difference between a user's mental model of program behaviour and the program visualisation (what the program actually does as it executes). To continue the example of a user writing a program to transpose two letters, consider the possibility that when the users executes their program, the program does not behave as expected. Although the program does not behave as expected, it uses the same symbols and behaviour that the user wrote the program in (the icons and behaviour of their word processor), so we argue that there is a low gulf of visualisation (in this example we are using the notation the user wrote their program in as a cognate for their mental model).

We define the gulf of representation as the cognitive difference between the users mental model of their program and the program representation. As an example, consider our user who is writing a program to transpose two letters. After they have written their program, they run their program and discover that it doesn't work as expected. When they view their program, they see conventional textual program code. Like the gulf of visualisation, we can use the notation the user wrote the program in as a cognate for their mental model. As the notation consisting of the icons and behaviour of a user's word processor is very different from a conventional textual programming language, we argue this example creates the risk of a high gulf of representation.

Smith, Cypher, and Tesler have also done work contextualising Norman's gulfs in a programming context [18]. They argue that the appropriate way to make programming easier is to reduce Norman's gulfs by moving the system closer to the user. Their assumption in this is that there is a consistent representation of the programming system: people write, read, and watch programs using the same notation. Our structure of the three activities (reading, writing, and watching) and the three gulfs (execution, representation, and visualisation) gives us flexibility to argue about systems that use different notations for the three activities, or even different notations for the same activity. We use an abstract syntax for how notations are used in programming environments called Language Signatures. Language Signatures are described in the following section.

1.4 Language Signatures

By describing how programming environments use notations for our three activities we can compare different programming environments and discover ways of using notations that help users when programming and ways of using notations that hinder users when programming. Unfortunately, the description of how different notations are used for different activities is long, increasing the chance that someone will misread a notation description. They also make comparing how notations are used in different programming environments harder. For an example description, consider Logo. Logo is a programming environment where programmers describe how a turtle moves around a 2D surface and draws lines. An example logo program with output is shown in Figure 2. The description of how notations are used in Logo is: *users read and write programs using textual commands and watch a turtle draw lines*.

To avoid parsing problems, and to help people compare different how different programming en-

vironments use different notations, we use Language Signatures. A Language Signature succinctly expresses how a programming environment uses different notations for the three fundamental programming activities. It is written inside square brackets with plus (+) symbols separating different notations. Each notation is described by stating the activities it supports, abbreviated to RE (Reading), WR (Writing), and WA (Watching) and separated by slash (/) symbols. For clarity, a textual description of the notation's symbols can be subscripted to the description of the activities supported by that notation. To continue our example, Logo's Language Signature is [RE/WR_{text} + $WA_{turtle,\ lines}$].

An alternative to the Language Signature syntax we decided on is to have the Language Signature describe first activities and second the notations used for that activity. The Logo signature might then look like this: [Re_{text} + Wr_{text} + $Wa_{turtle,lines}$]. This way of expressing Language Signatures suffers several flaws: it is hard to immediately determine the number of notations present in an environment; it is hard to determine if Logo uses the same notation for reading and writing, and it is hard to see which activities a notation does not support.

2 Advanced Multiple Notation Environments

2.1 Environments with multiple notations for reading

This section describes programming environments that have multiple notations for reading and one notation for writing. These programming environments all risk creating a gulf of expression: with multiple notations for reading and one for writing they must have at least one read-only notation. Two environments in this category overcome this gulf by providing an transient read-only notation: the program is read through the computer's speakers. Unfortunately only informal studies have examined the effects of a transient notation.

We found three programming environments in this category: AgentSheets, Pecan, and Mondrian.

AgentSheets is a programming environment for visual simulations [16]. Users describe agent behaviour with an iconic language. The AgentSheets environment can also read a program through the computers speakers, and has a [RE/WR/WA_{iconic} + RE_{spoken} + WA_{agents}] Language Signature. Anecdotal evidence about AgentSheets reveals that using multiple program representations helps users write and understand programs[2]. We believe that AgentSheets' spoken notation does not create a gulf of expression because the spoken representation is transient: users intuitively know they cannot edit a spoken representation. However, more research is needed to confirm this belief, and examine the effect of letting people write using a spoken notation: letting people dictate programs to computers (which would change AgentSheets' signature to [RE/WR/WA_{iconic} + RE/WR_{spoken} + WA_{agents}]).

Mondrian is a programming by example system for creating interactive drawings [8]. Users can program new commands into Mondrian by creating before-after rules. While users are creating these rules, Mondrian reads the users' commands through the computer's speakers, and Mondrian can also convert the user-defined rules into Lisp code.

Pecan produces a read-only Nassi-Shneiderman diagram from code [14]. Although we could not find any papers describing usability studies of Pecan, we argue that Pecan risks creating a gulf of expression because users might build a mental model of program behaviour based on the Nassi-Shneiderman diagram, and then want to edit the Nassi-Shneiderman diagram directly.

2.2 Environments with multiple notations for writing

Environments with multiple notations to write programs, but one way to read programs, risk creating a gulf of representation: users must map from the notation they used for writing to the notation they use for reading. We found one three-language environment with multiple ways to write a program: Leogo [2]. Leogo levers the gulf of representation to teach users to program and is pictured in Figure 8.

Leogo is an extension of Logo where users can program using three different notations: textual logo code, an iconic version of the code using buttons for commands and sliders for amounts, and a

[2] Informal conversation with Alexander Reppenning

Type	Description
Text	The notation is based on users typing or manipulating textual statements. The statements might be like conventional code or more natural. Example environments that use a textual notation are Alice [3], Hands [11], and C [7].
Iconic	Iconic notations are notations that use icons for programming statements or have animated icons for visualisations. Flowcharts programming notations are an example of iconic programming notations. Environments with an iconic visualisations include StageCast and PatternProgrammer (see Figure 3).
User Interface	Some programming environments use the set of symbols present in a standard user interface for writing and watching programs. Typically, these environments are programming by demonstration environments (PBD) where users can demonstrate behaviour using a standard user interface and the PBD environment attempts to infer the user's program and use the program at a later time.
Tangible	Tangible notations use (for program representation) physical items in place of program statements, or (for program visualisation) physical items that move and interact. An example environment that has a tangible representation is AlgoBlock: users join together physical cubes where each cube represents a single Logo statement [19]. An example environment that uses a tangible visualisation is Electronic Blocks: when a program is run, users see physical blocks make sounds, lights, move around, and interact with each other [23].

Table 1: Description of some common notation types used in Language Signatures.

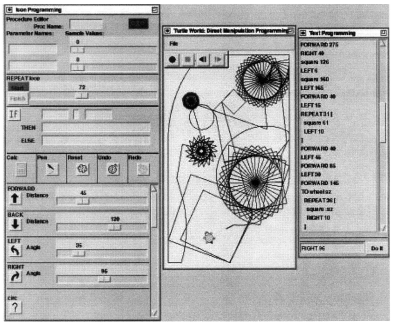

Figure 8: The Leogo programming environment. Users can manipulate any of three notations (textual, iconic, or directly-manipulate the turtle), and see the results immediately in the other two notations. For example, a user who wants to move the turtle forward 50 units can either drag the turtle forward 50 units and see the statement "FD 50" appear as well as the icon for forward movement depress and a slider advance to 50, or they can type "FD 50" and watch the changes in the iconic representation and the output domain.

direct-manipulation notation where users can manipulate the turtle immediately. Changes in one notation in Leogo are immediately reflected in the other two notations (see Figure 8). The rational behind using multiple notations was to help users lever their knowledge of how a turtle and iconic interfaces work to help children learn Logo. An evaluation of Leogo found that children could use Leogo, and that they tended to pick one notation and stay with that. Leogo's Language Signature is [WR/WA$_{turtle,\ lines}$ + WR/WA$_{iconic}$ + RE/WR/WA$_{text}$], and is shown in Figure 8.

One of Leogo's motivations was to help aid knowledge transfer from a notation children would be familiar with (the way the turtle moved) to a notation that provided more power (the textual notation). Both these environments provide multiple notations for reading and writing—one close to the task domain and one close to conventional code. The designers argue that this decision is useful as it can aid knowledge transfer from domain specific knowledge to knowledge about conventional code. Unfortunately the designers of the two environments did not perform an evaluation of the successes of the knowledge transfer.

Figure 9: A screen snapshot of Mulspren: A MUltiple Language Simulation PRogramming ENvironment.

2.3 Environments with Multiple Notations for Reading and Writing

Mulspren is a three-language environment that uses multiple languages to read and write programs [22]. Mulspren avoids the risk of creating gulfs of expression, representation, and visualisation by using duel languages for writing and reading: users can move seamlessly between two textual languages modifying either and viewing the changes in both. Both languages are animated when a program is executed. We plan to use Mulspren to evaluate the effects of multiple languages on users without the hindrance of gulfs of representation and expression. A screen snapshot of Mulspren is in Figure 9.

2.4 Environments with an Arbitrary Number of Notations

The only environment we reviewed that supports an arbitrary number of notations is the Garden programming environment [15] (Garden is a successor of the Pecan programming environment, which is described in the previous section). Users of Garden write programs in one of any number notations, and can add new notations to the Garden environment. Reiss describes how a user can extend Garden so the user can program using petri-nets. Despite this expressive power of Garden, users must use the same notation to read their program as the one in which they wrote their program. This means that Garden provides n notations for writing but only one for reading—the notation that the user wrote their program in. We write Garden's Language Signature as [(WR+WR+...)/RE] rather than [RE/WR + RE/WR + ...], as the latter conveys that users can move between the different notations after they had started writing code.

Unfortunately no user studies were performed on Garden, so we can neither analyse what the effects of user-extensible environments are on users nor examine what gulfs are created in a user-extensible programming environment.

3 Green and Petre's Cognitive Dimensions

Cognitive Dimensions is a framework developed to analyse visual programming notations and the programming environments used to create, modify, and execute the notations [6]. The framework has thirteen dimensions, ranging from error-proneness (how easily can programmers make errors) to viscosity (how easily can programmers change an existing program). The framework was developed to create a set of heuristics that an expert can use to critique a programming notation. In this way, they are related to the heuristics used for ordinary user-interface evaluation [9], but are tightly focused on usability issues related to programming notations.

Cognitive dimensions and Language Signatures examine different and complimentary aspects of notation use in programming environments. Whereas Cognitive Dimensions provides heuristics to examine the usability of a particular notation, Language Signatures provide heuristics to examine how multiple notations in an environment notations interact with each other and with the user. This difference in scope of the two frameworks means that both are useful when designing and building programming environments.

While the Cognitive Dimensions framework was created with the goal of analysing programming notations, some of the dimensions can be extended to analyse the the relationships between programming notations.

Abstraction Gradient. The abstraction gradient dimensions analyses the minimum and maximum levels of abstraction, and looks at the ability for a programmer to abstract fragments of a program. An extension of this dimension to multiple notations would examine

the relationships between the abstraction gradient of each notation, and look what happens to one notation when a programmer abstracts part of the program in the other notation.

For example, consider a programmer who is working in a multiple notation programming environment and wants to create a new abstraction by refactoring some common code into a method. Conventionally, the abstraction gradient dimension would examine how much work the programmer must do to perform the refactoring. However, in a multiple notation programming environments, a environment designer must also consider the effects on the other notations of the refactoring.

Closeness of Mapping. This dimension examines the mapping between the problem world and the syntax and semantics of the programming notation. The extension for multiple notations also examines the closeness of mapping between the multiple notation: how much cognitive effort a user must expend when switching notations.

Error-proneness. This dimension examines how easy it is to make an error, and more importantly how easy it is to recover from an error. An extension of this dimension into a multiple notation system would examine the effects of making an error in one notation on the other notation.

For example, consider a user who makes an error in one notation of a multiple notation programming environment. An analysis of the error-proneness dimension in a multiple notation environment should consider questions including examining the effects of the error on the other notations, and the recoverability from the error using the other notations.

Hard Mental Questions. Multiple notation programming environments can create many additional cognitive tasks that users must overcome to be able to use the environment. These were enumerated by Ainsworth *et al* and include: understanding the relationships between representations and the domain, translating between representations, and, if designing representations, selecting and constructing an appropriate representation [1]. A cognitive dimensions analysis of multiple notation programming environments should include an analysis of how hard these tasks are for a user to perform.

Role Expressiveness. This dimension refers to the ease in which programs can be read (as opposed to Hard Mental Questions or Closeness of Mapping which refer to the ease in which a program can be written). In a multiple notation programming environment, users of this dimension to evaluation notation usability should examine how easy it is for people to understand the relationship between the notations as well as read the individual notations.

Secondary Notation and Escape From Formalism. A secondary notation refers to extra information that is not part of the actual program: commenting and indentation. Green and Petre argue that support for secondary notation is important for programming notations. In a multiple notation programming environment, a programmer using this dimension to analyse the notations should consider the effects of modifying a notation on the comments and layout of the other notations.

Viscosity. Viscosity refers to a notations resistance to local change, or to the ease in which a programmer can make small changes to a program. In a multiple notation programming environment, small changes in one notation could lead to large changes in another notation. This high inter-notation viscosity is especially likely if the two notations use very different representations of the program. For example, consider a multiple notation programming environment with two representations of a program: a control flow representation and a data flow representation. A simple change in the data flow representation could equate to a large change in the control flow representation. Programmers analysing multiple notation programming environments should consider the effects of small changes in one notation on the other notation.

4 Summary

This paper introduced our view of how notations are used in programming environments and performed a preliminary analysis of how cognitive

dimensions could be extended to analyse the relationship between notations in a programming environment. We believe that extending the analysis of cognitive dimensions would make interesting reading in the special issue journal on cognitive dimensions.

References

[1] S.E. Ainsworth and N Van Labeke. Using a multi-representational design framework to develop and evaluate a dynamic simulation environment. In Dynamic Information and Visualisation Workshop, Tuebingen, July 2002.

[2] A. Cockburn and A. Bryant. Leogo: An equal opportunity user interface for programming. Journal of Visual Languages & Computing, 8(5–6):601–619, 1997.

[3] Matthew Conway, Steve Audia, Tommy Burnette, Jim Durbin, Rich Gossweiler, Shuichi Koga, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James Patten, Joe Shochet, David Staak, Richard Stoakley, John Viega, Jeff White, George Williams, Dennis Cogrove, Kevin Christiansen, Rob Deline, Jeff Pierce, Brian Stearns, Chris Sturgill, and Randy Pausch. Alice: Lessons Learned from Building a 3D System for Novices. In Human Factors in Computing Systems: CHI 2000 Conference Proceedings (USA), pages 486–493, April 2000.

[4] Jay Fenton and Kent Beck. Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. In Proc. OOPSLA '89, pages 123–137, New Orleans, Louisiana, United States, 1989.

[5] Ephraim P. Glinert, editor. Visual Programming Environments, Paradigms and Systems. IEEE Computer Society Press Tutorial, 1990.

[6] T.R.G. Green and M. Perte. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. Journal of Visual Languages & Computing, 7:131–174, 1996.

[7] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice Hall, 1978.

[8] Henry Lieberman. Watch what I do: Programming by Demonstration, chapter 16: Monderain: A Teachable Graphical Editor. MIT Press, 1993.

[9] Jakob Nielsen. Usability Engineering. Morgan Kauffman, 1993.

[10] DA Norman. The Psychology of Everyday Things. London: Basic Books, 1988.

[11] John F. Pane, Chotirat "Ann" Ratanamahatana, and Brad A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies, 54:237–264, 2001.

[12] S Papert. Mindstorms — Children, Computers, and Powerful Ideas. Harvester Press, Brighton, 1980.

[13] Cyndi Rader, Cathy Brand, and Clayton Lewis. Degrees of Comprehension: Children's Understanding of a Visual Programming Environment. In Human Factors in Computing Systems: CHI '97 Conference Proceedings (USA), pages 351–358, March 1997.

[14] S.P. Reiss. Pecan: Program development systems that support multiple views. In [5], pages 324–333. IEEE Computer Society Press Tutorial, 1990.

[15] S.P. Reiss. Working in the garden environment for conceptual programming. In [5], pages 334–345. IEEE Computer Society Press Tutorial, 1990.

[16] Alexander Repenning. Agentsheets : an Interactive Simulation Environment with End-User Programmable Agents. In Proceedings of the IFIP Conference on Human Computer Interaction (INTERACT '2000, Tokyo, Japan), 2000.

[17] David Canfield Smith, Allen Cypher, and Jim Spohrer. KidSim: Programming Agents without a Programming Language. Communications of the ACM, 37(7):54–67, July 1994.

[18] David Canfield Smith, Allen Cypher, and Larry Tesler. Novice Programming Comes of Age. Communications of the ACM, 43(3):75–81, March 2000.

[19] Hideyuki Suzuki and Hiroshi Kato. Interaction-Level Support for Collaborative Learning: AlgoBlock — An Open Programming Language. In John L. Schnase, editor, Proc. Computer Supported Collaborative Learning '95, pages 349–355, Bloomington, Indiana, October 1995.

[20] Tim Wright. Pattern programmer. Unpublished programming environment.

[21] Tim Wright. Collaborative and Multiple-Notation Programming Environments for Children. PhD thesis, University of Canterbury, 2005.

[22] Tim Wright and Andy Cockburn. Mulspren: a MUltiple Language Simulation PRogramming ENvironment. In IEEE Symposia on Human-Centric Languages and Environments, pages 101–103, Arlington, Virginia, September 2002.

[23] Peta Wyeth and Helen C. Purchase. Programming Without a Computer: A New Interface For Children Under Eight. Proc. Australian Computer Science Conference, 22(5):141–148, 2000.