ACM SRC 2012: Detailed Research Outline

'Misfits in Abstraction: Towards User-centered Design in Domain-specific Languages for End-User Programming'

Hiroki NISHINO
Graduate School for Integrative Sciences and Engineering
National University of Singapore
CeLS, #05-01, 27 Med. Drive Singapore 117456
g0901876@nus.edu.sg

1. Problem and Motivation

The Ph.D study by the author is a design of a new computer music language and partly discuss how a new language can be designed with better usability for computer musicians. Computer musicians can be classified as 'expert end-users', the type of users who possess expertise in their own professional domain, but not in programming [6]. As the U.S. Department of Labor estimates 30 % of new jobs could require programming skills by 2012 [1], how a DSL can be designed with better usability for expert end-users is becoming an important issue.

The design of a domain-specific language (DSL) for exert end-users propose interesting research problems. Firstly, there is a problem how user centered design for programming language can be made possible. Secondly, there is a question how such a DSL for expert end-users can be evaluated in this perspective of user centered design. As Markstrum discusses in [15], *claim-evidence correspondenc*e for usability claims has been rarely endorsed in programing language design history. Thirdly, expert end-users largely differ from 1st computer science students or children as Blackwell argues [6]. There is a significant necessity to take the characteristics of expert end-users in the target application domain of a DSL.

The author's work for SRC 2012 describes the early phase of this study on these issues. This document also described the further extension and discussion.

2. Background and Related Work

2.1 Abstraction and Usability Problems

Abstraction can cause usability problems when it is incompatible with the user's conceptualization. As Blackwell discusses, "*even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centered design"* [5].

Blandford and her colleagues developed a usability evaluation method called CASSM (Concept-based Analysis of Surface and Structural Misfits), " *the purpose of which is in the identification of misfits between the way the user thinks and the representation implemented within the system"* [8].

Such an view to abstractions is also highly desirable to domain-specific language design for expert end-users; As programming activity is modeled as a problems solving activity and programmers are considered to "*use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping"* [11, p.22], it is of significant importance to avoid such conceptual misfits between expert knowledge and programming language design.

2.2 Abstraction Layers and DSL Design

In [14], Lee argues that *"the core abstraction of computing need to be rethought to incorporate essential properties of the physical systems, most particularly the passage of time"*. He discusses the physical passage of time is "*a key aspect of physical processes almost entirely absent in computing*" and being lost in abstraction layers in computing and this *"failure of abstraction"* is the cause of significant "*loss of predictability and repeatability*. Lee discusses such loss is mostly due to that 'time' is abstracted away in the abstraction layers in computing. While Lee mostly discusses such an abstraction issue mainly for *Cyber Physical Systems*, in which predictable and repeatable behaviors in timing can be crucial, his perspective that abstraction layers can cause a significant problems is also important for DSL design, since DSLs are normally built on some underlying software frameworks or libraries.

In [12], Fowler discusses such a view to DSLs as following. "*Another way of looking at a DSL is as a way of manipulation an abstraction. In software development, we build abstractions and then manipulate them, often on multiple levels. The most common way to build in abstraction is by implementing a library or framework"* and *"In this view a DSL is a front-end to a library providing a different style of manipulation to the command-query. In this context, the library is the Semantic Model of the DSL"*. Thus, since a DSL can be considered such a '*front-end*' to the underlying framework, which constitute the semantic model of the target domain, if there exists any *conceptual misfit* as Blandford discusses in her CASSM approach between the users' conceptualizations in expert knowledge and the underlying software framework, it can appear as usability problems in DSL design.

Such a characteristic of DSLs should not be considered lightly. Unlike general purpose programming languages, in which programmers are supposed to build abstractions by themselves, DSLs provides predefined 'semantic model'. If there is any entity that exists in the expertise knowledge doesn't have any counterpart in the semantic model, this can hinder expert end-users programmers

from mapping between the problem domain and the computing domain. Furthermore, since the library/framework design often involves the abstraction layers, it is also of significance to consider if there exists any problematic abstractions.

For instance, if there may be some entity which is abstracted away or made hardly accessible in the layers of abstractions in the semantic model that the underlying library or framework provide, but the counterpart entity in expert knowledge is involved in problem solving, then this can appear as a usability problem in a DSL. For another instance, if there exists any layers in semantic model that doesn't exist in the organization of expert knowledge, this may also cause some difficulty in programming activity; Thus, it is of significant importance to consider *conceptual misfits* as Blandford discusses in the design of DSL for expert end-users.

2.3 Related Work in HCI

Though programming language design in the context of user centered design has been rarely discussed compared to other topic such as GUIs, researchers have been making considerable efforts to investigate usability issues in programming languages. There exist the communities such as *Psychology of Programming Interest Group* [23] and *Empirical Studies of Programmers Workshop.* Yet, there is still a significant question as Blackwell and Green discuss; *"how do these research results get applied by the people who design new user interfaces?"* [3] and the "*designers must be able (and interested) to interpret and apply theoretical results"*.

Yet, such an interpretation is a considerably hard task for programming language designers, who are not HCI specialists, especially because programming language design can be more difficult to apply results from HCI because programming languages are significantly complex information artifacts. As Blackwell and Green discuss, "*HCI has a gap"* in this area and "*there is no approach that addresses all types of activity, that can lead to constructive suggestions for improving the system or device, that avoids details, and that allows similar problems to be readily identified in very different situations."* [3].

For instance, McIver designed GRAIL, a programming language designed to minimize syntax errors and compared GRAIL to LOGO [16]. While she found the groups using GRAIL made less errors then groups using LOGO, it is questionable if such a study can directly applicable to programming language design in the other situations, as error rates is only one of many aspects in programming activity and there is a certain trade-off with the other aspects of language design. Furthermore, the improvement in error rates itself doesn't say anything about how much each part of the language design actually contributes to the improvement.

For such a necessity for *broad-brush analysis* that can avoid such *death-by-details* situations, Green and his colleagues developed the framework called *Cognitive Dimensions of Notations*. The framework can offer "*terms that were readily comprehended by non-specialists"* and can describes the trade-offs between different aspects of usability problems, and is applicable "*not just to interactive systems but also to paper-based notations and other non-interactive systems"* [8]. Cognitive Dimensions of Notations framework provides the '*dimensions'* for broad-brush usability analysis and the trade-offs between these dimensions can be clarified for further discussion in notation design. The framework has been applied to the evaluation of a new programming language [10] or the prototyping phase of a end-user programming environment [4]. Sadowski mixed Cognitive Dimensions of Notations together with Nielsens' heuristics [17] for evaluation of additional language features in [26].

3. Approach and Uniqueness

3.1 Uniqueness

While there are many notable works in novice programming such as 'natural programming' by Pane and Myers [21], the scope of the interests of this SRC work is in designing a new DSL for computer musicians. As Blackwell discusses in [6], the research on expert end-users should be distinguished from the research on 1st year computer science students on children, as they "*may not be directly relevant to needs of expert end-user programmers"*. At the same time, the scope of this SRC work significantly differs from those previous study to investigate the usability of the additional features of programming environment [22] or language [26]. There is a significant focus on the investigation the analysis of how conceptual misfits between expert knowledge and language design can lead to usability problems and how such misfits can be removed to improve usability. The uniqueness of the work lies in such a focus on expert end-users and conceptual misfits.

3.2 Approach

Yet, unlike GUIs or HIDs, with which designers can assume the set of the tasks that users may perform to a considerable degree, what end-users may want to solve by programming is largely unforeseeable; if one can assume a finite set of the tasks that users would perform, then there would be not much necessity to provide a programming language to users and probably be much better simply to design software that can perform the tasks; Thus, exhaustive task analysis and collection of user needs may be not much adequate, as they are largely unforeseeable unlike GUIs or other software.

It is also considered important to provide *claim-evidence correspondences* for language design if it is related to usability issues as possible; as Markstrum discusses such an effort has been significantly lacking in programming language design research [15].

The SRC submission only discussed the concept in analysis of usability problem in language design to discuss conceptual misfits in a DSL and how the analysis and removal of such conceptual misfits can be beneficial. The work is already extended to discuss to the design process that integrates HCI practices and being experimentally applied to a new computer music language design, which is the main focus of the author's Ph.D study. This language design process and early iteration of the design is described in [19]. Figure 1 below shows the overview of the design process.

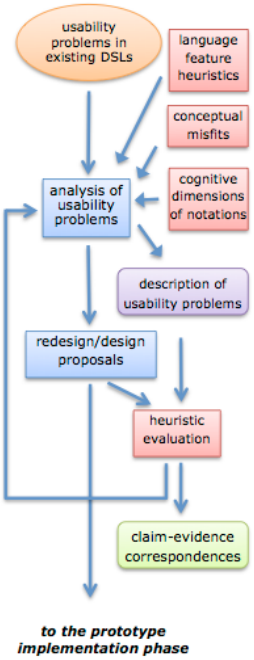

Figure 1. The overview of our experimental design process

First, the process begins with the analysis of the usability problems in the existing programming languages. Since the case study performed in [19] is on computer music language design, there are several common well-known usability problems in major computer music languages. This approach is taken because potential problems in programming language design are largely unforeseeable and it is unlikely to be realistic to perform exhaustive tasks analysis and collection of user needs. Moreover, such an approach can be beneficial to investigate if there is any conceptual misfit between expertise knowledge and the representation implemented within the libraries/frameworks under a DSL, to avoid the same problem in a new language.

Then, Such usability problems will be described and analyzed by the frameworks such as Cognitive Dimensions of Notations and Sadowski's language feature heuristics [26], with consideration on conceptual misfits. These frameworks also provide vocabulary to clarify the usability problems/analysis and design claims/design proposal for new languages.

After the analysis phase, external evaluators perform heuristic evaluation. The evaluators are chosen from the experts with appropriate background in the target domain. The evaluation phase is performed using the same framework applied for the analysis phase. The evaluators evaluate both the original examples with the usability problems in the existing languages and the redesign proposals, without being informed the designer's analysis and the design claims. By using the same framework both for design and evaluation, claim-evidence correspondence for usability issues can be provided as evaluation results. After the evaluation, the designer performs the next iteration of the design process and repeats the cycle until the design is brushed up good enough for the implementation phase.

4. Results and Contributions

4.1 Results

The above language design process is already being under actual design iterations of a new computer music language. First, the author began with one of the well-known usability problem of *single sample feedback* frequently seen in the existing computer music languages as described in the SRC work and discussed more in detail in [18]. This paper analyzes conceptual misfits between *Road's nine musical time-scale* [25, p3] as expertise knowledge and the representation implemented within the underlying framework of *SuperCollider* computer music language [28].

While the conference paper is mainly discussing the problem of *single sample feedback*, the analysis suggested another conceptual misfit in so-called *microsound synthesis* in computer music programming. As implied by this misfit, computer music researchers have been arguing such difficulty in software design and implementation of microsound synthesis and it is considered as a major concern in computer music language/software design [2,9,27]. Furthermore, the previous study do not discusses such difficulty in implementing microsound synthesis as a conceptual misfit in the representations between expertise knowledge and software/language design. This is considered as a significant redesign opportunity in computer music language, as the previous study are still largely depends on the sound synthesis framework design based on the traditional *unit-generator* concept [24, p.1234] which was invented almost 40 years ago and the concept significantly differs that of microsound synthesis techniques[1].

The design proposal for a new language by the author considered such a misfit and integrated the related objects and manipulation for microsound synthesis techniques into language design. Such objects and manipulation for microsounds has not considered in the related work in computer music. The evaluations by external experts were very positive to this novel abstraction and well endorsed the design claims by the author. At the same time, the evaluations also proposed sound criticisms for further improvement. The suggestions by this evaluation phase will be considered in the next iteration of language design phase.

The first prototype for this new computer music language is currently under development and the next iteration of design phase is also planed in the near future.

4.2 Contribution

While the SRC work is from the very early phase of the Ph.D study by the author and only briefly describes the concept, it already focuses on the removal of conceptual misfits between expertise knowledge and language design with consideration of abstraction layers and on expert end-users. Such an aspect of the work significantly differs from the previous study that focused rather on end-user in general and general purpose programming languages. As many users who write programs today to solve the problems for their own professions and they can be considered as

[1] See [25, p.57] for more detailed discussion.

expert end-users, such a study as the author performed can be important to facilitate programming activity by expert end-users.

While the language design process is of the type of *broad-brush analysis,* the approach to begin with the usability problems in the existing languages and to integrate HCI practices both in language design phase and evaluation phase seems to be very beneficial as in our case study to design a new computer music language. The language design process also can provide a good claim-evidence correspondence for desirable language properties such as practicality, simplicity, familiarity, and readability, which have been rarely endorsed by evaluations as Markstrum argues [15]. Such a language design process that provide claim-evidence correspondence for usability related issues can be beneficial for sound practices for programming language design, especially for ones designed for end-users.

It should be also noted that the design process itself can be entirely by 'pencils and papers' as in many other prototyping methods. Since the cost for implementing programming languages can be large and the cost the modification of the library/framework under a DSL can be also significantly expensive, such a language design process that can provide sound criticism from experts with clear description supported by HCI practices is considered significant. As Norman discussed in [20], the designers hardly can avoid any misconceptions on users as they are expert in what they are designing. This is also very true to domain-specific languages for expert end-users since the designers are assumed to be experts both in programming and target domain. There is a significant necessity to involve external evaluations to avoid false assumptions on users of programming language to be designed.

5. Future Work

Since the main topic of the Ph.D study by the author is not usability in a DSL itself and in the development of a new computer music language, this study is considered as a part of the research framework practiced to guide the design phase and provide claim-evidence correspondence to the usability-related claims in the language design. The further study is focusing more on the new programming concept for computer music and technical contributions in computer music. Yet, since even such technical issues can appear as usability claims in the layer of programing language, the further iteration of design and evaluation phase is also planed during the Ph.D study.

6. REFERENCES

[1] "Occupational Outlook Handbook," U.S. Dept. of Labor, Bureau of Labor Statistics 2004. http://stats.bls.gov/oco.

[2] Bencina, R. Implementing Real-Time Granular Synthesis. In *Audio Anecdotes III*, A.K Peters. 2006

[3] Blackwell, A.F. Psychological issues in end-user programming. In *End User Development*. Kluwer Academic. 2006

[4] Blackwell, A.F., Burnett, M.M. and Jones, S.P. Champagne Prototyping: A re-search technique for early evaluation of complex end-user programming systems. In *Proc. VL/HCC04*. 2004

[5] Blackwell, A.F., Church, C. and Green, T.R.G. The Abstract is 'an Enemy': Alternative Perspectives to Computational Thinking. In *Proc. PPIG08*. 2008

[6] Blackwell, A.F., and Collins, N. The Programming Language as Musical Instrument. In *Proc. PPIG'05*. 2005

[7] Blackwell, A.F. and Green, T.R. Notational Systems – the Cognitive Dimensions of Notation framework. In *HCI Models, Theories and Frameworks: Toward a Multidisciplinary Science*, Morgan Kaufmann. 2003.

[8] Blandford, A. et al. Evaluating System Utility and Conceptual Fit Using CASSM. In *Intl Journal of Human-Computer Studies, Vol.66.* 2008. pp.393-409

[9] Brandt, E. Temporal Type Constructors for Computer Music Programming. *Ph.D Thesis*. Carnegie Melon University. 2002

[10] Clarke, S. Evaluating a new programming language. In *Proc PPIG01*, 2001

[11] Détienne, F., Software Design - Cognitive Aspects. Springer Verlag, 2001

[12] Fowler, M. Domain-Specific Languages. Addison- Wesley. 2010

[13] Green, T.R. and Blackwell, A.F. Cognitive Dimensions of Information Artefacts: a tutorial. 1998.

[14] Lee, E. Computing Needs Time. *Communications of the ACM, Vol.52, No.5*. 2009

[15] Markstrum S., Staking Claims: A History of Programming Language Design Claims and Evidence. In *Proc. PLATEAU*10, 2010

[16] McIver, L. The Effect of Programming Language on Error Rates of Novice Programmers. In *Proc. PPIG'02.* 2002

[17] Nielsen, J. Usability Inspection Methods. Wiley, 1994

[18] NISHINO, H. Conceptual Misfits in Computer Music Programming. In *Proc. Asian Computer Music Project*. 2011

[19] NISHINO, H. How Can a DSL for Expert End-UsersBe Designed for Better Usability? : A Case Study in Computer Music. In *Proc. SIGCHI12*. 2012

[20] Norman, D. The Design of Everyday Things. Basic Books. 2002.

[21] Pane, J.F, Ratanamahanatana, A.C. and Myers, B.A. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. In *Intl Journal of Human-Computer Studies*, *Vol.54*, 2000, pp.237-264

[22] Pane, J.F., Myers, B.A. and Miller, L.B. Using HCI techniques to design a more usable programming system. In *Proc. HCC02,* 2002

[23] Phychology of Programming Interest Group (PPIG), http://www.ppig.org/

[24] Roads, C. The Computer Music Tutorial. The MIT Press. 1996

[25] Roads, C. Microsound. The MIT Press. 2004

[26] Sadowski, S and Kurniawan, S. Heuristic Evaluation of Programming Language Features: Two Parallel Programming Case Studies, In *Proc. PLATEAU11*. 2011

[27] Wakefield, G.D. and Simith, W. Using Lua for Multimedia Composition, In *Proc*. *ICMC07*. 2007

[28] Wilson, S. et al. The SuperCollider Book. The MIT Press. 2011