DOMAIN SPECIFIC LANGUAGE FOR MUSIC

A Literature Review

Presented to

Dr. Sharmin Khan

Department of Computer Science

San José State University

In Partial Fulfillment

Of the requirements for the Class

CS 200W

By

Mayuri Wadkar

April 2018

Abstract

Programming languages have been rarely considered from the perspective of user-centered design. The systematic literature review conducted in this paper focuses on design of a user-centered domain-specific language (DSL) for computer music. In this survey we also explore the limitations of existing software synthesizers and DSLs. The aim is to analyze current techniques employed to leverage the development of DSL for music composition. The rationale of these techniques and challenges in the DSL approaches addressed by the used techniques have been investigated. This analysis can be used for redesigning of the DSL and design of a new DSL. A synthesis describing the main trends in all the topics mentioned above has been done in this paper.

Index Terms—**Programming language, domain specific language, music, software synthesizers, user-centered design.**

TABLE OF CONTENTS

I. INTRODUCTION

Traditionally music composition solely relied on human creativity and hardware synthesizers. Advances in technology gave rise to computer music which is the application of computing technology in music composition, born of the affordability of portable computer systems powerful enough for real-time signal processing. It helped composers create new music with the help of algorithmic composition programs. The use of computer for digital music performance using software synthesizers has been practiced since the early 90s, particularly in Japan [1]. However, an easy to use and expressive programming language specific to music domain will increase a musician's productivity and it will tackle the usability problems with software synthesizers. Yet domain specific languages (DSLs) have not been widely accepted in the music domain due to issues with language design [2].

In this survey, we took an approach that starts with understanding of non-trivial usability problems in existing DSLs and then echo the analysis in redesigning of existing DSLs or designing of a new DSL. Based on the outlines in prior Human Computer Interaction (HCI) studies, such an approach can be very valuable for the clarification of design criteria and improved claim-evidence correspondences in language design.

Since the introduction of DSLs, varying DSL implementations have emerged. In this survey, we explore various software synthesizers and DSL implementations. The survey is aimed at addressing the following research questions:

RQ1: Why is there a need of DSL for music domain?

RQ2: Which DSLs have been created and are available for use in music domain?

RQ3: What good features have been provided in existing DSLs and software synthesizers?

RQ4: What are the limitations of current DSL implementations and software synthesizers?

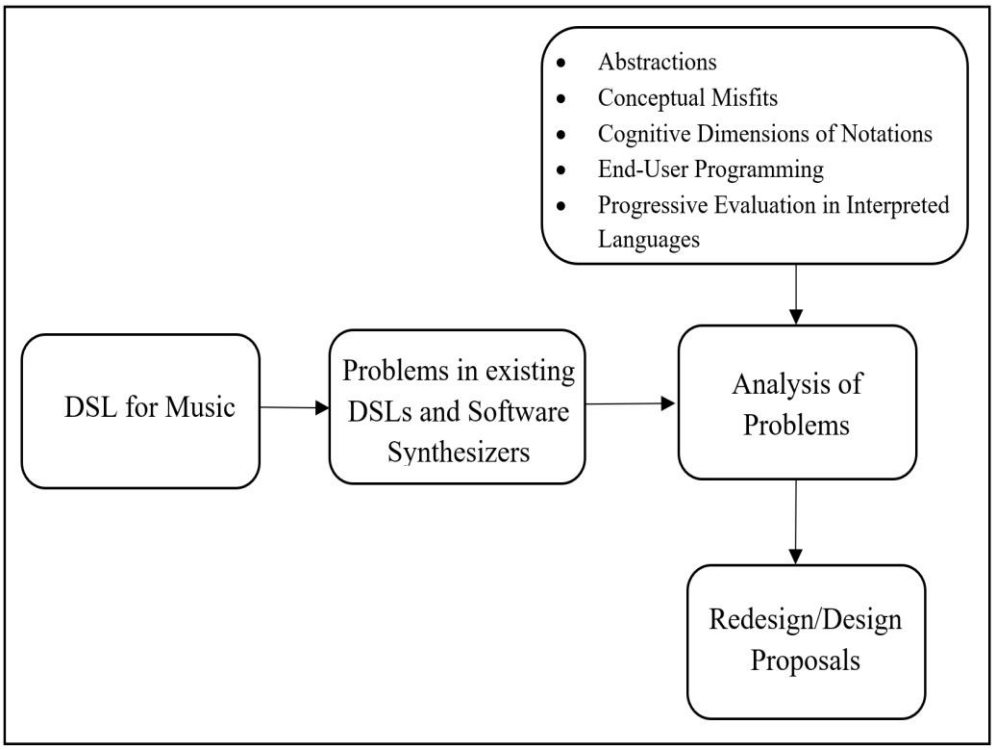

Fig. 1. Conceptual map of literature review

The organization of this review will follow the conceptual map as shown in Fig. 1.

II. DOMAIN SPECIFIC LANGUAGES FOR MUSIC

A. Domain Specific Language

In computer software, general purpose programming languages do not include language constructs designed to be used within a specific application domain. Conversely, a domain specific programming language is designed to be used within a specific application domain. Domain-Specific Languages (DSLs) offer significant gains in expressiveness and ease of use compared with general purpose programming languages. People find DSLs valuable because a well-designed DSL can be easier to program with than a traditional library. This improves programmer productivity, which is always valuable. However, there are a few issues in DSL approaches that make it difficult for the DSLs to be accepted and used in industry [3]. These issues are listed below.

- Interoperability with other languages
- Formal semantics
- Learning curve
- Domain analysis
- Scalability
- DSL evolution

There is a strong relation between DSLs and model-based engineering. DSL is defined as a set of coordinated models [4]. This is why, even though some of the studies selected for the literature review do not mention DSLs, but only discuss the connection between DSLs and metamodeling.

B. Issues with Existing DSLs

Popular computer music programs like Ableton Live and Reason offer sequencing, triggering and processing controls with rigid interfaces, but do not have the algorithmic maneuverability. Ableton Live software synthesizer has an attractive integrated GUI, of a style that is typical both of multimedia production software and of laboratory instrumentation systems. Ableton offers high visibility of operations, employing music technology paradigms of the virtual mixer, time against track event sequencer, with MIDI piano roll and waveform editors. However, to use these features one must purchase a full license software which costs 749 USD. Also, the abstraction hating interface of Ableton makes explicit assumptions about the music it will create. It has default rhythm set to 120 beats per minute with a 4/4-time signature, which is not tunable and typical of a disco music.

Yet difficult to master, but with exploratory potential, come programming languages for music like SuperCollider [5] or ChucK [6]. SuperCollider is a Smalltalk derived textual language

with C-like syntax and ChucK is a concurrent threading language developed to enable on-the-fly programming. These languages have adapted notations and language constructs which are more for people who have programming expertise. ChucK is run as an intimidating command line executable, for which input code must be written in a separate text editor. Operations available in ChucK are accessible via a separate web page detailing what possible instructions can be typed on the command line and in source code files. As all operations in Chuck are presented identically regardless of musical function, it has poor *closeness of mapping* and *role expressiveness*. This results in a very steep learning curve especially for someone new to ChucK.

III. IMPROVING DSL DESIGN

There are many factors involved in the design of a good DSL: syntax definition, correct semantics, tooling, methodology and documentation [7]. Depending on the executability level of the DSL [8] and the type of the DSL (internal or external) [7], these factors have different importance levels.

There are five stages in the DSL development process: decision, analysis, design, implementation and deployment. Different patterns have also been identified for the first four stages of the DSL development process. For example, patterns that occur in the decision stage are the need for: a new notation, a domain-specific analysis, verification, optimization, parallelization and transformation, etc. [8]. The DSL development process requires the domain and language development knowledge and thus is seen as a hard endeavor. The decision to develop a new DSL is thus not easy. In contrast, the language workbenches are becoming powerful and therefore, they decrease the effort put into the design of a new DSL. Language workbenches are tools that aid developers in defining and using DSLs efficiently [9]. These tools offer productive DSLs and APIs for the definition of their IDEs and languages.

In upcoming sections, we are proposing ideas which can be incorporated in the design of new DSL or while redesigning existing DSLs.

A. Abstractions

Blackwell describes, “even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centered design” [10]. Thus, an abstraction can cause usability problems when it is not compatible with user’s view. Blandford built an approach called Concept-based Analysis of Surface and Structural Misfits (CASSM), “the purpose of which is in the identification of misfits between the way the user thinks and the representation implemented within the system” [11]. It is of significant importance to design a DSL with better mapping to a musician’s knowledge. Thus, CASSM seems valuable for this purpose, though it hasn’t been applied to programming language design yet. Lee describes that significant design problems can occur if some access to the essential details like time or note velocity is abstracted away in the abstraction layers of computing [12]. This concept is also valuable in DSL design, since DSLs are usually designed on the top of some software frameworks with such abstraction layers.

B. Conceptual Misfits

A concept such as structural misfit [11] is beneficial to assess the gaps between the music domain knowledge and DSL design. Thus, it can play a significant role in the analysis of the usability issues. It is very difficult to gather the requirements of all users and to do an exhaustive task-analysis for programming activity. The way programmers write programs is highly opportunistic. Thus, it may not be appropriate to merely apply some task-specific inspection approaches.

C. Cognitive Dimensions of Notations

Green et.al. developed the Cognitive Dimensions of Notations (CDs) [13] framework, which states design principles for notations, user interfaces and programming languages. It is very useful for broad analysis of the usability problems in programming languages. These CDs were mixed together with Nielsen's heuristics for evaluation of features of a new language [14]. Green's 14 cognitive dimensions provide lexis for discussing many factors in notation or DSL design. Abstraction gradient, closeness of mapping, consistency, diffuseness, error-proneness, progressive evaluation, and role- expressiveness are few of the valuable dimensions to be considered while designing DSL notations for music.

D. End-User Programming

Creative developments in programming language design arise when trying to meet the requirements of a completely different class of user, one whose motives for doing programming are unlike the day-to-day work of computer scientists and mathematicians. An important characteristic of end-user programming research is that end-user programmers should not be regarded as "deficient" computer programmers, but recognized as experts in their domain of work. The attempt to understand "natural" programming concepts by studying school children before they have come across any other language, are of great interest to scientists [15], and are indirectly related to end-user programming. We may find creative solutions by studying new and unusual population of musicians and programmers. While designing programming language, many challenges arise when dealing with unusual kind of users. Smalltalk language was developed to satisfy the needs of children. The spreadsheet ToonTalk was developed to satisfy the needs of business school students and the unusual programming language MediaCubes was invented to solve the problem of configuring networked home appliances.

E. Progressive Evaluation in Interpreted Languages

The work of music artist is unlike programming, and composer's working process is unlike the professional programmer. End-users of the language can be better understood by analogy to musicians than by analogy to professional programmers. Musicians like to constantly listen to the results of their decisions. Most of the composers have a musical instrument close at hand in order to experiment with scores and try out ideas. Music composers continuously adjust their playing according to the sound they hear. The ability of a programming language to support this type of feedback loop is described as the cognitive dimension of *progressive evaluation* [13]. It is often exercised in the design of interpreted languages where the effect of any command needs to be tested immediately. Such interpreted language is beneficial for end-users, as it has been designed with a special view for them to use.

IV. CONCLUSION

DSL for music domain is a distinguishing variation of end-user programming. Thus, it is valuable to understand programming from a perspective that is very different from professional programming. Along with syntax and semantics, many other cognitive dimensions play vital role in designing a good DSL. These cognitive dimensions can help overcome limitations of existing DSLs and provide building blocks for new language design. Abstractions and progressive evaluation should be considered when designing a new DSL for music domain for its widespread acceptance and usage.

V. REFERENCES

[1] E. Loubet, "Laptop performers, compact disc designers, and no-beat techno artists in japan: Music from nowhere," *Computer Music J.*, vol. 24, no. 4, pp. 19-32, 2000. doi: 10.1162/014892600559498

[2] T. Kosar, S. Bohra, and M. Mernik, "A systematic mapping study on domain-specific languages," in *Information and Software Technology*, vol. 71, pp. 77-91, 2012. doi: 10.1016/j.infsof.2015.11.001

[3] T. Walter, F. Parreiras, and S. Staab, "An ontology-based framework for domain-specific modeling," *Software & Systems Modeling*, vol. 13, pp. 83 - 108, 2014. doi: 10.1007/s10270-012-0249-9

[4] I. Ivanov, J. Bézivin, and M. Aksit "Technological spaces: An initial appraisal," in *International Federated Conf.(DOA, ODBASE, CoopIS)*, Los Angeles, 2002. [Online] Available: https://research.utwente.nl/en/publications/technological-spaces-an-initial-appraisal

[5] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61-68, 2002. [Online] Available: http://muse.jhu.edu.libaccess.sjlibrary.org/article/37582/pdf

[6] G. Wang and P. Cook, "ChucK: A concurrent, on-the-fly, audio programming language," in *ICMC*, Singapore, 2003. [Online] Available: http://www.cs.princeton.edu/sound/publications/chuck_icmc2003.pdf

[7] M. Fowler, Domain-specific languages, Addison-Wesley, 2011. [Online] Available: https://martinfowler.com/books/dsl.html

[8] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316--344, 2005. doi: 10.1145/1118890.1118892

[9] M. Voelter, Generic tools, specific languages, Citeseer, 2014. doi: 10.1.1.701.5915

[10] A. Blackwell, L. Church, and T. Green, "The abstract is' an enemy': Alternative perspectives to computational thinking," in *Proceedings PPIG*, vol. 8, pp. 34-43, 2008. [Online] Available: http://www.ppig.org/sites/default/files/2008-PPIG-20th-blackwell.pdf

[11] A. Blandford, T. Green, D. Furniss, and S. Makri, "Evaluating system utility and conceptual fit using CASSM," *International Journal of Human-Computer Studies*, vol. 66, no. 6, pp. 393-409, 2008. doi: 10.1016/j.ijhcs.2007.11.005

[12] E. Lee, "Computing needs time," *Communications of the ACM*, vol. 52, no. 5, pp. 70-79, 2009. doi: 10.1145/1506409.1506426

[13] T. Green, A. Blandford, L. Church, C. Roast, and S. Clarke, "Cognitive dimensions: Achievements, new directions, and open questions," *Journal of Visual Languages & Computing*, vol. 17, no. 4, pp. 328-365, 2006. doi: 10.1016/j.jvlc.2006.04.004

[14] C. Sadowski and S. Kurniawan, "Heuristic evaluation of programming language features: two parallel programming case studies," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 9-14, 2011. doi: 10.1145/2089155.2089160

[15] J. Pane, C. Ratanamahatana, and B. Myers, "Studying the language and structure in non-programmer's solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237-164, 2001. doi: 10.1006/ijhc.2000.0410