A Perspective on the Evolution of Live Programming

Steven L. Tanimoto
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195, U.S.A.
tanimoto@cs.washington.edu

***Abstract*—Liveness in programming environments generally refers to the ability to modify a running program. Liveness is one form of a more general class of behaviors by a programming environment that provide information to programmers about what they are constructing. This paper gives a brief historical perspective on liveness and proposes an extension of a hierarchy given in 1990, to now account for even more powerful execution-oriented tools for programmers. In addition, while liveness concerns the timeliness of execution feedback, considering a broader array of forms of feedback is helpful both in better understanding liveness and in designing ever more powerful development tools.**

***Index Terms*—Liveness, live programming, live coding, debugging, software development tools, liveness levels, integrated development environment, software engineering, code completion, program inference, tactical prediction, strategic prediction.**

I. Introduction

There are two ways a do-it-yourselfer might replace an old lightswitch with a dimmer: (1) first turn off the circuit breaker, or (2) wire it hot. Hot wiring has two advantages: it's probably faster, and in some cases it may be easier to tell which wire is which by touching them to a light bulb or voltmeter. However, hot wiring is dangerous.

In programming, working live need not be dangerous, and the opportunity it offers for immediate feedback can be very valuable. Here are some of the motivations for liveness in programming:

- minimizing the latency between a programming action and seeing its effect on program execution.
- allowing performances in which programmer actions control the dynamics of the audience experience in real time.
- simplifying the "credit assignment problem" faced by a programmer when some programming actions induce a new runtime behavior (such as a bug).
- supporting learning (hence the early connections between liveness with visual programming and program visualization).

The aim of this paper is primarily to put current notions of liveness into a context that includes past notions and possible future ones. Secondarily, the paper suggests a more general category of feedback mechanisms in programming environments to which liveness belongs.

II. History

This section describes early notions of liveness, in order to put newer notions into perspective. It covers developments up through about 2003.

A. Fundamental Notion of Liveness

The traditional program development cycle involved the four separate phases: edit, compile, link, run[1]. Debugging sometimes altered the cycle by changing the run mode to include setting breakpoints, single-stepping, etc. Changing a program while it was being executed was rare outside of debugging sessions, and the changes made during debugging were more often to data values than machine code. Changes to the code were difficult to make, and when they were made, it was generally while execution was suspended at a breakpoint. Live programming was very much an exception to the norm.

In live programming, there is only one phase, at least in principle. The phase involves the program constantly running, even as various editing events occur. A system that supports live programming need not require that all programming performed within the system be live. At times, live programming is unnecessary and the execution of the program might be distracting, particularly when the program is in an intermediate state between useful versions with meaningful behavior.

B. Liveness and Visual Languages

Liveness as an attribute of a programming environment seems to have first been studied in the context of visual languages [1][2]. Visual languages and program visualization attempt to solve a similar problem to the one that liveness addresses, which is the problem of making programming easier by making it easier to understand quickly what a program is doing or supposed to do. Visual representations of programs appeal to the human ability to perceive spatial structure which helps to understand something like a program. A visualization

[1] While the edit-compile-link-run cycle was predominant in the 1960s, one could also consider interactive development with Read-Eval-Print loops, as in Lisp environments, as another tradition --- one a bit closer to live programming because of the interactivity of the language interpreter.

of a program's execution helps translate a process, otherwise hidden inside the machine, into a display (often dynamic, with animation), so that a programmer or user can more easily construct a mental model of the program that is consistent with its behavior.

Writing in the context of visual languages in a relatively early paper [1], I distinguished four separate levels of liveness, culminating in level 4 ("fully live"). A system supporting fully live programming is one that permits a programmer to edit a program while it is running, and furthermore the system continues the execution immediately and without noticeable interruption according to the updated version of the program. The four liveness levels are shown as the first four levels on a new, extended hierarchy in Fig. 1, which I discuss later on.

C. A Few Example Systems

Some early interactive computer systems had live qualities. For example, Sutherland's Sketchpad [3] allowed a user to specify graphical objects interactively, and the appearances and properties of objects were computed and displayed in real time. Although we don't consider Sketchpad to be a programming system, it was not a big leap from interactive drawing programs to tools that supported the drawing and running of "executable flowcharts." Executable flowcharts and executable dataflow diagrams [5] exemplify level-2 liveness, in which a visual representation of the program is "significant" enough to the computer to be run.

The VIVA proposal [1] specified a system for fully live executable dataflow diagrams; its implementation by Birchman [4] in 1991 achieved level 3 liveness (edit-driven updates). A detailed study of the implementation requirements for achieving level-4 liveness was carried out for declarative visual languages by the leading academic research group on spreadsheet languages [2]. The 1995 release of the computer game "Widget Workshop" [6] marked an important point in the availability of live programming-like systems to the public. The Gamut programming-by-demonstration system did away with the run/build distinction, making it fully live [7].

The author's "Data Factory" is a more recent example of a fully live visual programming system for experimental use in education [8]. The primary purpose of the Data Factory was to offer a rich form of computation for which a student could develop a mental model almost immediately. Its key qualities were complete visibility, continuity of data movement during execution, natural parallelism, liveness and a factory metaphor.

D. Criticisms of Liveness

One might argue that liveness at level 4 is probably not very important in most kinds of programming. First of all, it doesn't seem to apply to programs that simply run for a few milliseconds and then terminate. And even if a program runs for a long time, editing a part of the program where execution has passed and to which it will not return will not allow the liveness to do any good. Another criticism of liveness is that it requires too much of the computational resources of a system, since editing and execution have to happen simultaneously, and the edits might actually trigger compile or compile-link-load activities that have to happen so quickly that they are not noticed. Computational resources certainly were limiting factors during the days when our concept of an IDE was developed. Another issue is that there may be semantic inconsistencies between an execution and the latest version of the program.

However, today, there are a variety of ways to address these criticisms of liveness.

III. The Present

I'll draw the line between past and present at about 2003, at the risk of seeming ancient, simply as an organizational convenience. This will allow me to discuss live coding for performances as a recent phenomenon, although it does go back at least a decade; the year 2003 saw the publication of Collins et al in this area [9]. That year was also the 40th anniversary of the publication by Sutherland of Sketchpad [3].

A. Addressing the Criticisms

Let's start this part of the discussion with answers to the criticisms of liveness stated in the previous section. First, let's acknowledge that liveness by itself is not a panacea for programming environments. Neither is it necessary for all programming tasks. But let me now argue that it is potentially very important for many kinds of programming. As with many new features, a programmer probably won't miss it until s/he has used it and it is taken away. Interactive debuggers are a case in point. The ability to easily inspect a computation and modify it is taken for granted in most IDEs. Adding liveness is a straightforward enhancement.

The criticism that liveness is meaningless when running short programs can be addressed by adjusting the run-test configuration as follows. "Auto-repeat" mode simply runs the program over and over again, until it is explicitly stopped by the user. Live edits then lead to nearly instantaneous changes in the perceived execution (assuming the relevant branches are taken, etc.). A similar approach addresses the criticism that live updates might be useless due to execution having passed the location of the updates; like breakpoints, special "start section" and "end section" locations can be set, which define an interval of code that should be indefinitely repeated while live editing is performed on it.

While scarcity of computational resources might have been a disincentive to live programming in the 1980s or 1990s, modern computers can generally handle concurrent editing and running of a program with no difficulty, even if compiling and linking or loading have to be performed after each edit event.

The possibility of semantic inconsistency between an execution and the latest version of a program may problematical in some settings, such as the operation of critical infrastructure. In such settings, one should either not use live programming to begin with, or special precautions should be taken to prevent some kinds of inconsistency and automatically detect and correct others. In other settings, the inconsistent states may not matter. For example, in the Scratch system [10], which permits live editing of running programs, deleting a

game avatar (e.g., Pac Man) and undoing the deletion in the middle of the game can render the reincarnated character inoperable for the remainder of the game --- not a valid game state, but a possibly instructive result to a budding programmer, nonetheless.

B. Liveness in IDEs

In modern integrated development environments such as Eclipse for Java programming, there are many features that work as the programmer codes, to provide feedback to the programmer. These include syntax highlighting, code completion suggestions, and indications of problems associated with various locations in a source file. Facilities for editing running code also exist. The Java virtual machine from version 1.4 has included a "hot-swap" feature that enables the replacement of a class file by a new one while the overall program is running. That permits IDEs to offer a code "push" feature to quickly compile a new version of a class and/or object and insert it into the running JVM. When programmers code live, their behavior may be different than without liveness, and this can lead to new means to infer structures such as unit tests [11].

C. Live Coding in Performance Arts

An important context for the study of live programming is as part of a performance. Although computer music and visual shows such as light shows are often scripted in advance (like classical music programs), some artists emphasize the extemporaneous, improvisational aspects of performance. When musical or visual output is driven by software, changes to the software, whether parameter changes through GUIs or code changes through editors, can be used to cause particular effects in real-time. Although the goals of such programming are very different from that of constructing a usable piece of software, the concerns for human control and feedback are at least parallel, if not clearly similar Collins and Blackwell have analyzed many of the human factors involved in live coding for performance art, drawing on the psychological framework of cognitive dimensions [9][12].

When one contemplates the possibilities of a modern computer in comparison with those of a traditional instrument such as a bassoon or trumpet, the challenge to exploit the power of computers for live music creation is intriguing. In ensemble playing (e.g., in the genre of bebop jazz), the force of time (the steady beat) may add a sense of urgency to any live coding that may be involved. On the other hand, certain program structures can relieve some of the urgency; having a main loop that is repeated once per jazz beat or per measure or per 4-bar unit, etc., may require only that the live coder change a value that controls a case/switch branch. In other words, the live coding could simply be the manipulation of two or three key variables that control the execution.

A different approach to the production of live music would be to incorporate "software sensors" within an IDE, such as Eclipse, in such as way that normal programming actions create "coding music" as a byproduct of the software-development process. A wind chime's music reflects the weather, and an IDE's music reflects the dynamics of coding.

Live programming is taking place in more and more contexts, including web-server scripting, learning environments, and professional tools. This trend is likely to continue.

IV. The Future

While live programming is likely to become ubiquitous, with its increasing incorporation into IDEs, scripting environments, and tools for learning, there are some qualitatively different possibilities, with much of the character of liveness, on the horizon for programming environments. In order to try to offer a broad perspective on liveness, I want to incorporate one of these notions, in two degrees of intensity, into the hierarchy originally offered in 1990 [1].

There is a natural rationale for this new aspect of liveness as an extension of level-4 liveness, and it is about the temporal relationships between programmer actions and computer responses. In liveness level 2, the programmer would do

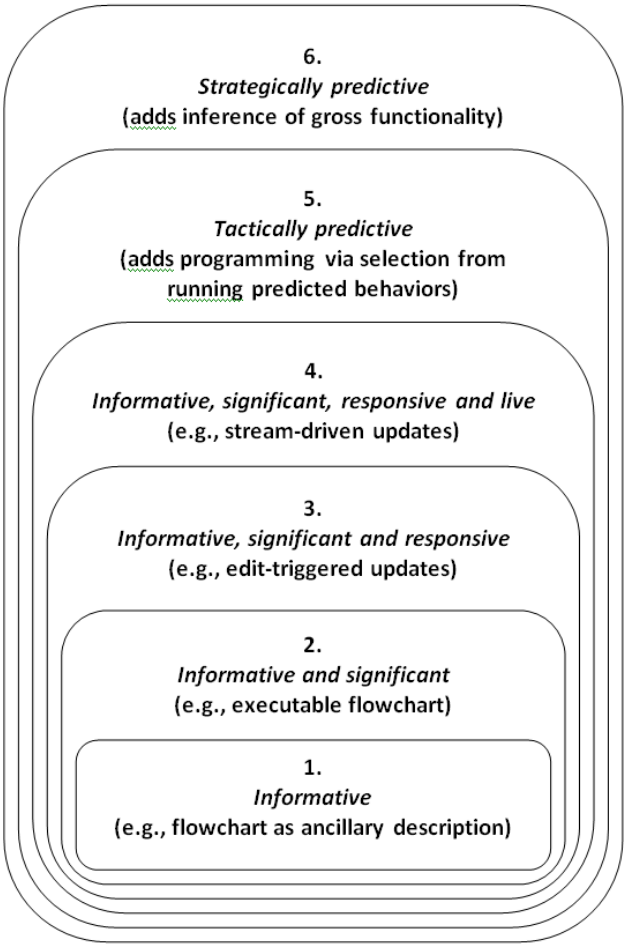

Fig. 1. Proposed extended version of the liveness hierarchy. The tactically predictive and strategically predictive levels incorporate "extrapolation tools" that could offer a new look and feel to the process of programming.

something, would ask for a response, and some time later, the computer would respond. In level 3, the computer would wait and sometime after the programmer did something, would respond. In level 4, the computer wouldn't wait but would keep running the program, modifying the behavior as specified by the programmer as soon as changes were made.

In the new liveness level 5, the computer not only runs the program and responds, but also predicts the next programmer action (with possibly multiple alternatives predicted together) and runs one or more of the predicted resulting versions of the program (probably in separate virtual machines or separate sandboxes). Instead of the environment lagging behind, or just keeping up with the programmer, it stays a step *ahead* of the programmer. Such prediction appears to be feasible through the use of machine learning technology. Programmer actions can be described at multiple levels: lexical, phrase, code block, semantic, etc. Statistical models of programmer behavior can provide one basis for prediction [13], and logical reasoning about meaningful choices can help refine predictions based on the numbers.

Liveness at level 5 can be called "tactically predictive" liveness, because the programming environment plans ahead slightly to discover possible nearby program versions that the programmer might be interested in choosing, either as the next version in the program's evolution, or as an approximation to it that s/he will hand-edit. As in level-4 liveness, the new versions are presented running, and the programmer can select, inspect, or kill any or all of them.

As an example of tactical prediction, the pattern suggested by Hindle et al, of input/output calls of the form (OPEN, ACCESS*, CLOSE) can be used by an IDE to predict, after an OPEN call, that an ACCESS call will be made, possibly within a loop or called function [13]. Thus the tactical prediction might offer a program variant that includes a working call to ACCESS the file (and thus do a default read or write on it) and perhaps a CLOSE call, as well.

The incorporation of the intelligence required to make such predictions into the system is an incorporation of one kind of agency – the ability to act autonomously. Agency is commonly associated with life and liveness. (One might argue that here, liveness has spread from the coding process to the tool itself.)

Further in the future is the possibility of more intelligent inference of the programmer's intentions or desires. Rather than simply making tactical predictions, a system might be capable of successfully making strategic predictions. Such a prediction would cover the desired behavior of a larger unit of software. The system would then (quickly!?) synthesize a program with that behavior from a combination of the current program and a large knowledge base. This is, naturally, liveness level 6.

As an example, let us consider what it would mean to infer, at the strategic level, a specification in the input/output scenario that we just considered at the tactical level. Assuming that the IDE has access to a rich knowledge base, it might determine that the programmer, with some nontrivial probability, wants his or her program to read a text file, parse its contents, and use that data to assign values to a set of configuration variables. Some details would also be inferred, but others simply guessed.

ACKNOWLEDGMENT

I thank Alan Blackwell and Andy Ko for encouraging this submission and Adrian Kuhn for relevant discussion. Thanks also go to the four anonymous referees for their constructive comments, which in turn have improved the paper.

REFERENCES

[1] Tanimoto, S. "VIVA: A visual language for image processing," Journal of Visual Languages and Computing, Vol. 1, No. 2, pp.127-139, June 1990.

[2] Burnett, M. M., Atwood, J. W., Jr., and Zachary T. Welch, Z. T.. "Implementing level 4 liveness in declarative visual programming languages," Proc. of the Int'l. Symp. On Visual Languages VL'98, Sept. 1-4, 1998, Halifax, Nova Scotia, Canada. pp.126-133.

[3] Sutherland, I. E., "Sketchpad: A Man-Machine Graphical Communication System," New York: Garland Publishers, 1980.

[4] Birchman, J. J., "Visualization of Vision Algorithms (VIVA): An Implementation of a Visual Language on the NeXT Computer," Master's thesis, Dept. of Electrical Engineering, Univ. of Washington, 1991.

[5] Docker, T. W. G, and Tate, G. "Executable data flow diagrams," in D. Barnes and P. Brown (eds), Proceedings of the BCS/IEE Conference 'Software Engineering 86', Peter Peregrinus Ltd, 1986, pp. 352–370.

[6] Elliott Portwood Productions, Inc. "Widget Workshop: The Mad Scientist's Laboratory." Computer game published by Maxis, Inc. 1995.

[7] McDaniel, R. G., and Myers, B. A. "Building applications using only demonstration," Proc. IUI'98: 1998 Intr'l. Conf. on Intelligent User Interfaces, San Francisco, CA. pp.109-116.

[8] Tanimoto, S. L. "Programming in a Data Factory. Proc. Int'l Symposium on Visual Languages and Human-Centric Computing, VL/HCC'2003, Auckland, New Zealand.

[9] Collins, N., McLean, A., Rohrhuber, J., and Ward, A. "Live coding techniques for laptop performance," Organised Sound, Vol. 8, No. 3, 2003, pp.321–30.

[10] Maloney, J., Resnick, M. and Rusk, N., "Scratch: A sneak preview," Second Int'l Conference on Creating, Connecting, and Collaborating through Computing, Tokyo, Japan, 2004.

[11] Kuhn, A. "Liveness in Programming, on Extracting Unit Tests from Live Programming Sessions," Unpublished oral presentation at the Univ. of Washington. Dec. 12, 2012.

[12] Blackwell, A. and Collins, N. (2005). "The programming language as a musical instrument," in Proceedings of PPIG 2005, pp.120-130.

[13] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. "On the naturalness of software," Proceedings of the 34th International. Conference on Software Engineering, Zurich, pp.837-847.