

Live Programming

[UP](#) | [HOME](#)

Live programming is the act of programming systems while they are already executing code. While the idea of live programming dates back to LISP and the REPL, it has not been adopted by the software engineering community at large. These are my notes on live programming, but it will slowly turn into an essay.

TODO Status

I'm still trying to find a way to elegantly put together my readings on live programming, so this is a work-in-progress.

Table of Contents


- [TODO Status](#)
- [Literature](#)
 - [Tanimoto \[1\]](#), and [\[2\]](#)
 - [TODO Hancock \[6\]](#)
 - [TODO Collins et al. \[7\]](#)
- [What is Live Programming?](#)
 - [Liveness by example](#)
- [Implementations](#)
- [Themes of Live Programming](#)
 - [Music](#)
 - [Education & Visual Programming Languages](#)
 - [Robotics](#)
 - [TODO Time](#)
- [Bibliography](#)
- [Attach](#)
 - [Liveness levels ATTACH](#)

Literature

This is not an extensive overview of live programming literature, but a sort of guided tour of ideas through papers.

Tanimoto [\[1\]](#), and [\[2\]](#)

In *VIVA: A Visual Language for Image Processing* [\[1\]](#), Tanimoto introduces a classification of “liveness” for visual languages. It has been often used as a measure for the liveness of programming environments, but adjusted accordingly for non-visual languages and other domains like music [\[3\]](#).

 [liveness.png \[1\]](#)

Level 1 liveness

originally referred to having some kind of visual document that expressed, but did not define, the behaviour of the system under discussion. e.g., notes/diagrams in a notebook, a printed out flowchart expressing business logic.

Level 2 liveness

refers to a representation of a system that can be executed. I think originally the idea was dataflow programs, but people use this to talk about normal source code also. The liveness is in the edit/compile/run cycle where the programmer has to make changes, then trigger the system and wait.

Level 3 liveness

is the property of a system that reacts quickly to edits, but still needs to be triggered.

Level 4 liveness

is level 3 liveness, but “online”. That is, instead of waiting for an edit to be signalled, the system will react at the earliest opportunity to source changes.

In *A Perspective on the Evolution of Live Programming* [2], Tanimoto presents a little overview of live programming to date and extends liveness with two more levels. I haven’t seen these levels referenced much, but they describe a system that is more active and may anticipate future changes to the system. For example, the system might see you are creating a while loop and suggest boilerplate for you.

Evidently people find this idea valuable because it is used all the time in live programming literature, but it has received critique and extensions to keep it up to date. Notably liveness diagrams, which are feedback loops in the style of systems dynamics, in [3] and [4], and numerous clarifications of Tanimoto’s intent, as in [5].

The actual content of the original paper, VIVA, receives little attention, but fits in with some common themes of live programming: education, and dataflow languages.

TODO Hancock [6]

Real-Time Programming and the Big Ideas of Computational Literacy is Hancock’s PhD thesis on developing two software environments for teaching programming to children. It is commonly cited for Hancock’s definition and discussion of the “steady frame”. In reference to a comparison between hitting a target with a bow and arrow in archery, and hitting the same target with a water hose, Hancock defines the *steady frame* as:

A way of organizing and representing a system or activity, such that

1. relevant variables can be seen and/or manipulated at specific locations within the scene (the framing part), and
2. these variables are defined and presented so as to be constantly present and constantly meaningful (the steady part).

Where in the example the relevant variable is aim, so the water hose provides a steady frame where the bow and arrow do not.

The steady frame is then one of the common ideas used to explain the value of continuous feedback that live programming delivers, and to make distinctions between continuous feedback that is useful and feedback that is not.

Some of the things I found interesting in this thesis:

The parallel universe problem

is the problem of having great tools that only exist in their own world that can’t really interact with the outside world. To take advantage of the tools you have to “move in” to the parallel universe. Smalltalk is often given as an example of a computing environment with liveness built-in, and due to its all-encompassing nature I think it would qualify for the parallel universe problem. Many many many live programming

tools exist only as prototypes or embedded in some inaccessible world, so this is a real problem.

Variables

are discussed in the context of real-time and live programming. Hancock highlights particularly how children selected and used variables in their programs. Particularly, appropriate variables seemed to serve the steady frame of the program, but also improve the abstractions used in the programs to make them concise and easier to understand.

Steps vs processes

was one of the big ideas in the thesis. Hancock found that in Flogo I (the visual dataflow language), users were having trouble expressing mixed stateful/continuous systems. e.g., a robot driving to a point is continuous behaviour, and when it gets there it may need to swap (state) to a different continuous behaviour. Hancock then embedded these ideas into Flogo II (the text language), but found it hard to express the ideas in simple English that did not confuse children.

Live programming as modelling

TODO

Sean McDirmid on Hacker News:

Flogo 2 and Hancock's dissertation are my goto for live programming origin work. – <https://news.ycombinator.com/item?id=16100842>

and again here <https://news.ycombinator.com/item?id=7761705>.

TODO Collins et al. [7]

Live coding in laptop performance is equal parts manifesto and technical discussion. The authors don't like pre-configured graphical interfaces and prefer the excitement of developing their own systems. What for? For *live coding*, an artistic performance, in this case it is music, performed by devising algorithms on the fly. It is simply trying to push the boundaries and to truly use the programming language as an instrument.

Live coding is often a source for bleeding edge ideas in live programming. Sorensen and Gardner [4] highlight the real-time aspect of live audio production. At a sample rate of 44.1KHz, and with a fixed buffer size, there is only so much time to perform all the computation required for a large signal chain with many instruments and DSP operations. So live coders have taken to developing systems that meet these real-time requirements and provide live programming capabilities. That makes for interesting research.

This attitude towards live programming is different than what you see from people like McDirmid, Edwards, and Victor. There is less focus on education (excepting Sonic Pi), and on ideas that don't make sense in the domain of music. There is some work that looks at features like reified computation, time travel, and rotoscoping (TODO insert citations), which don't make sense in music and other real-time fields. That's not to say live coders aren't interested in making user friendly systems, but live algorithmic music is quite a specific goal.

The Extempore programming language [8] was born out of live coding and is trying to push into new domains. (TODO this)

TODO:

- Show us your screens
- [\[9\]](#)
- [\[10\]](#)

What is Live Programming?

I use the terms “live coding” and “live programming” almost interchangeably, because to me the interesting aspect is the “liveness”. But I understand the desire to disambiguate, especially as someone who used to get upset when my Mum would call just about anything a “Game Boy” when it was really a Game Boy Color or a Game Boy Advance. So this is to be clear:

Live coding normally refers to a type of performance where an artist performs by programming music and or visuals “live”, or “on-the-fly”, or “just-in-time”. The hallmark of live coding is the projection of the programmer’s screen for the audience to see. You can get a sense of this [in this video](#) of Andrew Sorensen, which includes live commentary.

Live programming is the more general term which is the act of programming systems while they are executing code. There is a little nuance to this idea though.

Liveness by example

Imagine you are writing a program that draws a red circle on the screen:

```
def draw_circle(x, y):
    setFill('red')
    diameter = 100
    ellipse(x, y, diameter, diameter)

def main():
    while True:
        draw_circle(50, 100)
        sleep(0.0167) # roughly 60FPS
```

You decide you no longer want a red circle because blue is a better colour. The typical edit cycle would be to close the running program, edit the source code in a text editor, and run it again to see the result. We’re going to say that is **not** live programming.

Imagine we repurpose the system so that you can now send it new function definitions while it is running, and the next time that function gets called it will run the new code. What you can do now is leave the program running, and then send over your new definition of `draw_circle` which draws a blue circle instead of a red one. The next time we go around the main loop we’ll get the blue circle without needing to restart the program.

[Liveness](#) is basically the degree of interactivity and immediacy offered by a system. We can see that the scenario where you send your new definition over and see it in use immediately is more live than the typical edit/run cycle.

Our original red circle program would be classified as level 2, whereas the updated program would be level 3.

You can probably imagine what our circle drawing program would be like at level 4 liveness. When we edit the source code to say 'blue', we would see the change immediately take effect.

Implementations

- REPLs (lisp, python, ruby, ghci, ...)
- JS hot-reloading
- Video game engines (especially old Naughty Dog software)
https://www.gamasutra.com/view/feature/131394/postmortem_naughty_dogs_jak_and
- Clojure and the reloaded workflow
<http://thinkrelevance.com/blog/2013/06/04/clojure-workflow-reloaded>
- Sonic Pi, Extempore, ChuckK, Gibber, Tidal, PD, MAX/MSP, ...

Themes of Live Programming

Music

Education & Visual Programming Languages

Many people believe that tightening the programming feedback loop makes programming more engaging and easier to understand, so it's common to see research that says "typically in programming you have an edit/compile/run cycle, but live programming lets us remove this and make programming easier".

Campusano & Fabry [5]

"Typically, development of robot behavior entails writing the code, deploying it on a simulator or robot and running it in a test setting. ... This process suffers from a long cognitive distance between the code and the resulting behavior, which slows down development and can make experimentation with different behaviors prohibitively expensive. In contrast, Live Programming tightens the feedback loop, minimizing the cognitive distance."

Resig [11]

"In an environment that is truly responsive you can completely change the model of how a student learns: rather than following the typical write -> compile -> guess it works -> run tests to see if it worked model you can now immediately see the result and intuit how underlying systems inherently work without ever following an explicit explanation. ... When code is so interactive, and the actual process of interacting with code is in-and-of-itself a learning process, it becomes very important to put code front-and-center to the learning experience."

A commonly cited thesis on live programming and visual languages is Hancock's *Real-Time Programming and the Big Ideas of Computational Literacy* [6]. This is an early push to get live programming into educational systems, but I think with better principles than others I have seen. Bateson's comparison of continuous feedback

Robotics

TODO Time

[12]

[13]

Bibliography

Please forgive the formatting for the moment, this is a work in process.

1. . Tanimoto, Steven L., *{VIVA}: {A} visual language for image processing*, Journal of Visual Languages \& Computing, 1(2), pp. 127–139 (1990).
[http://dx.doi.org/10.1016/S1045-926X\(05\)80012-6](http://dx.doi.org/10.1016/S1045-926X(05)80012-6).
2. . Tanimoto, Steven L., *A {Perspective} on the {Evolution} of {Live} {Programming}*, Proceedings of the 1st {International} {Workshop} on {Live} {Programming}(2013).
3. . Church, Luke; Nash, Chris and Blackwell, Alan F., *Liveness in {Notation} {Use}: {From} {Music} to {Programming}*, ().
4. . Sorensen, Andrew and Gardner, Henry, *Systems {Level} {Liveness} with {Extempore}*, Proceedings of the 2017 {ACM} {SIGPLAN} {International} {Symposium} on {New} {Ideas}, {New} {Paradigms}, and {Reflections} on {Programming} and {Software}(2017).
5. . Campusano, Miguel and Fabry, Johan, *Live {Robot} {Programming}: {The} language, its implementation, and robot {API} independence*, Science of Computer Programming, 133, pp. 1–19 (2017). <http://dx.doi.org/10.1016/j.scico.2016.06.002>.
6. . Hancock, Christopher Michael, *Real-time {Programming} and the {Big} {Ideas} of {Computational} {Literacy}*, Massachusetts Institute of Technology(2003).
7. . Collins, Nick; McLEAN, Alex; Rohrerhuber, Julian and Ward, Adrian, *Live coding in laptop performance*, Organised Sound, 8(3), pp. 321–330 (2003).
<http://dx.doi.org/10.1017/S135577180300030X>.
8. . . , *Extempore docs*, <http://dx.doi.org/https://extemporelang.github.io/>.
<http://dx.doi.org/nil>.
9. . Magnusson, Thor, *Herdning {Cats}: {Observing} {Live} {Coding} in the {Wild}*, Computer Music Journal, 38(1), pp. 8–16 (2014).
http://dx.doi.org/10.1162/COMJ_a_00216.
10. . Brown, Andrew R. and Sorensen, Andrew C., *aa-cell in practice : an approach to musical live coding*, Proceedings of the {International} {Computer} {Music} {Conference}(2007).
11. . . , *John {Resig} - {Redefining} the {Introduction} to {Computer} {Science}*,
<http://dx.doi.org/https://johnresig.com/blog/introducing-khan-cs/>.
<http://dx.doi.org/nil>.
12. . McDirmid, Sean and Edwards, Jonathan, *Programming with {Managed} {Time}*, Proceedings of the 2014 {ACM} {International} {Symposium} on {New} {Ideas}, {New} {Paradigms}, and {Reflections} on {Programming} \& {Software}(2014).
13. . Aaron, Samuel; Orchard, Dominic and Blackwell, Alan F., *Temporal {Semantics} for a {Live} {Coding} {Language}*, Proceedings of the 2Nd {ACM} {SIGPLAN} {International} {Workshop} on {Functional} {Art}, {Music}, {Modeling} \& {Design}(2014).

Attach

Liveness levels ATTACH