Live end-user programming

A demo/manifesto

Jonathan Edwards
CDG Labs
jonathanmedwards@gmail.com

Jodie Chen
MIT
jodiec@mit.edu

Alessandro Warth
CDG Labs
alexwarth@gmail.com

Abstract

How will live programming get from our current aspirational demos to use in the real world? Modern professional programming will not change easily: our technology stack is a vast edifice built up over decades to optimize performance and compatibility, not ease of use. It is unlikely we can retrofit live programming into this edifice without substantial redesign and reengineering, which would face immense technical, economic, and cultural challenges. The one way forward we see is to retrace the steps of the original live programming environment: spreadsheets. Spreadsheets help non-programmers solve small-scale problems. If we do likewise, we can offer a fully live and radically simplified programming experience that is actually useful in practice, albeit to non-programmers. Perhaps that could be a launching pad to subsequently address professional programming. As a case in point we demonstrate the Chorus project (previously named Transcript), which focuses on do-it-yourself mobile social apps. By restricting ourselves to small problems and non-professional programmers we can provide a highly integrated programming experience that for the first time incorporates live database programming. We demonstrate our initial progress in order to spark a discussion in the live-programming community about the tradeoffs of researching professional vs. end-user programming.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords end-user programming, live programming, mobile applications

1. Getting real

Live programming is best known for a number of aspirational demos of radically improved programming experiences. As the authors of some of these demos we feel obligated to point out their limitations. Many of them deal only with easy cases like numeric functions and 2D graphics. What they lack is a roadmap leading to a complete live programming environment, and often they do not even mention the formidable obstacles in the way, issues like side-effects, non-determinism, visualizing complex graph-structured data, persistence, concurrency, etc. It is important to offer inspiring visions, but eventually we have to actually make it work. How will we get there?

We face not only technical obstacles but also even more daunting socio-economic forces. Software is a major industry with a vast capital investment in a giant pyramid of technologies built up over decades. These technologies have been optimized primarily for performance and compatibility – not the programming experience. The standard response to the difficulty of programming is not to improve the technology but to hire better programmers. Understandably, programmers are all for this. Programmers' status and earning power are coupled to the years and decades they have invested into mastering the tech pyramid. All of these factors are a recipe for stasis. Any attempt to fundamentally improve programming will of necessity involve reducing performance, breaking compatibility, depreciating investments, and obsoleting expertise. Such change will not happen easily nor voluntarily.

The one way forward we see is to retrace the steps of the original live programming environment: spreadsheets. Spreadsheets provide a simple and consistent conceptual model that can be quickly learned by non-programmers. A surprisingly large range of small-scale problems can be fit into the formula-grid metaphor, making spreadsheets far more popular than all other programming technologies combined. The lesson we take from spreadsheets is that there is great value in enabling non-experts to solve small-scale problems relevant to their needs. HyperCard (Goodman 1987) was another much-loved product with a similar moral.

We propose that live programming could make much progress by emulating these successes. By focusing on small-scale problems we are freed from performance constraints, which are not only a technical problem but also the knee-jerk excuse for dismissing new programming ideas.

By building a standalone environment we are freed from the compatibility constraints that inexorably pull us back into the well-worn ruts. Non-programmers have no investment in the status quo and so are open to experimentation with fresh approaches. Most importantly, by delivering real value to real users we have a much better chance of getting people to actually use our new programming technology. Perhaps from that base we could then attack mainstream programming. This strategy and vision has motivated the Chorus project (previously named Transcript).

2. Chorus: do-it-yourself mobile social apps

Say you have started a book club. You need to coordinate with the members to choose books to read, schedule meetings, decide who is bringing the wine and cheese, etc. You probably use some spreadsheets, some web services like message forums and polls, and make it all work with a flood of email. You watch your life passing by as you robotically copy and paste between email and spreadsheets and all these services. What you wish you had is a custom application that automates all this clerical work and runs on modern computers (i.e., phones). But you aren't willing or able to invest much more effort than that required by a spreadsheet.

Chorus allows non-programmers to build simple social applications mediated by shared cloud documents. Like other cloud document systems, users subscribe to a set of documents, with changes bidirectionally synchronized between their phones and the cloud. But unlike a typical textual document, Chorus documents are statically typed tree structures that contain specialized components implementing common conversational patterns like comment threads, polls, calendars, and todo lists. We call these components *social datatypes*. Notifications, task tracking, and off-line operation are all built in. Chorus could be thought of as HyperCard reimagined for the era of smart-phones and social apps. We have built a working prototype of the end-user experience, demonstrated in this video: (Edwards 2015). Figure 1 shows a screenshot of our browser-based prototype environment.

To bootstrap the system, we started with conventional textual syntax in a text editor, which we are currently replacing with a live non-textual programming environment. It is based on the idea of a meta-document: a document that represents the design (i.e. schema/type) of a family of other documents. Meta-documents allow the system to host its own programming environment and maintain a uniform UI so that, like spreadsheets, there need not be a sharp distinction between users and programmers. As an added benefit, editing a meta-document captures higher-level intentions than textual editing does – for example editing a field name implicitly causes a rename refactoring and database migration. Our goal is that the programming experience should be so simple and immediate that, like a spreadsheet, it isn't considered "real" programming. Nevertheless Chorus is in fact

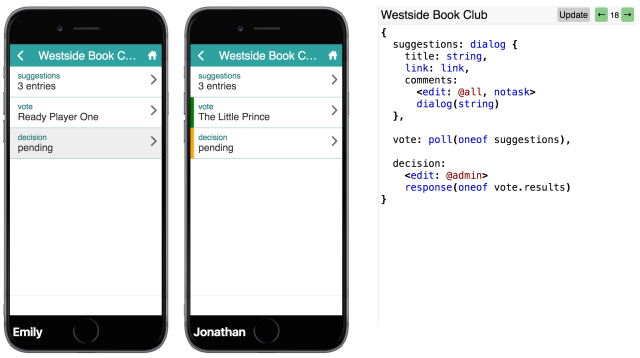

Figure 1. Bootstrap programming environment. Two users are simulated collaborating through the Chorus document defined by the textual syntax on the right.

a Turing-complete programming language, just one that is tailored for simple programs of a certain social flavor.

3. Live database programming

Typical applications are built from at least three distinct technology stacks: a database, a programming language, and a UI framework. Each of these technologies has its own semantics, and much of the complexity of application programming stems from the need to glue them together. Chorus instead provides a single unified model built upon our prior work on Subtext (Edwards 2004-2014). Our statically typed tree structures are effectively databases: the types are schemas, collections serve as tables, and references serve as relationships. All data is persistent, and all execution is performed in concurrent transactions. Program logic is embedded into the document, specifying both imperative event triggers as well as declarative updatable views. The UI is automatically generated from the document's tree structure and the datatypes involved. This integration greatly simplifies application programming, but also gives us the opportunity to extend live programming to live database programming.

We will demonstrate the following key features of live database programming in Chorus (which are still under development at the time of submission):

1. Editing the schema (type) of a document performs a coordinated code refactoring, schema change, and database migration, all done live. We believe this is novel.
2. Transactional concurrent live programming. Edits to code and schema are done in long-running transactions concurrent with ongoing activities by other users who are unaware of the pending refactorings until they are atomically committed. We believe this is novel.
3. Code and schema changes can be applied to the current state of a document or retroactively from the beginning of the document's history (which is fully retained). The latter capability allows us to use documents as testing

scenarios that are continually replayed to visualize and validate execution histories.

4. A key problem of live programming is how to visualize execution histories. History can be visualized linearly as with the timeline of video editors, but it is problematic to visualize state in general programming languages. We will demonstrate how our tree-structured document model provides a simple linear visualization (using the obvious pre-order traversal) that enables us to fully visualize execution history using just the two dimensions available on screen.

4. Commencement

We find it highly unlikely that live programming can be easily retrofitted into our current technology pyramid, which after all was never designed for it. Substantial redesign and reengineering will be necessary, and therefore will face immense technical, economic, and cultural challenges. To make progress, Chorus focuses on the needs of end-users rather than current professional programmers. In this way we hope to build a fully live and radically simplified programming environment that is actually useful in practice. We will demonstrate our initial progress in order to spark a discussion in the live-programming community about the tradeoffs of researching professional vs. end-user programming. We look forward to a vigorous exchange of ideas at the workshop.

References

Jonathan Edwards. Subtext project website. 2004-2014 `http://subtext-lang.org`

Jonathan Edwards. Transcript: End-user programming of mobile social apps. YOW! December 2015 `https://www.youtube.com/watch?v=XBpwysZtkkQ`

Danny Goodman. The complete HyperCard handbook. 1987. `https://archive.org/details/The_Complete_HyperCard_Handbook`