It’s Alive!

Continuous Feedback in UI Programming

Sebastian Burckhardt
Manuel Fahndrich
Peli de Halleux
Sean McDirmid
Michal Moskal
Nikolai Tillmann
Microsoft Research

Jun Kato
The University of Tokyo
Live Programming: Archer Analogy

[Hancock, 2003]

- Archer: aim, shoot, inspect, repeat

- Hose: aim & watch
Live Programming : Archer Analogy
[Hancock, 2003]

• Archer:
  aim, shoot, inspect, repeat
  edit, compile, test, repeat

• Hose:
  aim & watch
  edit & watch
Quick Demo:
What is Live Programming?
What is TouchDevelop?
Question:
How to do live programming?

• Target:
  Event-driven apps with graphical user interfaces (GUI’s)
  • User input events (tap button, edit text, ..)
  • I/O events (e.g. asynchronous web requests)

• We can think of code editing as an event
  (replace old program with a new one)

• What should we do in this situation?
on code changes, just replay execution from beginning

How to do live programming?

Replay Based ?
How to do live programming?

• Inputs?
  • Must record or repeat user inputs and I/O

• Divergence?
  • Recorded events may no longer make sense after code change

• Side effects?
  • Replaying external side effects can have surprising consequences

• Performance?
  • Apps with GUIs can run for a long time, replay not efficient

on code changes, just replay execution from beginning
on code changes, just replay execution from beginning

Replay is difficult. Worse: it does not always make sense.

• Inputs?
  • Must record or repeat user inputs and I/O

• Divergence?
  • Recorded events may no longer make sense after code change

• Side effects?
  • Replaying external side effects can have surprising consequences

• Performance?
  • Apps with GUIs can run for a long time, replay not efficient
Widen the Scope.

Question:
How to do live programming?

Replay Based

?
Widen the Scope.

**Question 1:**
How to program event-driven apps with GUIs?

- Retained Model-View
- Stateless Model-View

**Question 2:**
How to do live programming?

- Replay Based

?
Widen the Scope.

Question 1: How to program event-driven apps with GUIs?
- Retained Model-View
- Stateless Model-View

Question 2: How to do live programming?
- Replay Based
- Model-View Based
Question 1: How to program GUls?

• Model-View-Controller: Well established pattern for interactive applications
• Many variations exist
Question 1: How to program GUIs?

• Model-View-Controller: Well established pattern for interactive applications
• Many variations exist. We eliminate controller and put event handlers into the view.
Question 1: How to program GUIs?

• Model-View-Controller: Well established pattern for interactive applications
• Many variations exist. We eliminate controller and put event handlers into the view.

View
- UI Elements
- Event Handlers

Model
- Data definitions

Key question: How to define and maintain correspondence between view and model?
How to program GUIs?

Program = Model + View-Construction + View-Update

- **Model**: Data definitions that define the model
- **View-Construction**: Code that defines how to construct the view for a given model
- **View-Update**: Code that defines how to update the view in reaction to model changes
How to program GUIs?

- Model: Data definitions that define the model
- View-Construction: Code that defines how to construct the view for a given model
- View-Update: Code that defines how to update the view in reaction to model changes

Program = Model + View-Construction + View-Update

Retained View

Redundant

Error prone
How to program GUIs?

<table>
<thead>
<tr>
<th>Retained View</th>
<th>Stateless View</th>
</tr>
</thead>
</table>

Program = Model + View-Construction + View-Update

Program = Model + View-Construction

- **Model**: Data definitions that define the model
- **View-Construction**: Code that defines how to construct the view for a given model

Update is simple: throw away old view, build new one.
Example.

Very simple app: list of strings.

User can add entries by hitting the “add” button.

Program = 
Model + View-Construction
Example

Model

- **data** entries: String Collection
- **data** field: String

View = Tree, decorated with attributes and event handlers
How to write view construction code?

Many frameworks are hybrids between a general-purpose language and a declarative language (e.g. C# + XML).

We would prefer: stay within single host language, but make code look as declarative as possible.

Host language for our prototype: TouchDevelop

Host language in the paper: lambda-calculus
Idea: extend host language

• Special construct: nested “boxed” statements

```plaintext
boxed {
    .... nested code here....
}
```

• When executing, creates box tree implicitly
• view structure is implied by program structure, no need for programmer to manipulate collections!
• Code looks similar to declarative code.
Code Example.

Model

- **data** entries: String Collection
- **data** field : String

```
entry1 entry2

add [entry3] x
```

View-Construction Code

```
display
  boxed
    box → use horizontal layout
  for each s in entries do
    boxed
      labelstyle()
      s → post
  boxed
    box → use horizontal layout
    boxed
      buttonstyle()
      "add" → post to wall
      on tapped(() => entries → add(field))
    boxed
      inputstyle()
      box → edit(field, (x) => field := x)

function buttonstyle()
  box → set border(colors → foreground, 0.1)
  box → set margins(0.5, 0.5, 0.5, 0.5)
  box → set padding(0.2, 0.2, 0.2, 0.2)
  box → set background(colors → orange)
  box → set width(10)
```
No need for separate language or special collection classes.

- Adapt layout to various conditions – **use a standard conditional**
- Repeated elements – **use standard loops**
- Keep your code organized – **use functional abstraction**
- Provide widget collection – **write a library**

User interface element = just a function.
Question 1: How to do live programming?

• This is now much easier to get a grasp on.
Question 1: How to do live programming?

Answer:

on code changes, migrate model, build fresh view
Does Model Migration Work?

- Currently, we do something very simple
  - Variables whose types have changed are removed from model
- Experience: behaves reasonably in practice w.r.t to typical changes and user expectations
- More interesting solutions conceivable for structured data (cf. schema evolution, dynamic code updating)
Valid Concern: Speed?

• Isn’t it too slow to reconstruct the view from scratch every time?

• In our experience (Browser-based, Javascript):
  • Re-executing the compiled display code is no problem for our apps (never more than 1000 objects on screen)
  • However, recreating the DOM tree from scratch is too slow (browser takes too much time) and has other issues (e.g. lose focus while typing in a textbox when it is replaced)
  • Fix: We implemented optimization that modifies the DOM tree incrementally when reexecuting the display code.
Yes, but what does all this mean, exactly?

• Paper contains a careful formalization of these concepts!
• Lambda calculus + UI primitives (boxes)
• Operational semantics
• System model for event-handling with page stack, UI, and code change events
• Type and Effect System
Expressions:

\[ e ::= v \] (value)

\[ e_1 e_2 \] (application)

\[ f \] (function)

\[ (e_1, \ldots, e_n) \] (tuple), \( n \geq 0 \)

\[ e.n \] (projection), \( n \geq 1 \)

\[ g \] (read global)

\[ g := e \] (write global)

\[ \text{push } p \ e \] (push new page)

\[ \text{pop} \] (pop page)

\[ \text{boxed } e \] (create box)

\[ \text{post } e \] (post content)

\[ \text{box}.a := e \] (set box attribute)
<table>
<thead>
<tr>
<th><strong>Expressions:</strong></th>
<th>( e ::= v )</th>
<th>( e_1 e_2 )</th>
<th>( f )</th>
<th>( (e_1, \ldots, e_n) )</th>
<th>( e.n )</th>
<th>(value)</th>
<th>( (application) )</th>
<th>( (function) )</th>
<th>( (tuple), (n \geq 0) )</th>
<th>( (projection), (n \geq 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Read/Write Model</strong></td>
<td>( g )</td>
<td>( g ::= e )</td>
<td></td>
<td></td>
<td></td>
<td>(read global)</td>
<td>(write global)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Navigation</strong></td>
<td>( \text{push } p \ e )</td>
<td>( \text{pop } e )</td>
<td></td>
<td></td>
<td></td>
<td>(push new page)</td>
<td>(pop page)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>View Construction</strong></td>
<td>( \text{boxed } e )</td>
<td>( \text{post } e )</td>
<td>( \text{box.a } ::= \ e )</td>
<td></td>
<td></td>
<td>(create box)</td>
<td>(post content)</td>
<td>(set box attribute)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System State:
\[ \sigma ::= (C, D, S, P, Q) \]

System Components:
- \[ C ::= \epsilon \mid C \cdot d \] (program code)
- \[ D ::= \bot \mid B \] (display)
- \[ S ::= \epsilon \mid S[g \mapsto v] \] (store)
- \[ P ::= \epsilon \mid P(p, v) \] (page stack)
- \[ Q ::= \epsilon \mid Q \cdot q \] (event queue)

Program Definitions:
- \[ d ::= \text{global } g : \tau = v \] (global)
- \[ \mid \text{fun } f : \tau \text{ is } e \] (function)
- \[ \mid \text{page } p(\tau) \text{ init } e_1 \text{ render } e_2 \] (page)

Box Content:
- \[ B ::= \epsilon \] (empty)
- \[ \mid B \cdot v \] (leaf content)
- \[ \mid B[a = v] \] (box attribute)
- \[ \mid B \langle B \rangle \] (nested box)

Events:
- \[ q ::= [\text{exec } v] \] (execute thunk)
- \[ \mid [\text{push } p \cdot v] \] (push new page)
- \[ \mid [\text{pop}] \] (pop page)
System Execution Steps

Three rules that enqueue events:

\[
(\text{STARTUP}) \quad (C, D, S, \epsilon, \epsilon) \to_g (C, \bot, S, \epsilon, [\text{push start} ()])
\]

\[
(\text{TAP}) \quad [\text{ontap = v}] \in B \\
(C, B, S, P, Q) \to_g (C, \bot, S, P, [\text{exec v}] Q)
\]

\[
(\text{BACK}) \quad (C, D, S, P, Q) \to_g (C, \bot, S, P, [\text{pop}] Q)
\]

Three rules that handle events:

\[
(\text{THUNK}) \quad (C, S, Q, v ()) \to^*_s (C, S', Q', ()) \\
(C, D, S, P, Q [\text{exec v}]) \to_g (C, \bot, S', P, Q')
\]

\[
(\text{PUSH}) \quad C(p) = (f_i, f_r) \quad (C, S, Q, (f_i v)) \to^*_s (C, S', Q', ()) \\
(C, D, S, P, Q [\text{push p v}]) \to_g (C, \bot, S', P (p, v), Q')
\]

\[
(\text{POP}) \quad P = P'(p, v) \quad \text{or} \quad P = P' = \epsilon \\
(C, D, S, P, Q [\text{pop}]) \to_g (C, \bot, S, P', Q)
\]

One rule to refresh the display:

\[
(\text{RENDER}) \quad C(p) = (f_i, f_r) \quad (C, S, \epsilon, (f_r v)) \to^*_r (C, S, B, ()) \\
(C, \bot, S, P (p, v), \epsilon) \to_g (C, B, S, P (p, v), \epsilon)
\]

One rule to change the program code:

\[
(\text{UPDATE}) \quad C' \vdash C' \quad C' : S \triangleright S' \quad C' : P \triangleright P' \\
(C, D, S, P, \epsilon) \to_g (C', \bot, S', P', \epsilon)
\]
Two execution modes with different allowed side effects

Event handler execution
- Can mutate model
- Can push/pop pages

Display code execution
- Can set box attributes
- Can create boxes
System Model Visualization

System State:
\( \sigma ::= (C, D, S, P, Q) \)

System Components:
- \( C ::= \epsilon \mid C d \) (program code)
- \( D ::= \bot \mid B \) (display)
- \( S ::= \epsilon \mid S[g \mapsto v] \) (store)
- \( P ::= \epsilon \mid P(p, v) \) (page stack)
- \( Q ::= \epsilon \mid Q q \) (event queue)

Initial \((C, -, \epsilon, \epsilon, \epsilon)\)
- Push start page

Active \((C, S, P(p, v), Q)\)
- Handle events in queue until queue empty
- Need Render \((C, S, P(p, v), \epsilon)\)
- Build view from model

Ready \((C, B, S, P(p, v), \epsilon)\)
- User taps button or hits back button
- User changes code
Type & Effect System

• Judgments

\[ \Gamma \vdash_\mu e : \tau \]

\[ \mu ::= p \mid r \mid s \]  

pure, render, state effect

• Allows us to tell what kind of function we are looking at

• Lets us ensure that \{event handlers, display code\} only have the allowed side-effects for the given mode
Practical Experience

• Type/Effect system is sometimes too restrictive. For example, does not allow this in display code:

```javascript
var x = new object(); x.field := value;
```

• More useful in practice: runtime checks that allow allocating fresh objects in a display heap, and allow mutation of the display heap
Goals

- Programming Model
  - Support **succinct programming of apps with GUIs** (graphical user interfaces)
  - Support **live editing**
  - Precise reactive semantics (user events, code changes)

- Implementation
  - Embed into TouchDevelop (language, runtime, IDE)
  - Enforce correct use of feature (separation of model and view)

Contributions

- Live-View Approach
- Formal System Model
- Static Type/Effect System
- Language Integration

**Feature is public**
Runs on all devices
[TouchDevelop.com](http://touchdevelop.com)