Live Programming by Example: Using Direct Manipulation for Live Program Synthesis

Christopher Schuster   Cormac Flanagan
University of California, Santa Cruz
{cschuste,cormac}@ucsc.edu

Abstract

To provide a better programming experience, live programming environments allow changes to the code of running programs. These changes are usually made by editing the source code. In this paper, we introduce *live programming by example* which enables updates to the code by direct manipulation of the program's user interface. Besides a formal definition of live programming by example, we also present a concrete prototype implementation for JavaScript that enables the programmer to change string literals in the source code by direct manipulation of the HTML output based on a dynamic string origin analysis. While this prototype only supports light-weight synthesis, future live program synthesis algorithms could support a wider range of program edits.

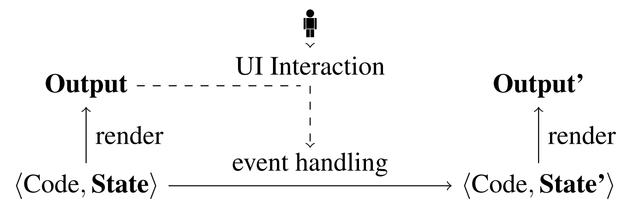
Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Live Programming, Programming by example, Program Synthesis, Direct Manipulation, JavaScript

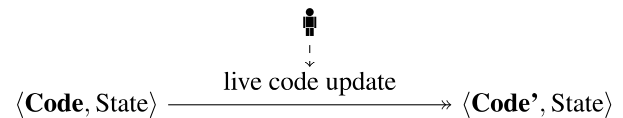
1. Introduction

Most interactive UI applications today are based on an event-based programming model in which user interactions are represented as *events*. The main event loop then dispatches these events to registered event handlers that change both the internal application state as well as the visible user interface. However, imperative updates to a stateful user interface can results in data dependencies that are difficult to keep consistent. Therefore, patterns such as Reactive Programming (Bainomugisha et al. 2013) and Model-View-Controller (Krasner and Pope 1988) have been adopted that separate the *handle* code for updating the application state

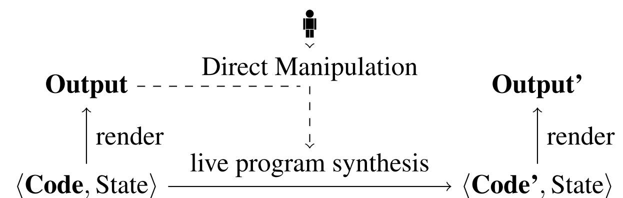
[Copyright notice will appear here once 'preprint' option is removed.]



(a) Regular User Interaction in an Event-based System



(b) Live Programming with Source Code Updates



(c) Live Programming by Direct Manipulation of the User Interface

Figure 1: In an event-based system, the user interacts with the output to change the state of the program (a). Live programming allows code updates to running systems (b). With live programming by example, code updates can be performed by direct manipulation of the user interface (c).

from the *render* code for generating the output. Figure 1a illustrates how this programming model processes user interactions.

While common user interactions only affect the application state and output, live programming environments like Smalltalk (Goldberg and Robson 1983) and TouchDevelop (Burckhardt et al. 2013) also allow changes to the code of a running application in order to provide earlier feedback to the programmer for both debugging and development. Large code updates may require a restart or manual data

migration but smaller code updates can often be applied automatically while retaining the current application state. In contrast to regular user interactions performed through the application's user interface, these live code updates are performed in the source code editor of the development environment (see Figure 1b).

In this paper, we introduce a new kind of user/developer interaction that extends live programming with UI interactions. Instead of supplying a change to the source code, the current user interface can modified through *direct manipulation* (Shneiderman 1983) to change the running code. The changed user interface serves as *example* for the intended output of rendering the current state. Depending on the domain and programming language, there are various different programming-by-example techniques (Lieberman 2001; Gulwani 2011) to 'repair' or 'synthesize' a new program whose rendered output partially or fully conforms to the provided output example. To better describe the general design space of these systems, as illustrated in Figure 1c, we also include a minimal formalization.

Additionally, we present a concrete prototype based on a live programming environment for JavaScript for applications that separate rendering from event handling (Burckhardt et al. 2013). The environment allows string constants in the *render* code to be changed directly in the HTML output based on a dynamic string origin analysis (Wang et al. 2012). This programming environment serves as a precursor for future systems that allow synthesis of more complex live code modifications by direct manipulation.

To summarize, the contributions of this paper are

- a new kind of user interaction that combines live programming of stateful event-based applications with programming by example based on direct manipulation of the user interface,
- a formal definition of such systems, and
- a JavaScript implementation for direct manipulation of string constants in the *render* code of running programs.

2. Related Work

The term *live programming* is used in different contexts and can denote the act of programming as part of a live art or music performance or the feature of programming environment to continuously evaluate expressions and displaying results alongside code. However, this research focusses on live programming as the ability to change the code of a running *stateful* application without restarting it, also known as *hot-swapping* or *dynamic software update* (Hicks et al. 2001), while providing immediate and continuous feedback about these code changes. Such live programming systems were described and motivated by Hancock (Hancock 2003) and further explored by systems like SuperGlue, which uses dynamic inheritance and explicit FRP signals (McDirmid 2007), and Elm, which demonstrates live programming and time traveling with first-order FRP (Czaplicki and Chong 2013).

The solution described in this paper is based on earlier work on live programming for event-based systems (Schuster and Flanagan 2015) and generally follows the live programming technique for TouchDevelop which requires UI rendering and stateful computation to be separated and prohibits function values (closures) in the application state to allow the state and the code to be updated independently (Burckhardt et al. 2013). More recent work outlined the possible design space between live programming systems that resume computation (with a possibly inconsistent state) and systems that record and replay execution (McDirmid 2013), as well as introducing *managed time* as concept for supporting both live programming and time travel (McDirmid and Edwards 2014).

Programming-by-example allows users to author programs by providing examples instead of writing source code. Prior work on programming-by-example and programming-by-demonstration ranges from domain-specific macro systems, visual programming languages and inference of string processing rules (Lieberman 2001; Gulwani 2011) to depth-limited generate-and-test approaches for general-purpose programming languages. Most noteworthy, CodeHint (Galenson et al. 2014) synthesizes short Java code snippets at runtime based on user-provided queries. In order to synthesize larger code snippets, SMT solver-aided approaches may be a promising alternative (Torlak and Bodik 2013).

Direct Manipulation (Shneiderman 1983) of the graphical user interface is a well-known form of user interaction with popular applications in Smalltalk (Goldberg and Robson 1983), Morphic (Maloney and Smith 1995) and others. However, direct manipulation usually only affects the current state of visible objects. Recent work on *Prodirect manipulation* (Chugh et al. 2016) shows how direct manipulation of SVG vector graphics can be used to automatically modify the SVG rendering code. The same idea has also been applied to the manipulation of string constants in PHP web applications (Wang et al. 2012). These two projects are most closely related to this paper but do not support live code updates of stateful applications without restarting the execution.

3. Live Programming for Event-based Languages

Live programming enables changes to the source code of running programs without restarting its execution. The inherent entanglement of state and code poses a challenge for live programming but approaches like functional reactive programming or a separation of rendering and event handling enable programmers to perform many live updates without manual data migration or restarting the execution.

```
 1 <div>
 2   <input id="i" />
 3   <p id="o"></p>
 4 </div>
 5 <script>
 6 var inp = document.getElementById("i");
 7 var out = document.getElementById("o");
 8 var str = "";
 9 inp.onkeyup = function() {
10   str = inp.value;
11   out.innerHTML =
12     str.replace(/keyboard/g,"leopard");
13 }
14 </script>
```

Figure 2: Example JavaScript code with an event handler performing imperative state and DOM updates.

```
var str = "";
function keyup(evt) {
  str = evt.target.value;
}
function render() {
  return (<div>
    <input value={str} onkeyup={keyup} />
    <p>
      {str.replace(/keyboard/g,"leopard")}
    </p>
  </div>);
}
```

Figure 3: JavaScript code corresponding to Figure 2 with separate rendering and event handling. Here, inline HTML tags are used to create a tree representation of the output and attach event handlers.

3.1 Example JavaScript Application

To illustrate the challenges of live programming for general imperative applications, we consider a simple interactive application that replaces keywords in a text according to a fixed rule[1].

Figure 2 shows a 'traditional' way of implementing this application with imperative updates to register and modify event handlers (`inp.onkeyup`), mutate global state (`str`) and update the graphical user interface (`out.innerHTML`).

In the context of live programming, changing `"leopard"` to `"butterfly"` in Figure 2 should ideally also update the visible output without modifying the application state. However, this would involve registering the modified function as new event handler (line 9) and updating the output (line 11 and 12) *without* also mutating the state (line 10). Due to the structure of the code, the only choice is to update the event handler and display an inconsistent output until the next `keyup` event, or to restart the application and force the user to re-enter the string.

[1] There are several browser extensions to replace keywords on webpages, some of which are inspired by https://xkcd.com/1031/.

3.2 Separating Rendering from Event Handling

In the example shown in Figure 2, any change in the application state involves corresponding updates in the output to ensure consistency. Programming models such as Reactive Programming (Bainomugisha et al. 2013) and Model-View-Controller (Kransner and Pope 1988) solve this issue by separating the rendering code from other stateful computation. Figure 3 shows an implementation of the same application but the output will be generated by a *pure* `render` function, which cannot modify the application state, and events will be handled without directly accessing the DOM output. Instead, any state modification automatically triggers a re-rendering to keep the output continuously consistent.

3.3 Live Code Updates

Separating rendering from event handling has the additional advantage that both can be updated independently. Following the example from Section 3.1, changing `"leopard"` to `"butterfly"` in Figure 3 can now automatically re-render the output with the new rendering code. Since `render` is pure, this process does not affect the application state.

In contrast to changes to the rendering code, changes to the event handling will not immediately be visible in the output. Instead, the updated code will be used to handle all subsequent events[2].

In addition to live programming, separating rendering from event handling also has advantages for back-in-time debugging. By either recording events or snapshotting the application state between events, it is possible to travel back in time and visualize past execution states using the pure rendering mechanism. Combining back-in-time debugging with live programming enables the programmer to navigate both execution as well as the version history simultaneously.

3.4 Limitations

Changes to the program code may involve added, modified or removed function definitions. Unfortunately, function values/closures in the application state cannot always be automatically transformed to match the new code, therefore closures are currently not allowed in the application state.

Additionally, an application that resumed execution with an updated event handler may behave different than an application that has been restarted. Depending on the development or debugging context, this may or may not be desirable.

Finally, changes to the initialization code of the program or the data type of the application state will likely result in an incompatibility and require either manual data migration or a restart.

[2] Alternatively, it is possible to restart the execution and replay past events.

4. Live Programming by Example

Live programming systems as described in the previous section enable program updates at runtime for changes made to the program source code. By separating rendering from event handling, the application output can be updated in reaction to code changes to provide visual feedback to the developer. Extending on this idea, the output of the program can also be used by the developer to *make* changes to the program — in particular in cases where direct manipulation is more convenient or intuitive than edits to the source code. This idea of *live programming by example* is also illustrated in Figure 1c.

4.1 Formal Definition

Live programming by example is applicable to a wide range of applications and programming languages. To clarify the semantics and system requirements for the approach outlined in this paper, we model the system configuration and its transitions on a higher level of abstraction.

Figure 4 shows a formal definition of a system that supports live programming by example. The input events q, user interface representation o, values v, expressions e and evaluation semantics $e \downarrow v$ all depend on the concrete implementation and are mostly left unspecified. However, the system has to be event-based in the sense that input from the environment is supplied as sequence of events q. Additionally, the output of the application is not generated imperatively (e.g. via `printf` statements); instead, it is summarized as a single value o (e.g. the DOM of a JavaScript application or the framebuffer of an OpenGL application).

Following the programming model described in Section 3.2, the program code is partitioned into event handling h and rendering r such that h is a function that mutates state s in response to events while r is a pure function that generates output o.

There are three different high-level interactions with the system. Regular input events are processed by evaluating the event handler and rendering the potentially modified state (E-EVENT). Updates to the program code h or r are performed without changing the state such that the output is re-rendered with the new rendering code r' and subsequent events are handled with h' (E-SWAP). Given output o, the developer can initiate live programming by example by changing the output o into a desired output example o' that is then used for synthesis (E-EXAMPLE). The judgement $(r, s, o) \bigtriangledown r'$ infers a modified render function r' from r such that, ideally, the output o'' generated by the synthesized render function r' exactly matches the user example $o' = o''$. However, it may be advantageous to use a heuristic that prioritizes smaller changes to the render function r and tolerates minor differences between new output and the example $o' \approx o''$.

Apart from language details of the implementation and the concrete UI representation of the output, this definition

$$
\begin{aligned}
q &::= [\text{keypress } v] \mid [\text{click } v\ v] \mid \ldots && \text{(Input Events)}\\
o &::= [\text{label } v] \mid [\text{row } o\ o] \mid \ldots && \text{(User Interface)}\\
h, r, s, v &::= \lambda x.\, e \mid \langle v, v \rangle \mid q \mid o \mid \ldots && \text{(Values)}\\
e &::= v \mid e(e) \mid x \mid \langle e, e \rangle \mid \ldots && \text{(Expressions)}\\
i &::= q \mid [\text{swap } h\ r] \mid [\text{example } o] && \text{(Interactions)}
\end{aligned}
$$

$$h \text{ (Handle)} \quad r \text{ (Render)} \quad s \text{ (State)} \quad e \downarrow v \text{ (Evaluation)}$$

$$\langle h, r, s \rangle \qquad \text{(System Configuration)}$$

$$\langle h, r, s \rangle \xRightarrow[o]{i} \langle h, r, s \rangle \qquad \text{(System Transitions)}$$

$$(r, s, o') \bigtriangledown r' \qquad \text{(Synthesize } r' \text{ to match example } o'\text{)}$$

$$\frac{h(s, q) \downarrow s' \qquad r(s') \downarrow o}{\langle h, r, s \rangle \xRightarrow[o]{q} \langle h, r, s' \rangle}\ \text{E-EVENT}$$

$$\frac{r'(s) \downarrow o}{\langle h, r, s \rangle \xRightarrow[o]{[\text{swap } h'\ r']} \langle h', r', s \rangle}\ \text{E-SWAP}$$

$$\frac{(r, s, o') \bigtriangledown r' \qquad r'(s) \downarrow o'' \qquad o' \approx o''}{\langle h, r, s \rangle \xRightarrow[o'']{[\text{example } o']} \langle h, r', s \rangle}\ \text{E-EXAMPLE}$$

Figure 4: Formal definition of a system that supports user input events, live code updates, and live programming by example. The program has to be partitioned into event handling h and rendering r but the evaluation semantics $e \downarrow v$ of the underlying language is left unspecified.

highlights the design space for live programming by example. Both the program synthesis technique ($\bigtriangledown$) as well as the user intent of supplied examples ($\approx$) enable a wide range of different approaches ranging from the simple manipulation of string literals to sophisticated direct manipulation interactions and end-user programming.

5. Editing String Literals in JavaScript Programs by Direct UI Manipulation

Based on prior work on live programming environments for event-based languages (Schuster and Flanagan 2015), we implement a first prototype that supports live programming by example for editing string literals in JavaScript application through UI manipulation. Its source code[3] and a live demo[4] are both publicly available.

[3] Source code at http://github.com/levjj/rde/

[4] Online live demo at http://levjj.github.io/rde/

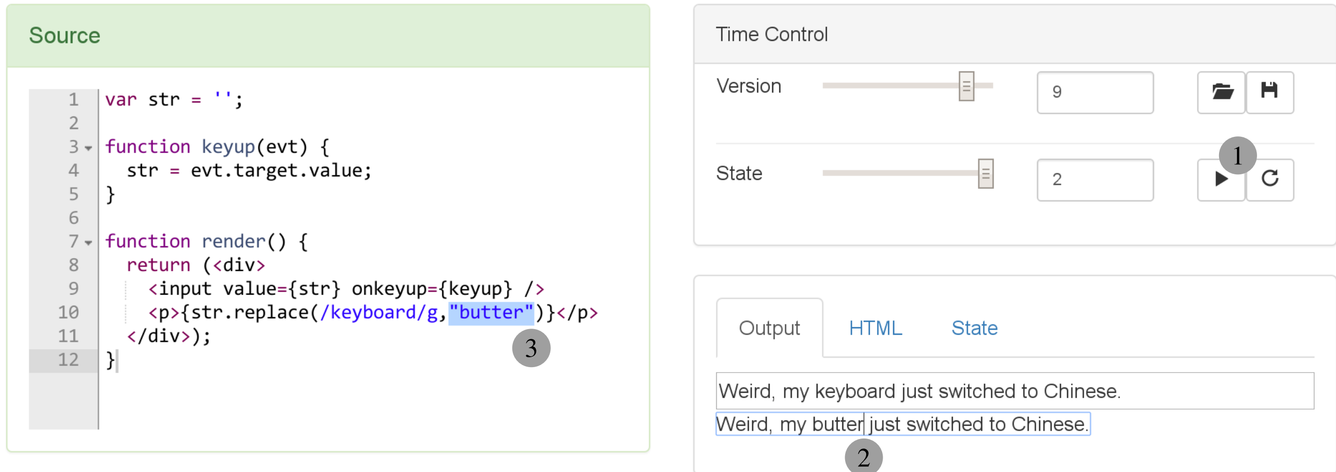


Figure 5: The live programming environment features an editor, a live view of the output as well as controls for traveling to previous code versions/execution states. Stopping the normal execution (1) prevents event processing but enables other forms of UI interaction like changes to text displayed in the output (2). This kind of UI manipulation is used to automatically change corresponding string literals in the source code (3).

5.1 Example Interaction

Given an example application for replacing keywords in a text according to a static replacement rule (see Section 3.1), this rule in the source code can be modified through UI manipulation. Our approach depends on a separation of rendering and event handling (see Section 3.2), so that changing the string constant `"leopard"` to `"butterfly"` immediately updates the output accordingly. This change can be done by directly editing the source code, but it can also be performed by editing the output such that the generating string literal in the source code will be changed to match the intended output example (see Figure 5).

Interactions with the application's user interface are usually handled by the application itself. In order to support live programming by example, the programming environment provides a way to halt execution (1) and enable a special interaction mode for the application's user interface. Alternatively, it may be possible to reserve certain controls for this purpose, e.g. reserving a special *meta key* for direct manipulation or using *halo controls* (Maloney and Smith 1995).

Different forms of direct manipulation may be available (e.g. resizing or reordering via drag and drop). In this example, the text of the static label (`<p>`) becomes editable by the user/programmer (2).

Parts of the label text `"leopard"` originate from a string literal in the program source code. Therefore, the string literal is highlighted in the editor (3) and changes in the user interface will also be applied to the source code. Text parts that do not originate from source code literals cannot be modified. Any change to the source code causes the output to be re-rendered, so if the word "keyboard" would appear more than once in the input text, all of its occurrences would be replaced according to the new `render` function thereby ensuring consistency between code and output.

5.2 String-origin Analysis

In order to support the interaction outlined in the previous section, the environment needs a mapping of strings in the output to their generating string literals in the source code. This mapping can be obtained by instrumenting the execution such that string values are tagged with provenance information. This form of dynamic analysis is closely related to dynamic taint analysis for information flow security.

As a first step, all string literals/constants in the source code are assigned a unique identifier by replacing them with constructor calls, yielding tagged string values with origin information.

```
    var x = "ab";
--> var x = stringlit("ab", 23);
```

Additionally, built-in unary and binary operations which cannot be instrumented are replaced with calls to custom functions implementing these operators.

```
    var y = x + "c";
--> var y = addop(x, stringlit("c", 42));
```

Tagged string objects include a custom implementation of concatenation, substring extraction, replacement and other common string operators (e.g. `str.toLowerCase()`), such that origin information is retained but otherwise behave like regular string values. As a result, different parts of a string can originate from different string literals. The origin information therefore includes all subparts of a string alongside the identifier of the generating string literal and offset.

```
var x = "ab";    //[["ab",23,0]]
var y = x + "c";//[["ab",23,0],["c",42,0]]
var z = y[1];    //[["b", 23,1]]
```

JavaScript code that is not part of the program, especially built-in/native code, is not subject to source code rewriting. To ensure correct behavior for applications passing tagged strings to built-in functions, the tagged string values are wrapped in a proxy (Van Cutsem and Miller 2010) that automatically converts tagged strings to primitive strings when no instrumentation is possible or necessary (e.g. for parsing strings as integers). Additionally, input events from the DOM passed to event handlers are wrapped in a proxy membrane that transparently converts primitive strings to tagged strings. While this approach preserves program behavior, it is possible for a string to 'lose' its origin information due to built-in JavaScript functions.

5.3 Programming Environment Integration

The kinds of UI interactions supported for live programming depend on the domain and concrete representation of the output. In the context of the live programming environment for JavaScript, the output is a tree of HTML/DOM elements and attributes. With string origin tracking, plain text content, attributes and element names can potentially contain origin information.

The programming environment shown in Figure 5 supports two ways of manipulating the output for the purpose of live program synthesis. The developer can either edit the raw HTML code or manipulate the graphical user interface. The HTML representation has the advantage that all parts of the output including element names and attribute keys can easily be modified textually. Manipulation of the actual UI is limited to the plain text content of visible HTML elements but is highly intuitive and immediate.

Given a modified DOM tree, the program synthesis generally follows the following informal algorithm:

1. Determine the previous output based on the current rendering code and application state.
2. Compute the difference between the provided example and previous output. (The modified DOM tree either has new characters inserted, existing characters removed or both[5].)
3. Check origin information of the modified characters. Modifications to string parts that do not have origin information cannot be handled and will be suppressed.
4. Determine source code location of the generating string literals using string origin information and a mapping from string literals to AST nodes.
5. Use the source code location and computed offsets to insert or delete characters in the program source code.
6. Recompile code and obtain a new rendering method.
7. Render output using the modified rendering code and the current program state.

[5] Changes to the tree structure of the DOM output are not currently supported and remain future work.

6. Discussion and Conclusions

Based on both live programming and programming by example, we introduce *live programming by example* as a way to change the code of a running program by direct manipulation of its user interface. We also describe a concrete live programming environment for JavaScript that allows changes to string literals in the source code by editing text in HTML/DOM output.

The approach presented in this paper requires applications to separate rendering from event handling. Thereby, any updates to the rendering code can be synthesized and applied immediately. Updates to the event handling code will only affect subsequent events and therefore live programming by example is not directly applicable to the event handling code. A possible solution is to replay past events instead of resuming execution with the existing state. However, it is not always clear how many events have to be replayed as replaying all events may not be practical or desirable for long-running applications and replaying just the last event may not suffice.

Changing string literals in the program by manipulating text in the output is a very simple implementation of live programming by example and thereby avoids ambiguities that are common in program synthesis applications[6]. However, more complex live programming by example solutions have to address potential ambiguities with heuristics or manual user intervention.

Finally, the program environment presented in this paper mainly serves as a precursor for future systems that support more sophisticated program synthesis guided by more flexible forms of direct manipulation. Moreover, the approach still needs to be evaluated for larger applications and development tasks — potentially as part of a user study.

References

E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013. ISSN 0360-0300. doi: 10.1145/2501654.2501666.

S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462170.

[6] The string synthesis technique is still ambiguous as inserted characters between two different generating string literals can be inserted either at the end of the first string literal or the start of the next one. We currently resolve this ambiguity by always prioritizing the first string literal.

R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, June 2016.

E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462161.

J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568250.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

C. M. Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Cambridge, MA, USA, 2003. AAI0805688.

M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 13–23, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2.

G. Kransner and S. Pope. Cookbook for using the model-view-controller user interface paradigm. *Object Oriented Programming*, pages 26–49, 1988.

G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988. ISSN 0896-8438.

H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

J. H. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM. ISBN 0-89791-709-X.

S. McDirmid. Living it up with a live programming language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

S. McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4.

S. McDirmid and J. Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 1–10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1.

C. Schuster and C. Flanagan. Live programming for event-based languages. In *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop*, REBLS '15, October 2015.

B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, Aug. 1983. ISSN 0018-9162.

E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4.

T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4.

X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 16:1–16:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9.