Generating Interactive Web Pages from Storyboards

Pavel Panchekha
University of Washington, USA
pavpan@cs.uw.edu*

ABSTRACT

Web design is a visual art, but web designs are code. Designers work visually and must then manually translate their designs to code. We propose using synthesis to automatically translate the storyboards designers already produce into CSS stylesheets and JavaScript code. To build a synthesis tool for this complex domain, we will use a novel composition mechanism that allows splitting the synthesis task among domain-specific solvers.

We have built a domain-specific solver for CSS stylesheets; solvers for DOM actions and JavaScript code can be built with similar techniques. To compose the three domain-specific solvers, we propose using partial counterexamples to exchange information between different domains. Early results suggest that this composition mechanism is fast and allows specializing each solver to its domain.

CCS Concepts

•**Software and its engineering** → **Automatic programming;**

Keywords

Synthesis, web design, cascading style sheets, JavaScript

1. INTRODUCTION

Web design is a visual art, but must ultimately be distributed as HTML, CSS, and JavaScript. This leads to a tension between free visual manipulation and automatic code generation. Tools like Photoshop allow designers to manipulate their design visually but require the design to be manually translated to code; tools like Dreamweaver automatically generate CSS and HTML, but restrict how designers manipulate their designs. We want to allow designers to work visually and then automatically generate the HTML, CSS and JavaScript that implements their design.

Designers work by producing *storyboards*: renderings of different states of a web page, with transitions between states labeled with a user action. Translating these storyboards to web pages is challenging for designers: subtle margin collapsing and float rules make CSS confusing and hard to learn; API limitations make the DOM hard to manipulate; and browser bugs and unintuitive semantics make JavaScript difficult to write. Recent advances in program synthesis make it possible to eliminate this manual translation process. Crucially, synthesis tools mesh well with designers' workflows: designers already produce the storyboards that a synthesis tool would use as its input, so designers can keep the same separation of tasks while shortening iteration time and focusing on the core visual task.

A synthesis tool must reason about the interaction of CSS, the DOM, and JavaScript; yet the three languages are so different that reasoning about them simultaneously is beyond the state of the art. To simplify the task, we propose a novel technique for building large and complex synthesis tools out of modular *solvers*. Each of HTML, CSS, and JavaScript will be handled by a domain-specific solver that reasons about that language exclusively. Each solver "inverts" its language: as an interpreter runs a program to produce input-output pairs, so a solver generalizes input-output pairs to produce programs. Such solvers have become possible thanks to advances in program synthesis. These solvers must then be *composed* to form the overall synthesis tool. We propose a novel method of composing solvers using *partial counterexamples*. In this method, each solver to reasons only about its domain, and uses partial counterexamples to communicate information between domains.

Background. Recent work in software engineering translates programming languages into mechanically-solvable equations. This method makes it possible to "invert" a programming language: to use mechanical reasoning to find a program in the language that satisfies certain properties. This work has produced major successes, such as FlashFill [2], Spiral [7], and solver-aided languages [8, 9]. These tools translate a language to a representation that can be mechanically reasoned about, then uses this mechanical reasoning to find programs that generalizes input-output pairs. We propose to extend these techniques to build domain-specific solvers for CSS, the DOM, and JavaScript.

2. APPROACH

The challenge of building a synthesizer for interactive web pages can be split into two steps. First, domain-specific *solvers* for CSS, the DOM, and JavaScript are written. Each of these solvers can mechanical reason about its language, and uses mechanical reasoning to generate programs from input-output pairs. Second, these domain-specific solvers are *composed* into the overall synthesis tool. This composition will use partial counterexamples to communicate information between the domain-specific solvers.

Domain-specific solvers generalize input-output examples to programs in their language. In this way, a domain-specific solver "inverts" its programming language's implementation. Our domain-specific solvers for CSS, the DOM, and JavaScript will target the subsets of those languages. For CSS and the DOM, the state of the art in synthesis makes it possible to target large subsets; for JavaScript, even small subsets suffice for implementing dialogs, dropdowns, accordions, and other common web interactions.

Subsets of CSS, the DOM API, and JavaScript are well-adapted to program synthesis. A core component of modern program synthesis is the representation of a language's semantics in a formal logic

*Advised by Michael Ernst

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983948

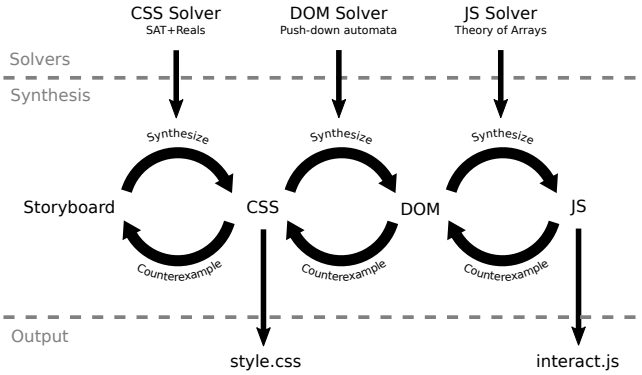

Figure 1: Composing domain-specific solvers into a synthesis tool for interactive web pages. Domain-specific solvers must synthesize code from input-output examples and counterexamples, or must provide partial counterexamples when no code can be synthesized. Each solver reasons only about its domain, and the partial counterexamples it creates communicate information to the other solvers.

with efficient decision procedures. Such a representation exists for all three of CSS, the DOM, and Javascript: we have already developed an encoding of CSS to the theory of quantifier-free linear real arithmetic [6]; the DOM API can be represented by push-down automata [3]; and the jQuery API can be represented in the extended theory of arrays. The existence of such a representation suggests that reasonable extensions of standard synthesis techniques will apply to all three web languages.

The domain-specific solvers must be composed into an overall synthesis tool. To synthesize a web page from a storyboard, our tool will synthesize a CSS stylesheet for the pages in the storyboard; then synthesize, for each transition, a sequence of DOM actions; and finally synthesize JavaScript code that executes those DOM actions. However, composing these domain-specific solvers is not as easy as sending output from one to the next. Not every CSS stylesheet, has a short sequence of DOM actions that implements the desired transitions between pages; nor does every sequence of DOM actions map to simple JavaScript code. We will use a novel composition method that uses partial counterexamples to communicate information between the domain-specific solvers.

In my proposed method, whenever a solver fails, it will identify the portion of its input that makes its synthesis problem impossible. These *partial counterexamples* are standard in the synthesis literature. Each partial counterexample produced by a solver is a precondition on that solver's input, which must be satisfied for that solver to succeed (see Figure 1). Whichever solver generated that solver's input must then synthesize a new input which avoids the partial counterexample. For example, a partial counterexample from the JavaScript synthesizer might be a DOM action that cannot both be performed in JavaScript code. The DOM solver must generate new DOM actions without using that one. This method generalizes clausal learning from satisfiability solving [5] and counterexample-guided inductive synthesis from program synthesis [4]. Thanks to this composition mechanism, each solver need only reason about its domain, yet information from different domains is passed between them to account for the interaction between the domains.

Expected Contributions. Besides the web page synthesizer itself, we expect several contributions in program synthesis. The CSS, DOM, and JavaScript solvers will use novel representations amenable to automated reasoning; these representations will be useful in automated reasoning tasks targeting similar languages. Composing domain-specific solvers using partial counterexamples would be a significant contribution to the synthesis literature, which has until now focused synthesizing programs in a single language.

3. CURRENT RESULTS

We have built a solver for CSS over the last year [6], which can synthesize stylesheets for realistic web pages in a few minutes. It passes the thousands of relevant tests from the official CSS WG test suite [1], showing that it understands CSS accurately and demonstrating that building solvers for web languages is possible.

We have also begun working on the proposed method of composing solvers. We have split the CSS solver into three separate solvers: solvers for CSS selectors, for CSS properties, and for the values of those properties. These three solvers are then composed using partial counterexamples. Each of the three components of the CSS solver uses a different representation of CSS, so that the overall modular solver is faster than the earlier, monolithic version. This exploratory work is an early demonstration of solver composition using partial counterexamples.

Evaluating Results. Each solver can be tested independently for correctness and expressivity: correctness by ensuring that all programs generated by the solver have the intended behavior when run in a browser, and expressiveness by ensuring that the solves can synthesize programs to implement the appearance or interactions of popular real-world websites.

The composition of domain-specific solvers into a synthesis tool for interactive web pages can be evaluated by using it to synthesize common web components (drop-down menus, carousels, accordions) and popular real-world web pages. The ultimate test of my work is its utility to designers. Designers should be able to use my work to turn the storyboards that they produce into interactive web pages. This capability is best evaluated by a user study with a group of web designers.

4. REFERENCES

[1] CSSWG. CSS2.1 test suite, 2011.

[2] S. Gulwani. Automating string processing in spreadsheets using input-output examples. POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[3] M. Hague, A. W. Lin, and L. Ong. Detecting redundant CSS rules in HTML5 applications: A tree-rewriting approach. *CoRR*, 2014.

[4] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. OOPSLA '13, pages 407–426, 2013.

[5] J. P. Marques-Silva and K. A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[6] P. Panchekha and E. Torlak. Automated reasoning for web page layout. OOPSLA '16, 2016. *To appear*.

[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[8] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. PLDI'14, 2014.

[9] R. Uhler and N. Dave. Smten with satisfiability-based search. OOPSLA '14, 2014.