Increasing Human-Tool Interaction via the Web

Thomas Ball Peli de Halleux Daan Leijen Nikhil Swamy
Microsoft Research
{tball,jhalleux,daan,nswamy}@microsoft.com

Abstract

Software tools researchers can accelerate their ability to learn by exposing tools to users via web technologies, allowing them to observe and test the interactions between humans and tools. At Microsoft Research, we have developed a web service (`http://www.rise4fun.com/`) for such a purpose that is available for community use.

Keywords web services, software tools

1. Human-Tool Interaction

The purpose of a software tool is to facilitate human computer interaction so as to increase programmer productivity or improve program quality/performance. However, software engineering and software tools researchers seldom get to observe the interaction between humans and the tools they create. To start with, designing and implementing a new tool is a complex endeavor; then, attracting just a small community of users to the tool is an even bigger challenge.

Today, there's good news for researchers in software engineering in general and software tools in particular: human-tool interaction is undergoing a radical shift due to web technologies, allowing unprecedented access to users as they edit, compile, run and debug programs.

In the past, researchers offered antiquated platform-specific command-line interfaces to tools (not even a remote possibility for a generation trained on smartphones and tablets), integration with baroque editors such as emacs, or in some cases integration with professional development environments like Eclipse. Implementing these interfaces represented a significant investment of time and energy and and the user experience was far from optimal. As a result, deployment of software tools to users was a source of friction, for both researcher and user.

Over the past few years, we've reached a turning point. Web technologies such as HTML, CSS and JavaScript have matured, standards are better supported by most browsers, and JavaScript performance is good enough for hosting substantial applications. Furthermore, as many languages can be compiled to JavaScript, it is possible to use JavaScript to provide an end-to-end experience with a software tool, even if this tool targets a language other than JavaScript. This makes it much easier to reach out to users and have them experiment with tools from the familiar interface of their web browsers, while the tools are maintained and updated on web servers. The uniformity of the browser interface and the vast population on the web open the door to observing the interaction between software tools and users on a world-wide scale.

This a classic win-win situation: the burden on both software tool researchers and users is reduced, enabling a much more fruitful discussion, backed by empirical evidence, about new ideas in software engineering. In particular, we believe that by embracing web technologies, software engineering and tools researchers stand to gain in at least the following ways:

- *Access to many more users*. The ability to simply go to a web page and start using a tool should not be underestimated: eliminating the need for users to download and install software, freeing the user from becoming enmeshed in platform requirements, is huge. Providing a "friction-free" experience for the end user does not require heroic efforts on the part of the researcher. For example, we have leveraged the web to make it easy to publish on-line tutorials that can help bring users up to speed quickly on new tools. Later, we discuss our experience deploying many tools via web technologies.
- *Detailed insights into how their tools are used.* When tools are hosted on web servers, the researcher can see all the programs being submitted as input to their tool. Over time, such a corpus of programs can be mined for useful data. For example, by looking at the user history we can learn about mistakes users make and how they fix them [6]; which tool features trip up users and which are less error-prone; and how much various features of a tool are used.
- *Improved reliability of both theories and implementations*. Building a large regression suite for a tool is hard work. But, by leveraging the crowd to provide test cases, we can find and fix bugs in our implementations. For researchers who develop formal models of existing software or hardware artifacts, the crowd can provide a useful "experimental semantics" to cross-validate the researcher's math. For researchers, such as those working in security, who work based on assumptions about the capabilities of specific attackers, the crowd can be used to vet the pragmatism of such assumptions.

The remainder of this short paper gives details on the basic design choices for making software tools available on the web (Section 2), presents a number of examples we have deployed (Section 3), provides a basic overview of the `RiSE4Fun` service (Section 4), takes a deep dive on one of the tools (Section 5) and concludes with a call to action and discussion of open issues (Section 6).

PASTE'13, June 20, 2013, Seattle, WA USA
Copyright © 2013 ACM 978-1-4503-2128-0/13/06. . . $15.00.

2. Web-based Human-Tool Interactions

Providing a browser-based software tool that meets the expectations of a modern programmer may seem to be non-trivial. Users have come to expect features like syntax highlighting and correction to more elaborate features like type-based code completion in their programming environments. Many of these features normally require significant implementation effort.

At Microsoft Research, we have developed basic infrastructure to simplify many of these tasks in a language-independent manner. For example, we have developed a declarative specification language for many services (like syntax highlighting, brace matching, auto indentation etc.) which significantly reduces the burden placed on tool authors. These specifications are written as JSON values and integrate directly into the web eco-system too. Section 4.2 provides more details on this language.

Behind the browser interface itself, one needs to rethink the architecture of the tool environment. Here we sketch three basic architectures that we have deployed:

1. *Tool hosted on a web server*. In this architecture, the tool and its associated run-time, if any, reside on a web server. Users submit programs to the web server, which are run through the tool, and the tool output sent back to the user. Precautions must be taken to protect the web server from users with malicious intent via sandboxing mechanisms, of which there are many. The service `http://rise4fun.com/` works exactly this way.
2. *Tool hosted on web server, produces JavaScript as output, run in web client*. In this architecture, the tool remains on the web server but outputs JavaScript on the backend, which is then incorporated into a web page and executes in the browser.
3. *Tool on web client*. In the previous two architectures, the target language of the tool itself is of little concern, as long as that language can be hosted on a web server. In the third architecture, the tool itself can be compiled to JavaScript and hosted in the web client, allowing for much more dynamic and fine-grained interactions.

3. Examples

In this section, we give brief descriptions of a variety of tools from Microsoft and their deployment via web technologies, covering all three architectures described above. The following tools are examples of architecture #1:

- The web site `www.pex4fun.com` allows people to test their programming skills on coding puzzles, where an automated test generation tool called `pex` [7] runs on a web server to produce both passing and failing tests for the user. The user responds to the results of the test by modifying their program until `pex` cannot find inputs that distinguish the user's program from a (secret) program stored on the server. This web site provided valuable data on mistakes programmers make, leading to the research reported in [6].
- *Z3* is an automated theorem prover that exposes various logics and queries via its C API. [1] A Python set of wrappers around the C API automatically handles many coercions between Z3 types, providing scripting level support for Z3.[1] This has enabled more people to learn Z3 on the web, where Python's operator overloading makes it possible to construct Z3 terms directly with Python's expression and list comprehension syntax.
- *Dafny* is a simple object-oriented language [5] designed with verification in mind.[2] Dafny has many users in education, as evidenced by the over 70,000 Dafny programs submitted to `RiSE4Fun` during the Spring 2012 semester. The Dafny verifier compiles and verifies Dafny programs (using the Z3 theorem prover) and returns the results of verification to the user.

The following tools are examples of architecture #2:

- *Bek* is a domain-specic language [4] for expressing text to text transformers and for analyzing various properties of those transformers.[3] The Bek compiler has a JavaScript backend, which allows the Bek output to be hosted and executed in the web client, where the user can type in text and immediately see it transformed.
- $F^\star$ is a verification-oriented dialect of ML.[4] Recent work uses it to develop a translation semantics for JavaScript [2], and then uses these semantics to prove a compiler from $F^\star$ to JavaScript fully abstract, which we provide more details on in Section 5.
- *Koka* is a function-oriented programming language that separates pure values from side-effecting computations, where the effect of every function is automatically inferred.[5] Koka has many features that help programmers to easily change their data types and code organization correctly, while having a small language core with a familiar JavaScript like syntax. Koka runs on the server but can compile to JavaScript and run the resulting programs on the client. The server that runs the Koka compiler also implements a mode where the compiler runs as a checker: during development of a program, the client continually sends the (partial) program to the server which returns a list of errors and warnings.

The last two tools represent architecture #3—the tool and runtime are coded in JavaScript, all running within the web browser:

- *TypeScript* is a typed superset of JavaScript that compiles to JavaScript. The TypeScript compiler is itself written in TypeScript and can be hosted in the web browser. As a result, in the TypeScript "playground"[6], the user can input TypeScript code on the left side of the web page and immediately see the compiled JavaScript on the right hand side, and then run it.
- *TouchDevelop* is a programming language and complete programming environment for programming via touch for smart phones and tablets. It is written entirely in TypeScript, compiled to JavaScript and hosted in the web client.[7]

4. The `RiSE4Fun` Interface

The `RiSE4Fun` service provides a web front end for software engineering tools, supporting architectures #1 and #2. It allows tool developers to showcase their tools in any environment, helps reviewers try out tools, and helps instructors deliver classes without worrying about putting a tool on every student machine.

4.1 Web Interface

The `RiSE4Fun` service is documented at `http://rise4fun.com/dev`. Here we provide a brief overview of the features of the service. The service uses the HTTP protocol carrying JSON values. To integrate a tool into `RiSE4Fun`, a tool author provides a simple web service that executes their tool on a server and returns the output. The tool author must implement the following three interfaces:

- `metadata` GET, returns the current metadata information about the tool, as well as built-in samples and, optionally, tutorials.

[1] `http://rise4fun.com/Z3Py/tutorial`

[2] `http://rise4fun.com/Dafny/tutorial`

[3] `http://rise4fun.com/Bek/tutorial`

[4] `http://rise4fun.com/fstar/tutorial`

[5] `http://rise4fun.com/koka/tutorial`

[6] `http://www.typescriptlang.org/Playground/`

[7] `http://touchdevelop.com/`

- `run` POST, gets the source input and returns the tool output. The returned version number is used by `RiSE4Fun` to refresh the metadata.
- `language` GET, (optional) returns the language syntax definition that will be used to do the syntax highlighting in the editor. More details on the language syntax definition are given later.

In return for implementing this service, the `RiSE4Fun` services provides a host of capabilities, as documented at `http://rise4fun.com/rest/help`, some of which include:

- `ask` POST: executes a tool over a program given its program source, returning the output as a raw string;
- `program/{tool}` POST: uploads a program and receives a (permalink) id back as a raw string. This is very helpful for sharing programs with other people via `RiSE4Fun`.
- `live/rss` GET: provides a live stream of queries submitted to `RiSE4Fun`, which can be filtered by tool.

Additionally, the `RiSE4Fun` service implements HTML/JavaScript sandboxing, which allows tools to return HTML and JavaScript to be interpreted in the browser.

4.2 Language Syntax Definition

An important part of many software tools is a powerful editor that can do proper syntax highlighting, and offer support for things like brace matching and auto indentation, extending all the way to full intellisense, name completion, and incorporation of tool output. As a first step we have developed a powerful declarative specification language for enabling custom syntax highlighting. These descriptions can be represented as a pure JSON value which makes it very easy to share custom syntax highlighting for any new language and tool. The following listing gives an example specification:

```
displayName:    'MyLanguage',
mimeTypes:      ['text/x-mylang'],
fileExtensions: ['ml'],

keywords:       ['abstract', 'continue', 'for', ...],
typeKeywords:   ['boolean', 'double', 'int', ...],

// The main tokenizer for our language
tokenizer: {
  root: [
    // identifiers and keywords
    [/[A-Z]\w*/,'type.identifier' ],
    [/[a-z_]\w*/, {
       cases: { '@typeKeywords': 'type.keyword',
                '@keywords':     'keyword',
                '@default':      'identifier' } }],
    // include the rules from the whitespace state
    { include:  '@whitespace' },
    ...
  ],
  whitespace: [
    [/\s+/,     'white' ],
    [/\/\*/,    'comment', '@comment' ],//enter comment
    [/\/\/.*/,  'comment.line'],
  ],
  comment: [
    [/[^\/*]+/, 'comment' ],
    [/\/\*/,    'comment', '@comment' ],//nested comment
    [/*\//,     'comment', '@pop'  ],
    [/[\/*]/,   'comment' ]
  ],
...
```

Note how it is easy to specifiy sets of names, like `keywords`, and match on those. The string results, like `'type.identifier'`, become literal CSS class attributes in the editor rendering that can be used to fully customize the look of the different lexical elements. Strings that are prefixed with an `@` sign refer to attributes in the specification. This is used for example to 'enter' the `comment` state. The lexer states are treated as a stack so we can properly process nested comments, using the special `'@pop'` action to pop a state and return to a previous one. Together with dynamic state suffixes, this gives our specifications the expressive power of push-down automata and they can be used to correctly highlight many complex lexical specifications.

As part of the syntax highlighting, we can also describe various editor related actions like brace-completion and auto-indentation. For example, it is easy to specify that when a user enters a left-brace ({) that a right-brace is inserted automatically (}), and when the user follows with an enter-key, that the cursor is indented correctly on the next line, while the right-brace is correctly outdented on the following line. Such actions are specified using the `bracket` attribute that specifies whether a recognized bracket is an open or close bracket. The editor then uses the CSS class to match up braces and do auto-indentation.

As an example, here is how we specify brace matching for HTML. This is a more dynamic problem since in general we would like to match up an arbitrary open tag, like `<foo>` with its closing tag (`</foo>`). Here is how we can specify this:

```
[/<(\w+)(>?)/,   { token: 'tag-$1', bracket: '@open'}],
[/<\/(\w+)\s*>/, { token: 'tag-$1', bracket: '@close'}],
```

Here, we use the special escape character $\$n$ to substitute the nth capture group of the regular expression. By using this in the returned CSS class, the editor can now dynamically match up the brackets and do proper brace-matching and indentation.

5. Experimental Semantics with the Full-abstraction Game

In this section, we show how a system like `RiSE4Fun` can enable new ways of working with and experimentally validating the formal semantics behind software tools. Formalizing software behavior is an important research activity that often precedes and enables the development of effective software analysis tools. A recent example is the work of Guha et al. [3] who develop a translation semantics for JavaScript. Such formal models provide a precise mathematical understanding of the software systems in question, but, in all these cases, the "real" objects of study are the deployed systems in wide use. Backing up these formal semantics with conformance testing is an important validation activity, particularly as these formalisms start getting adopted as de facto standards. In the sequel, we describe how we used `RiSE4Fun` to crowd-source the experimental validation of a theorem concerning the semantics of JavaScript.

In recent work [2], Fournet et al. take Guha et al.'s Lambda-JS as a de facto model of JavaScript and prove that a compiler from F* (an ML-like programming language being developed at Microsoft Research) to JavaScript is *fully abstract*. Full abstraction is the perfect property of translation—it ensures that the compiler preserves and reflects all source properties into the target language. Slightly more formally, the paper proves that two F* programs e_1 and e_2 are contextually equivalent in F* if and only if their compilations $[e_1]$ and $[e_2]$ and contextually equivalent in JavaScript.

To make this concrete, consider the two F* programs that implement the identity function for booleans:

```
let idbool0 (x:bool) = x
let idbool1 (x:bool) = if x then true else false
```

In F*, there is no context that can distinguish these two functions. Indeed, this is the formal justification behind code transformations that optimize idbool1 to idbool0. Now, for such reasoning to be applicable, clearly, we would like the transformations to be valid even after the program is compiled. However, a naïve translation of the above code to JavaScript might result in the following two JavaScript functions:

```
function f0(x) { return x; }
function f1(x) { return x ? true : false; }
```

However, `f0` and `f1` are not contextually equivalent in JavaScript, as the following context demonstrates:

```
function ctxt(f) { alert(f("hello")); }
```

The function call `ctxt(f0)` will cause the message `"hello"` to pop up, while the function call `ctxt(f1)` will cause the message `"true"` to pop up. Thus, under this naïve translation, simple reasoning principles about source programs are invalidated. In contrast, the compilation scheme developed by Fournet et al. is carefully designed to ensure that all source reasoning principles remain applicable even after translation to JavaScript, thus allowing an $F^\star$ programmer to reason about her programs using the semantics of $F^\star$ alone without ever being concerned with the tricky semantics of JavaScript.

To allow users to explore the concept of full abstraction and to validate the theorem of Fournet et al. against the reality of browser implementations of JavaScript, we used `RiSE4Fun` to create a "full abstraction game". The game proceeds as follows: the user is given two $F^\star$ values, f_0 and f_1, that are presumed to be contextually equivalent in $F^\star$, e.g., these could be idbool0 and idbool1. We compile these two programs to JavaScript values j_0 and j_1, respectively. The user's job, as the attacker, is to write some JavaScript code, J, that can reliably distinguish between j_0 and j_1. The game runs the user's code J on j_0 and j_1 many times, and reports if it can do better than just randomly guessing. The user also has the ability to provide her own candidate $F^\star$ programs in the source equivalence relation and can attempt to distinguish these programs after compilation, thus making the game easily extensible.

If a user is able to win the game (i.e., do better than guessing randomly) then she may have uncovered (1) one of the known limitations of the Fournet et al.'s theorem (e.g., its vulnerability to timing or resource-exhaustion attacks); (2) an experimental behavior in the browser that was not accurately modeled by the formal semantics; (3) an error in the proof of the theorem; (4) an error in the implementation of the compiler; or, all of the above. As such, given the large numbers of the web, we hope this powerful full-abstraction theorem will be scrutinized in much greater detail than the authors could ever hope to do on their own.

The web site[8] contains many more details about full abstraction and how `RiSE4Fun` was used to implement the full abstraction game. The short story is that the game is written in the $F^\star$ language itself. The game consists of a function play that accepts two $F^\star$ functions as well as an attacker function, in addition to a HTML/JavaScript harness for the user interface, and a set of pairs of $F^\star$ functions. The $F^\star$ code is submitted to the `RiSE4Fun` service for the $F^\star$ tool, where it is compiled into JavaScript, inserted into the harness, and sent back to the web browser to start the game.[9] The generated game allows the user to write a JavaScript function (which also will be sandboxed) to try and determine the difference between the JavaScript values j_0 and j_1. This game takes place solely within the web browser, as the $F^\star$ code and runtime have been compiled into JavaScript.

[8] `http://rise4fun.com/FStar/tutorial/jsStar`

[9] Here, the sandboxing features of `RiSE4Fun` are critical to preventing browser-side attacks via arbitrary JavaScript.

6. Conclusion: Call to Action and Open Issues

Software tools are an important interface for human-computer interaction, but often the ideas in research tools don't have the impact they deserve, simply because users are not able to easily experiment with the tools. If you wait for someone to read your PASTE paper and try out your ideas themselves, you'll probably be waiting a decade or more, if you're lucky. By making your tool available on the web, as outlined above, your great ideas will spread and make the world a better place sooner. We especially encourage people to look at `http://rise4fun.com/dev` as an easy way to get started.

While deploying tools on the web improves the reach of a tool (e.g., by removing portability issues), it is not without some difficulties.

First, deploying a tool on the web is not intended to be a complete replacement for making the tool available for closer inspection by other researchers in the community. Providing access to source code, to allow others to read it closely and modify it remains an important element of software science. However, web-based deployment is an invaluable complement to the traditional means of disclosure.

Second, `RiSE4Fun` has no support for graphical user interfaces, other than the automatically generated support for the language editor described in Section 4.2.

Third, where previously a software tool writer may have spent a lot of time working on portability of the tool, they may now have to worry about security, since the tool will be run via a web interface by potentially malicious clients. The `RiSE4Fun` service addresses many of the security issues by default (e.g., by providing a sandbox in which to run client code; running each tool within a "chroot jail" to limit its access to sensitive resources; by running the web server itself in a DMZ; and killing processes after a short time interval to limit DoS attacks). Implementing all of this correctly required additional care and, still, there may be some vulnerabilities that remain. However, by aggregating many tools into the `RiSE4Fun` site, the cost of these measures is significantly amortized.

More experience with deploying tools on the web and greater community participation will help provide a better understanding of these issues (and perhaps identify some others).

References

[1] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS: Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

[2] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL: Principles of Programming Languages*, pages 371–384, 2013.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP: European Conference on Object-Oriented Programming*, pages 126–150, 2010.

[4] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, 2011.

[5] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.

[6] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI: Programming Language Design and Implementation (to appear)*, 2013.

[7] N. Tillmann, J. D. Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *ICSE: International Conference on Software Engineering, Software Engineering Education (SEE) Track*, 2013.