Systematic adaptation of dynamically generated source code via domain-specific examples

Myoungkyu Song
University of Nebraska at Omaha, myoungkyu@unomaha.edu

Eli Tilevich
Virginia Tech

Follow this and additional works at: https://digitalcommons.unomaha.edu/compscifacpub

Part of the Computer Sciences Commons

Recommended Citation
Song, Myoungkyu and Tilevich, Eli, "Systematic adaptation of dynamically generated source code via domain-specific examples" (2017). Computer Science Faculty Publications. 66.
https://digitalcommons.unomaha.edu/compscifacpub/66
Abstract: In modern web-based applications, an increasing amount of source code is generated dynamically at runtime. Web applications commonly execute dynamically generated code (DGC) emitted by third-party, black-box generators, run at remote sites. Web developers often need to adapt DGC before it can be executed: embedded HTML can be vulnerable to cross-site scripting attacks; an API may be incompatible with some browsers; and the program's state created by DGC may not be persisting. Lacking any systematic approaches for adapting DGC, web developers resort to ad-hoc techniques that are unsafe and error-prone. This study presents an approach for adapting DGC systematically that follows the program-transformation-by-example paradigm. The proposed approach provides predefined, domain-specific before/after examples that capture the variability of commonly used adaptations. By approving or rejecting these examples, web developers determine the required adaptation transformations, which are encoded in an adaptation script operating on the generated code's abstract syntax tree. The proposed approach is a suite of practical JavaScript program adaptations and their corresponding before/after examples. The authors have successfully applied the approach to real web applications to adapt third-party generated JavaScript code for security, browser compatibility, and persistence.

1 Introduction

In modern software applications, some of the requirements may only be discovered at runtime. In some execution environments, a combination of users, computing devices, time-of-day, and user interactions often determines the required functionality and execution behavior an application is expected to exhibit. A common approach to fulfilling the requirements discovered at runtime is dynamic code generation.

One domain that has widely embraced the practice of generating code at runtime is web applications, an integral part of the modern computing infrastructure. Web servers host code generators that synthesise custom HTML and JavaScript code for different clients, with the client's browser subsequently downloading and executing the generated code. A web application is commonly divided into a static, fixed part, and a dynamic, generated part. It is the application's dynamic context that determines what code needs to be generated for every combination of the user and execution environment. For example, web applications use the Ajax mechanism [1], in which web browsers issue asynchronous, parameterised requests to server-side JavaScript code generators, which dynamically generate custom client code for different requests.

Web applications commonly integrate and execute the code generated by remote, third-party servers. Ads tailored for individual users and their browsing history, marketing strategies based on individual shopping histories, potential social network connections derived from mining the connection graph—all use dynamically generated JavaScript code, whose shape and features depend on the individual user's behavioural patterns, associations, and execution environments.

Using unsafe coding idioms and violating the host application's policies prevent third-party dynamically generated code (DGC) from satisfying the requirements. Consequently, programmers must adapt such DGC before it can be integrated into and executed by web applications. Unsafe programming idioms violate the security policy in place; they need to be replaced with safe alternatives. Browser-specific APIs would render the application unusable under certain browsers; these APIs need to be replaced with the equivalent functionality supported by the browser in place. A persistent web application needs to remember all user-entered data across invocations, and the data manipulated by the dynamically generated part of the code needs to be appropriately persisted. All these adaptation tasks require transforming the source code, whose exact structure will only be known at runtime.

How can one express the transformations required to adapt the source code that will only be generated in the future? When integrating third-party DGC, programmers can examine this code in a debugger or print it out to the browser's console. Even if examining such debugging information determines that the code must be adapted, programmers lack systematic approaches for effecting the required transformations. An approach that is commonly used under these circumstances is called 'monkey patching', in which a source code fragment (e.g. a function) is rendered as a string and manipulated by means of string matching and modification operations. Although a powerful adaptation technique, 'monkey patching' is inherently unsafe due to its reliance on string operations to modify the source code. In addition, DGC may change every time the application is run. Thus, a systematic approach to transform DGC should be resilient in the presence of some degree of variability in the generated code.

In this study, we introduce a variant of a by-example approach, which has been successfully applied to develop novel program transformation techniques [2–4]. These approaches ask the programmer to provide before and after examples demonstrating a program transformation. From these examples, a general program transformation is derived that can be applied to all other code fragments needing the same transformation. Since DGC needs to be adapted automatically without the programmer being present to control the process, we use a predefined set of before and after examples, with the programmer's role being limited to confirming whether given examples describe the intended adaptation. Our approach is domain-specific in cataloguing the variabilities of common adaptations of JavaScript programs. The approach focuses on JavaScript for two main reasons. First, JavaScript has recently become one of the most widely used [5]. Second, dynamically generating JavaScript code is a practice in modern web applications [6].

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/).
The programmer first chooses an adaptation from our catalogue. Our design assumes the presence of a basic catalogue containing representative examples to be used as guidelines to implement new examples customised for different domains. Ideally, only domain experts should be adding examples to the catalogue.) Then the system presents a series of before/after examples to disambiguate the context under which the specified adaptation should be applied. The system checks the programmer's answers for consistency to resolve any conflicting adaptation directives. In the end, the system generates an adaptation script that performs the specified adaptation by directly rewriting the DGC's abstract syntax tree (AST). The script is then included with the web application along with a small library containing our adaptation mechanisms by leveraging domain-specific knowledge. In a traditional by example program transformation approach [2–4], programmers provide before/after examples for a transformation engine, which then generalises the examples into an automated transformation. The automated transformations can then be applied to all the scenarios that are similar to the before/after examples from which the transformation was derived. In contrast, we provide a catalogue of adaptations, each of which comes with a series of predefined before/after examples, which are presented to the programmer. The programmer's responsibility is to identify which before/after examples reflect the intended adaptations. Based on the programmer's input, our approach then generates an adaptation script that parameterises our adaptation library. Next we demonstrate how our approach can adapt DGC in the examples presented in the previous section.

3 Program adaptation by domain-specific examples

Our approach raises the level of automation of by example mechanisms by leveraging domain-specific knowledge. In a traditional by example program transformation approach [2–4], programmers provide before/after examples for a transformation engine, which then generalises the examples into an automated transformation. The automated transformations can then be applied to all the scenarios that are similar to the before/after examples from which the transformation was derived. In contrast, we provide a catalogue of adaptations, each of which comes with a series of predefined before/after examples, which are presented to the programmer. The programmer's responsibility is to identify which before/after examples reflect the intended adaptations. Based on the programmer's input, our approach then generates an adaptation script that parameterises our adaptation library. Next we demonstrate how our approach can adapt DGC in the examples presented in the previous section.

3.1 Sanitising embedded HTML

The purpose of this adaptation is to insert calls to a sanitising library before dynamically generated HTML code is used. However, the programmer may decide that not all HTML code needs to be sanitised. In particular, the adaptation would sanitise only user-selected HTML injected into a DOM tree, as it can potentially introduce XSS attacks. Once the programmer selects the HTML SANITIZING item from the catalogue in Fig. 2a, the adaptation generator then presents three before/after examples. Fig. 2b asks the programmer whether the innerHTML DOM property returned by function getElementsByID should be sanitised. The examples are presented as simplified AST patterns. Intuitively, this example describes a program fragment. The innerHTML property is retrieved from the document object. This example captures an AST pattern. The document, getElementsByID, and innerHTML form a successor relationship.

In the next example (c), the programmer specifies whether to sanitise innerHTML retrieved through a call to getelementBy. In this case, the wild card is used for capturing all APIs with prefix getelementBy. The [10] construct expresses that each element of the array be sanitised. For consistency, the programmer will either include or exclude this and the previous transformations, whose after examples wrap the HTML string with a call to html sanitize. The generated transformation library will contain a stub to this function that the programmer needs to fill in to invoke an appropriate HTML sanitising API. The <STR> keyword stands for any string type, either literal or variable. The first two examples describe the scenarios that commonly occur in
If third-party generated DGC is incompatible with some browser, the code can be adapted by leveraging one of the well-known browser compatibility tables [11]. Since all adaptations of DGC can take place only at runtime, there is no longer any need for conditional browser-specific code—the type of browser in place is already known. Therefore, if DGC contains some API incompatible with the browser in place, the API should be replaced accordingly. To that end, our catalogue contains multiple adaptations specific to browser incompatibility. The details of this before/after example could be found in the Appendix [10].

3.3 Persisting program state

To render a variable persistent, its state should be written to and read from stable storage. This can be accomplished by replacing all the accesses and modifications of a variable with setter and getter methods, a facility provided by built-in __defineSetter__ and __defineGetter__ functions. The issue at hand is what kinds of variables should be persisted. In JavaScript, there are normal, global, and property variables. Our before/after examples determine what type of variable the programmer wishes to persist. In this scenario, the programmer wants to persist normal and property variables, but not global variables. The details of this before/after example creating an adaption script to persist variables could be found in the Appendix [10].

4 Approach

In this section, we describe the architecture, design and implementation of our adaptation infrastructure, AdcJS. We present our DSL that describes before/after examples and transformations. The details of summarising the syntax of the before/after examples and the adaptation scripts could be found in the Appendix [10]. Our adaptation engine applies adaptation scripts with the structural constraints before and after applying transformations to a program in terms of mapping rules and encodes ordering dependencies among transformation types to define which transformation types must be performed before others on composite transformations.

After showing the AdcJS workflow in Section 4.1, we demonstrate how AdcJS applies the dynamic adaptations to the above motivating examples in Section 4.2.

4.1 Infrastructure workflow

We implement AdcJS as a JavaScript library. Programmers declare AdcJS’s library in their applications. To modify dynamically evaluating JavaScript code, AdcJS proxifies related JavaScript functions such as eval, transforming text into executable code. It parses the argument of dynamically evaluating functions into ASTs and matches the ASTs with the before-state patterns specified in the adaptation scripts. When it finds a matched pattern, AdcJS transforms the ASTs based on the after-state patterns. Finally, AdcJS unparses the transformed ASTs to the argument of eval to be evaluated. Fig. 3 shows the dynamic adaptation workflow of AdcJS.

To parse JavaScript code, we use an AST parser, Esprima [12]. To unparses transformed ASTs, we use a code generator, Escodegen [13].

4.2 Transforming adaptation scripts into AST operations

Using a parser generation technique [14], each adaptation script is translated into a sequence of AST operations—Match, Add, Move, and Delete. We define them as the following.

- Match(N_i): find and return the nodes matching N_i.
- Tranx(OP_1,..., OP_n): perform a series of operations OP_i in sequence, or OP_i ∈ {Add, Move, and Delete}.
  - Add(N_i, N_j): add node N_i to node N_j as a child.
Fig. 3 ADGJS: runtime adaptation workflow

Algorithm 1 Translating an adaptation script into a collection of operations

```
input : Let BF and AF be the before/after-examples of an adaptation script

1: OP := ∅
2: foreach node x ∈ AF do
3:   if Equal(x, y) then
4:     NotFound = FALSE
5:   end
6:   if NotFound then
7:     OP := OP \ MoveOp (x, Parent(x), Parent(y))
8:   end
9:   NotFound := TRUE
10: end
11: foreach node x ∈ BF do
12:   if NotFound then
13:     OP := OP \ AddOp (x, Parent(x))
14:   end
15: end
```

4.3 Adaptation examples

To demonstrate how our adaptation infrastructure transforms ASTs of DGC, we revisit the three motivating scenarios described in Section 3.

4.3.1 Sanitising embedded HTML: Fig. 5 shows a tree transformation that inserts a call to function htmlSanitizeRight before HTML text is assigned to property innerHTML. This adaptation comprises matching a tree pattern, and then applying the add and move transformations described above to the matched nodes: Match([N15, N16, N17]) → Tranx(Add(N16, N15), Move(N15, N16, N17)).

This example shows how the original AST on the left is transformed into the one on the right. The before expression of the adaptation script describes the collection of nodes, [N15, N16, N17], that is to be matched; the pattern matching includes node types and program construct names. N16 and N15 are nodes expressing before/after the transformation. In this case, the nodes are matched as follows: node N16 (`innerHTML`) of type property is a direct predecessor of node N15 (`getElementById`) of type function, which in turn is a direct predecessor of node N17 (`document`) of type object. The matching mechanism in place matches both the node types as well as the names of the program constructs they represent.

The AST on the right shows the results of the performed add and move operations. The subtree rooted in N16 was added to N15; then N17 was moved to the rightmost child position, thus becoming a child node of N15. Note that because of the use of a wildcard, this adaptation will be applied to the innerHTML property returned by all the methods in the document objects starting with the prefix getElementsBy: getElementsByName, getElementsById, getElementsByClassName etc. This adaptation's generality is possible only because we use pre-defined, domain-specific before/after examples that encompass our analysis of JavaScript coding idioms. Such a general adaptation would be impossible if JavaScript programmers had to come up with the before/after examples on their own.

4.3.2 Achieving browser compatibility: Fig. 6 shows a tree transformation that adapts DGC to render it browser compatible. In particular, it renames property innerText intotextContent, whenever this property is a successor of document. This adaptation makes DGC compatible with Firefox browsers. This adaptation comprises matching a tree pattern, and then applying the add and delete transformations described above to the matched nodes: Match([N18, N19, N20]) → Tranx(Add(N19, N18), Delete(N19, N20)).
First, properties that are named `innerText` and are successors of `document` are matched, and their direct predecessor nodes identified. A node with the wildcard value of (`*`) represents any single AST node. In this example, the wildcard will match any node, whose direct successor has the value of `innerText` and whose predecessor (direct or indirect) is the `document` object.

Then, a new node `N_{a1}` (`textContent`) is added to the identified predecessor nodes (`N_{b5}`), whatever they happen to be. Finally, the existing node `N_{b6}` (`innerText`) is deleted from the tree. In essence, combining the delete and add operations forms a replace operation. However, to keep our design minimalistic, we chose not to include any operations that can be expressed by combining the existing operations.

4.3.3 Persisting program state: Fig. 7 shows a tree transformation that renders DGC persistent. This adaptation introduces special functions, `defineGetter` and `defineSetter`, which cause all accesses and modifications of a given normal variable or property to be replaced with the provided getter and setter functions. Getters retrieve the requested values from persistent storage, and setters store them there. This adaptation comprises matching a tree pattern, and then applying a pair of add operations to the matched node: Match (`N_{b6}`) → Trax (*): `Add(N_{a1}, N_{b1}), Add(N_{a10}, N_{b1})`.

Node `N_{b6}` represents all the normal variables and properties that are matched. Then, subtrees `N_{a1}` and `N_{a10}`, describing the getter and setter functions, respectively, are added to the root (`*`) of

* stmt: statement • decl: declaration • arg: argument • left: left node • right: right node • var: normal variable • pro: property • N_{b}: node of before • N_{a}: node of after

`context node` `match node` `new node` `removal node` `dependence

![Fig. 5 Transforming DGC to insert html sanitize at the AST level](image5)

![Fig. 6 Transforming DGC to replace innerText with.textContent at the AST level](image6)

![Fig. 7 Transforming DGC to wrap persist APIs with setter/getter at the AST level](image7)
24 websites as reported by www.alex.com. To create a controlled environment, we used TracingSafari, an instrumented version of the Safari 5 browser as described in [15]. This instrumentation approach makes it possible to record the execution traces of JavaScript programs. Although our approach works with standard web browsers and does not require any instrumentation, using TracingSafari to collect and record the test data made our case studies reproducible.

![Image](image.png)

**Fig. 8** Performance in adaptation

The tree. In this transformation, the persisted program construct’s name in N₀ is the same of the literals represented by the nodes Nₐ and Nₐ’. While the literals represented by the nodes Nₐ and Nₐ’ have the values that concatenate the enclosing function’s name and the persisted program construct’s name. For anonymous functions, this adaptation uses the prefix ‘anonFun_N,’ where N is a counter maintained by the transformer.

### 5 Case studies

For assessing ADGJS’s effectiveness, we performed case studies. We first assessed ADGJS’s adaptation of DGC. In the second study, we assessed performance in real scenarios. To guide our evaluation, we defined the following research questions:

- RQ1. Can our approach accurately adapt the DGC of real-world web applications?
- RQ2. Can our approach efficiently transform the DGC of real-world web applications?

#### 5.1 Experimental design

To evaluate our adaptation approach, we applied ADGJS to the DGC found in 14 diverse, real-world, commercial web applications. We selected these applications from the list of the top 24 websites as reported by www.alexa.com. To create a controlled environment, we used TracingSafari, an instrumented version of the Safari 5 browser as described in [15]. This instrumentation approach makes it possible to record the execution traces of JavaScript programs. Although our approach works with standard web browsers and does not require any instrumentation, using TracingSafari to collect and record the test data made our case studies reproducible.

#### 5.2 Study results and discussion

For each web application, we have attempted to locate three kinds of DGC that could be sanitised, rendered browser compatible, and made persistent. For each subject web application, Table 1 reports the total size of the adapted DGC in kB (SZ), the number of AST nodes of the adapted DGC (ND), and the total number of adaptations applied (AS).


<table>
<thead>
<tr>
<th>W</th>
<th>R</th>
<th>Size</th>
<th>Sanitizing Nodes</th>
<th>Adapt</th>
<th>Size</th>
<th>Nodes</th>
<th>Adapt</th>
<th>Size</th>
<th>Nodes</th>
<th>Persisting</th>
<th>Adapt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1.0</td>
<td>244</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>214.6</td>
<td>70,565</td>
<td>15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>97.4</td>
<td>3,237</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>90.1</td>
<td>22,573</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>106.0</td>
<td>22,834</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>990.4</td>
<td>197,912</td>
<td>54</td>
<td>91.6</td>
<td>24,507</td>
<td>4</td>
<td>38.3</td>
<td>8,256</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>3,942.5</td>
<td>446,566</td>
<td>117</td>
<td>2,212.2</td>
<td>362,748</td>
<td>34</td>
<td>231.3</td>
<td>29,050</td>
<td>773</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>37.4</td>
<td>7,940</td>
<td>13</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>76.7</td>
<td>14,667</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>200.7</td>
<td>44,776</td>
<td>8</td>
<td>115.5</td>
<td>17,199</td>
<td>551</td>
</tr>
<tr>
<td>H</td>
<td>10</td>
<td>162.2</td>
<td>31,324</td>
<td>16</td>
<td>80.9</td>
<td>17,726</td>
<td>1</td>
<td>345.0</td>
<td>74,240</td>
<td>1,782</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>993.6</td>
<td>255,347</td>
<td>9</td>
<td>890.1</td>
<td>226,385</td>
<td>7</td>
<td>297.4</td>
<td>96,134</td>
<td>1,270</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>14</td>
<td>661.3</td>
<td>103,993</td>
<td>62</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>663.0</td>
<td>130,586</td>
<td>3,445</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>18</td>
<td>2,279.9</td>
<td>209,482</td>
<td>148</td>
<td>1,354.4</td>
<td>297,585</td>
<td>26</td>
<td>1,535.5</td>
<td>339,537</td>
<td>8,051</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>19</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>77.6</td>
<td>18,411</td>
<td>4</td>
<td>109.2</td>
<td>28,490</td>
<td>1,026</td>
</tr>
<tr>
<td>M</td>
<td>21</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>500.8</td>
<td>102,757</td>
<td>2</td>
<td>704.9</td>
<td>142,533</td>
<td>2,990</td>
</tr>
</tbody>
</table>

5.2 Study results and discussion

For each web application, we have attempted to locate three kinds of DGC that could be sanitised, rendered browser compatible, and made persistent. For each subject web application, Table 1 reports the total size of the adapted DGC in kB (SZ), the number of AST nodes of the adapted DGC (ND), and the total number of adaptations applied (AS).

RQ1. Can our approach accurately adapt the DGC of real-world web applications? Our case studies have confirmed that our approach can be applied to adapt the DGC of real-world applications. The adaptations that we extracted from our predefined, domain-specific before/after examples can be accurately applied to such applications. The accuracy was checked by manually inspecting the adapted DGC. Regarding the validation process, the first author analysed ADGJS’s results. The results then were validated in the meetings with the remaining authors. When there was any disagreement, each issue was put to a second analysis round, and a joint decision was made. In some cases, we could not perform on transformation when an application does not implement APIs related to adaptation in our approach, where the dash character marks the applications, whose DGC did not need any of the studied adaptations. For example, the DGC used by Facebook did not contain any coding idioms that could be sanitised or rendered browser compatible.

Another example, the DGC used by Amazon could not be sanitised, but could be adapted to be compatible with Firefox. As yet another example, the DGC used by LinkedIn could be sanitised, but did not contain any browser-specific idioms.

RQ2. Can our approach efficiently transform the DGC of real-world web applications? To discuss the performance results of our approach, we analyse the asymptotic computational complexity, which can correlate the execution time of our approach with the size of the DGC being adapted. The number of nodes in the DGC’s AST is a more accurate parameter to consider than the DGC’s physical size. Large, text-rich JavaScript codebases can be parsed into ASTs with moderate numbers of nodes. Therefore, we use the AST’s size in all performance-related discussions. For an AST of size n, the complexity of an exhaustive tree walk (we use the depth-first order) to match the nodes to transform is O(n). The complexity of transforming a matched tree node is constant. Thus, the overall complexity of our approach is O(nC), where C is a constant. As a result, the runtime of our approach should be proportional to the AST size of the adapted DGC. Indeed, the results of our performance benchmark, presented in Fig. 8, clearly show that the actual running time of our approach grows linearly with the size of the DGC’s AST. Our approach is efficient in real world settings, since its execution time is directly proportional to the size of the DGC being adapted.
Discussion: How difficult is it for a domain expert to develop a set of before/after examples for a new adaptation? In essence, the before/after examples in our approach configure adaptations rather than provide input to a learning routine to generalise them into a general program transformation. Thus, if an adaptation is amenable to our approach, developing the examples, in which the before/after parts have the distance of one, is rather straightforward. It took us around an hour to design, implement, and verify each set of the before/after examples described in the paper.

6 Threats to validity

Regarding studies on adaptation, in terms of construct validity, the accuracy of the AST parser Esprima [12] and the code generator Escodegen [13] directly affects A$dG$’s capability in DGC adaptation. The correctness of adaptation catalogues also affects its adaptation. When multiple interfering transformations are designed in the same catalogue, A$dG$ may generate false positives or negatives. Our design goal of the adaptation script is to create one-to-one mapping rules in the transformation. We provide a catalogue of adaptations that consists of concrete and abstract pattern matches. To prevent mapping rules from conflicting each other, we present concrete/abstract before/after examples to capture concrete expressions and then partial abstract before/after examples for the abstract representation matches resulting in most specific transformation. In terms of internal validity, we adapt the DGC portions of applications for security, browser compatibility, and persistence. Not all identified DGC portions are indeed to be adapted and could be intentional. For example, if a programmer trusts the server’s execution, they may accept static HTML contents without sanitisation. In terms of external validity, our results do not generalise beyond our data set and the subject applications. Our evaluation with only open source projects that are implemented in JavaScript may not generalise to projects. Further investigation is required to validate A$dG$ on projects that are developed with different settings, such as programming languages, application domains, or development organizations.

7 Related work

7.1 Program transformation by example

Programming by example, a general methodology behind program transformation by example, has been applied to a variety of software development contexts [2, 16–18]. For example, Galenson et al. present CodeHint to interactively transform a program by using code fragments as an example. Model transformation by example (MTBE) [4, 19, 20] is an automated approach for generating transformation rules by applying inductive inference on example-based specifications. By using context and dependent annotations, MTBE translates rules by leveraging constraints and domain-specific knowledge. To map representative examples, pattern matching has been advocated to generalise transformation rules [21–24].

Unlike these prior efforts, our approach presents a predefined, domain-specific set of before/after AST examples for each adaptation for the programmer to confirm. Using predefined adaptations and examples makes it possible for us to adapt DGC automatically outside the programmer’s purview.

7.2 Program transformation languages

JTL [25], JavaCOP [26], and CIL [27] are high-level languages and infrastructures for transforming Java and C programs. A recent work presents Ann, a new language for design and validation of Java annotations [28]. The design of our transformation infrastructure has been inspired by the technique described in these prior efforts, albeit adapted for the needs of JavaScript.

7.3 AST differencing

CHANGE/DISTILLER [29] computes the difference between two program versions from their ASTS. CHANGE/DISTILLER employs AST structural analysis to produce tree modification operations, such as insert, delete, move and update. Similarly, Falleri et al. [30] analyse AST edits, focusing on move and update edit operations to tackle limitations of textual-based different techniques. DOM schema transformation approaches [31–33] infer differences by comparing the ASTs of different versions, including the elements of XML documents. Our approach’s implementation is closely related to these approaches in modifying ASTs directly; however, we also put forward a DSL for before/after examples and adaptation scripts.

7.4 Transformations for web applications

Several recent research studies [34–36] transformed JavaScript using aspect-oriented programming (AOP) configured via XML or expression templates [37] dynamically implementing JavaScript programs at the AST level at runtime. AspectScript [38] extends JavaScript with a dynamic AOP mechanism implemented as a source-to-source translator. Lerner et al. [39] provide an AOP extension for JavaScript, integrated with a JIT compiler, whose aim is to support principled runtime adaptation. BrowserShield [40, 41] have provided their parsers to by rewriting JavaScript to increase the level of security against vulnerable threats of DGCs. In contrast our approach provides before/after domain-specific before/after examples to configure the required transformations.

8 Conclusion

In this study, we presented a systematic approach for A$dG$ code in web applications that follows a program-transformation-by-example methodology. In our prior approaches following this methodology, we provide predefined, domain-specific examples. By approving the examples that describe the desired transformations, the programmer configures an adaptation script. We demonstrated how our approach can adapt DGC for security, browser compatibility, and persistence accurately and efficiently. We have developed a DSL for expressing program transformations at the AST level. Our experimental results of adapting DGC’s from 14 real-world web applications indicate that our approach can become a practical tool in the toolset of web developers.

9 References
