8-21-2017

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

IET Software

Research Article

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

ISSN 1751-8806
Received on 30th August 2016
Revised 18th May 2017
Accepted on 14th June 2017
doi: 10.1049/iet-sen.2016.0211
www.ietdl.org

Myoungkyu Song[1] ✉, *Eli Tilevich*[2]
[1]*Department of Computer Science, University of Nebraska, Omaha, USA*
[2]*Department of Computer Science, Virginia Tech, Blacksburg, USA*
✉ *E-mail: myoungkyu@unomaha.edu*

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].

```
document.getElementById('ID').innerHTML = html_sanitize(
'<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>');
```

a

```
var elem = document.getElementById('anID');
if(isBrowser('mozilla'))
   elem.textContent = '...';
else
   elem.innerText = '...';
```

b

```
EmailCtrler.compose = function (){
  this.from; this.to; this.subject; this.content;
  this.__defineGetter__('from', function (){
    return persist.getItem('EmailCtrler.compose_from');
  });
  this.__defineSetter__('from', function (from){
    persist.setItem('EmailCtrler.compose_from', from);
  });
  ...
};
```

c

Fig. 1 *Motivating examples for security, browser-compatibility, and persistence*
(a) Sanitising HTML codes by a JavaScript API, ***(b)*** JavaScript API differences between web browsers, ***(c)*** Persisting program state

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 other 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 engine. In our case studies, we have successfully applied our approach to adapt the DGC of real, third-party web applications for better security, browser-compatibility, and persistence. Although our approach is JavaScript-specific to take advantage of the ubiquity of web applications, the general principles we have developed can be applied to other languages and application domains.

This study makes the following main contributions:

- A systematic domain-specific approach to **A**dapting **D**ynamically Generated **J**ava**S**cript (ADGJS) code based on predefined before/after examples.
- A domain-specific language (DSL) for specifying and performing transformations of JavaScript ASTs.
- Empirical results of adapting the DGC portions of third-party commercial web applications for security, browser compatibility, and persistence.

2 Motivating examples

Next we present three scenarios arising in web application development that require adapting DGC for security, browser-compatibility, and persistence reasons.

A large class of security vulnerabilities arises as a result of incorrectly or maliciously formed HTML statements dynamically injected into existing HTML code. A particularly dangerous vulnerability is cross-cite scripting (XSS) [7], in which an HTML hyperlink redirects the user to an unsafe website. A known solution to defending against XSS attacks is sanitising—analysing browser DOM trees for the presence of unsafe content and neutralising it. In fact, multiple sanitising libraries [8, 9] have been developed. Hence, when integrating third-party DGC, a web developer may want to invoke a preferred sanitising function before new HTML statements are injected into the DOM tree. However, sanitising all HTML statements can incur a prohibitively large performance overhead. A web developer may decide that some dynamically generated HTML is safe and should not be sanitised. One policy can be to sanitise only the HTML strings assigned to the innerHTML property of the JavaScript DOM API. Fig. 1*a* shows a snippet of JavaScript adapted to include a call to a sanitising library —html_sanitize. The introduced code appears in blue.

Fig. 1*b* demonstrates how introducing a conditional statement can support browser-specific APIs. Another adaptation strategy can detect browser features to determine which API should be used. Fig. 1*c* demonstrates how the state of a dynamically generated Email function can be rendered persistent. Special getter and setter functions can introduce the persistence functionality by means of the persistence library in place.

The above example motivates the need of adapting DGC for the unique requirements of diverse web applications. Although the adaptation may seem straightforward, the main difficulty lies in the need to specify them without knowing exactly what the generated code will look like. Web developers may have a general idea of what these adaptations should entail. However, it is nearly impossible to consider all the possible patterns under which a program needs to be transformed to put these adaptations into effect.

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 '[10] HTML SANITIZING' item from the catalogue in Fig. 2*a*, the adaptation generator then presents three before/after examples. Fig. 2*b* asks the programmer whether the innerHTML DOM property returned by function getElementById 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, getElementById, 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 [$idx] 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

```
[1] HTML SANITIZING
[2] ..
> Select a rule from the following catalog: 1
```

a

```
[BEFORE EXAMPLE]
document
    getElementById(<str>)
        innerHTML = <str>

[AFTER EXAMPLE]
document
    getElementById(<str>)
        innerHTML = html_sanitize(<str>)

> Include this transformation? (1) Yes, (2) No
```

b

```
[BEFORE EXAMPLE]
document
    getElementBy*(<str>)
        innerHTML[$idx] = <str>

[AFTER EXAMPLE]
document
    getElementBy*(<str>)
        innerHTML[$idx] = html_sanitize(<str>)

> Include this transformation? (1) Yes, (2) No
```

c

```
[BEFORE EXAMPLE]
document
    write(<str>);

[AFTER EXAMPLE]
document
    write(html_sanitize(<str>))

> Include this transformation? (1) Yes, (2) No
```

d

```
Script sanitizingScript
document
  getElementById(<str>)
    innerHTML = <str> |
  getElementBy*(<str>)
    innerHTML[$idx] = <str>
=>
document
  getElementById(<str>)
    innerHTML = html_sanitize(<str>) |
  getElementBy*(<str>)
    innerHTML[$idx] = html_sanitize(<str>)
```

e

Fig. 2 *Creating an adaptation script to sanitise HTML from a series of before/after examples*
(a) HTML sanitising item in the catalogue, ***(b)*** Before/after example #1 (included by the programmer), ***(c)*** Before/after example #2 (included by the programmer), ***(d)*** Before/after example #3 (excluded by the programmer), ***(e)*** Resulting adaptation script

user-driven interactions, in which a malicious user may enter an HTML string that can be exploited in a future XSS attack.

In contrast, example (d) describes displaying HTML content sent by the server. If the programmer trusts this server's provider, they may choose not to sanitise their static HTML content. For example, if this content contains advertisements, it may be subject to a legal agreement that forbids modifying it in any way. In other words, the assumption of this adaptation is that the developers of DGC may be negligent in not sanitising user-entered HTML, but they are not malicious to send HTML code containing XSS attacks. The resulting adaptation appears in part (e), which contains a script that agglomerates all the programmer-approved transformations, with the before and after parts separated by the => marker. This script will then be applied to transform the AST of DGC at runtime. These examples are domain-specific so that they capture common coding idioms in browser-based JavaScript.

3.2 Rendering APIs browser compatible

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 `__defineGetter__` and `__defineSetter__` 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, ADGJS. 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 ADGJS workflow in Section 4.1, we demonstrate how ADGJS applies the dynamic adaptations to the above motivating examples in Section 4.2.

4.1 Infrastructure workflow

We implement ADGJS as a JavaScript library. Programmers declare ADGJS's library in their applications. To modify dynamically evaluating JavaScript code, ADGJS 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, ADGJS transforms the ASTs based on the *after-state* patterns. Finally, ADGJS unparses the transformed ASTs to the argument of `eval` to be evaluated. Fig. 3 shows the dynamic adaptation workflow of ADGJS.

To parse JavaScript code, we use an AST parser, Esprima [12]. To unparse 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_x): find and return the nodes matching N_x.
- Tranx(OP_1,…, OP_n): perform a series of operations OP_i in sequence, or $OP_i \in$ {Add, Move, and Delete}.

  ○ Add(N_x, N_y): add node N_x to node N_y as a child.

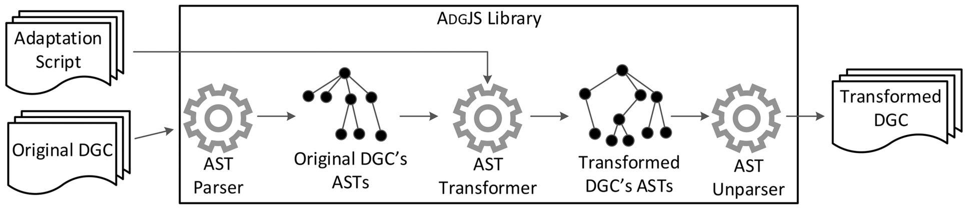

Fig. 3 *ADGJS: runtime adaptation workflow*

input : Let `BF` and `AF` be the before/after-examples of an adaptation script
output: Let `OP` be a collection of the transformation operations: *AddOp* – an add operation, *MoveOp* – a move operation, and *DelOp* – a delete operation

```
1  OP := ∅
2  foreach node x ∈ AF do
3  |  NotFound = TRUE  foreach node y ∈ BF do
4  |  |  if Equal(x, y) then
5  |  |  |  NotFound = FALSE  if LocationNotEqual(x, y) then
6  |  |  |  |  OP := OP ∪ MoveOp (x, Parent(x), Parent(y))
7  |  |  |  end
8  |  |  end
9  |  end
10 |  if NotFound then
11 |  |  OP := OP ∪ AddOp (x, Parent(x))
12 |  end
13 end
14 foreach node x ∈ BF do
15 |  NotFound = TRUE  foreach node y ∈ AF do
16 |  |  if Equal(x, y) then
17 |  |  |  NotFound = FALSE
18 |  |  end
19 |  end
20 |  if NotFound then
21 |  |  OP := OP ∪ DelOp (x, Parent(x))
22 |  end
23 end
```

Fig. 4 *Algorithm 1 Translating an adaptation script into a collection of operations*

o Move(N_x, N_y, N_z): move the child node N_x from its parent node N_y to the new parent node N_z.

o Delete(N_x, N_y): remove node N_x from node N_y.

Algorithm 1 (see Fig. 4) shows our approach to generate AST operations based on adaptation scripts. To create transformation operations, Algorithm 1 (Fig. 4) takes as input the AST patterns representing the before/after examples of an adaptation script; the resulting output is a set of transformation operations that can be applied to the matched nodes of the AST of DGC. Recall that both the before (`BF`) and after (`AF`) parts of an adaptation script are represented as ASTs, which can be traversed and examined. Lines 2 and 6 identify the move operations by calculating the differences between the `BF` and `AF` AST trees. A move operation is generated whenever the `BF`/`AF` trees contain identical subtrees but located at different distances from the root; in other words, these identical subtrees have different tree indexes. Line 11 shows the logic for generating the add operations. An add operation is generated whenever the `AF` tree contains a subtree that is not present in the `BF` tree. Lines 14 to 21 show the logic for generating the delete operations. A delete operation is generated whenever the `BF` tree contains a subtree that does not appear in the `AF` tree. As is common for tree manipulations, these three operations are defined recursively. In terms of the algorithm's efficiency, since it compares all the occurrences of a given subtree parameter with all the other subtrees in before/after trees, the running time is quadratic to the size of the before/after examples.

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 `html_sanitize` right 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([N_b^4, N_b^5, N_b^6]) → Tranx(Add(N_a^1, N_b^7), Move(N_b^8, N_b^7, N_a^3)).

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, [N_b^4, N_b^5, N_b^6], that is to be matched; the pattern matching includes node types and program construct names. N_b and N_a are nodes expressing **b**efore/**a**fter the transformation. In this case, the nodes are matched as follows: node N_b^6 ('innerHTML') of type property is a direct predecessor of node N_b^5 ('getElementBy*') of type function, which in turn is a direct predecessor of node N_b^4 ('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 N_a^1 was added to N_b^7; then N_b^8 was moved to the rightmost child position, thus becoming a child node of N_a^3. 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 `getElementBy`: `getElementByName`, `getElementById`, `getElement-ByClass` 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` into `textContent`, 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([N_b^4, N_b^5, N_b^6]) → Tranx(Add(N_a^1, N_b^5), Delete(N_b^6, N_b^5)).

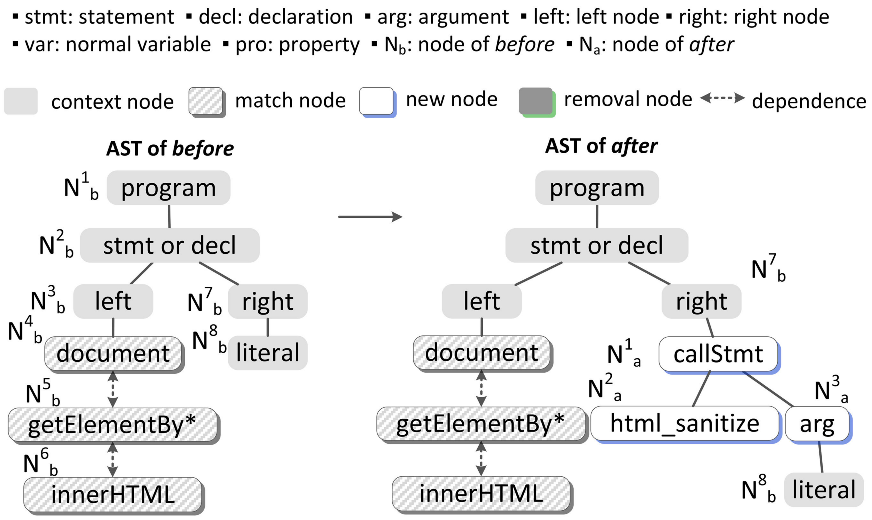

Fig. 5 *Transforming DGC to insert* `html_sanitize` *at the AST level*

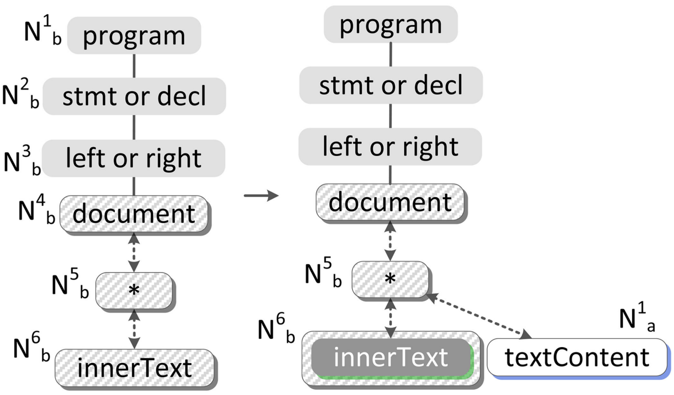

Fig. 6 *Transforming DGC to replace* `innerText` *with* `textContent` *at the AST level*

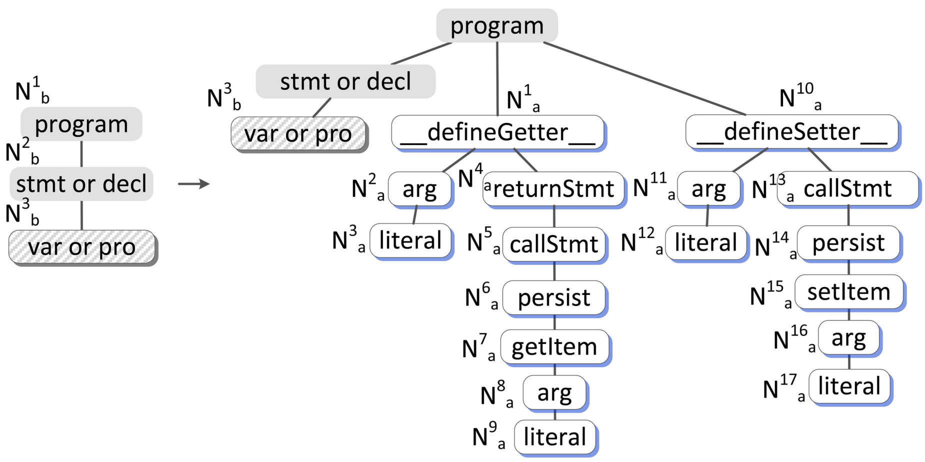

Fig. 7 *Transforming DGC to wrap* `persist` *APIs with setter/getter at the AST level*

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_a^1 ('`textContent`') is added to the identified predecessor nodes (N_b^5), whatever they happen to be. Finally, the existing node N_b^6 ('`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_b^3) → Tranx (*' Add(N_a^1, N_b^1), Add(N_a^{10}, N_b^1)).

Node N_b^3 represents all the normal variables and properties that are matched. Then, subtrees N_a^1 and N_a^{10}, describing the getter and setter functions, respectively, are added to the root ('program') of

Table 1 Using our approach to adapt DGC found in commercial web applications (W: Webpages, R: Rank, A: Facebook, B: Google, C: Youtube, D: Yahoo, E: Wikipedia, F: Live, G: Amazon, H: Twitter, I: Blogspot, J: Linkedin, K: MSN, L: Ebay, M: Bing, and N: Wordpress)

W	R	Sanitizing			Browser compatibility			Persisting		
		Size	Nodes	Adapt	Size	Nodes	Adapt	Size	Nodes	Adapt
A	1	—	—	—	—	—	—	1.0	244	5
B	2	214.6	70,565	15	—	—	—	97.4	3,237	70
C	3	90.1	22,573	3	—	—	—	106.0	22,834	341
D	4	990.4	197,912	54	91.6	24,507	4	38.3	8,256	227
E	6	3,942.5	446,566	117	2,212.2	362,748	34	231.3	29,050	773
F	7	37.4	7,940	13	—	—	—	76.7	14,667	384
G	8	—	—	—	200.7	44,776	8	115.5	17,199	551
H	10	162.2	31,324	16	80.9	17,726	1	345.0	74,240	1,782
I	12	993.6	255,347	9	890.1	226,385	7	297.4	96,134	1,270
J	14	661.3	103,993	62	—	—	—	663.0	130,586	3,445
K	18	2,279.9	209,482	148	1,354.4	297,585	26	1,535.5	339,537	8,051
L	19	—	—	—	—	—	—	169.0	37,697	731
M	21	—	—	—	77.6	18,411	4	109.2	28,490	1,026
N	23	701.0	153,934	44	500.8	102,757	2	704.9	142,533	2,990

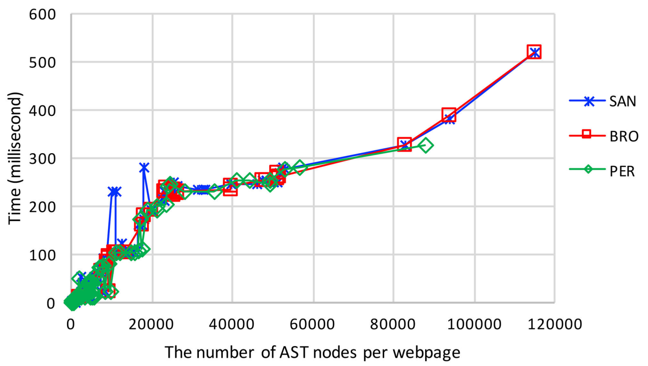

Fig. 8 *Performance in adaptation*

the tree. In this transformation, the persisted program construct's name in N_b^3 is the same of the literals represented by the nodes N_a^3 and N_a^{12}. While the literals represented by the nodes N_a^9 and N_a^{17} 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).

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 $O(n)C$, 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 ADGJS's capability in DGC adaptation. The correctness of adaptation catalogues also affects its adaptation. When multiple interfering transformations are designed in the same catalogue, ADGJS 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 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 generalize to projects. Further investigation is required to validate ADGJS 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 analysis, MTBE infers transformation 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

CHANGEDISTILLER [29] computes the difference between two program versions from their ASTs. CHANGEDISTILLER 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 expressive patterns. AjaxScope [37] dynamically instruments 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 domain-specific before/after examples to configure the required transformations.

8 Conclusion

In this study, we presented a systematic approach for ADGJS code in web applications that follows a program-transformation-by-example methodology. Unlike 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 DGCs from 14 real-world web applications indicate that our approach can become a practical tool in the toolset of web developers.

9 References

[1] Deitel, P., Deitel, H.: '*Ajax, rich internet applications, and web development for programmers*' (Prentice Hall PTR, 2008)

[2] Lieberman, H. (Ed.): '*Your wish is my command programming by example*' (Morgan Kaufmann, 2001)

[3] Meng, N., Kim, M., McKinley, K.S.: 'LASE: locating and applying systematic edits by learning from examples'. Int. Conf. Software Engineering, 2013, pp. 502–511

[4] Balogh, Z., Varró, D.: 'Model transformation by example using inductive logic programming', *Softw. Syst. Model.*, 2009, **8**, (3), pp. 347–364

[5] The top 10 programming languages. http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages, accessed May 2017

[6] Richards, G., Hammer, C., Burg, B., *et al.*: 'The eval that men do: a large-scale study of the use of eval in JavaScript applications'. Int. Conf. Object-oriented Programming, 2011, pp. 52–78

[7] Grossman, J., Hansen, R., Petkov, P.D.,, *et al.*: '*XSS attacks: cross site scripting exploits and defense*' (Oxford, 2007)

[8] Ohara, C.: 'Node validator'. https://github.com/chriso/node-validator

[9] JsHtmlSanitizer. http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer, accessed May 2017

[10] Appendix to Systematic Adaptation of Dynamically Generated Source Code. http://faculty.ist.unomaha.edu/msong/adagejs/appendix.pdf

[11] Compatibility overview. http://quirksmode.org/compatibility.html, accessed May 2017

[12] Esprima: 'ECMAScript parsing infrastructure for multipurpose analysis'. http://esprima.org/, accessed May 2017

[13] Escodegen: ECMAScript code generator from parser API AST. https://github.com/Constellation/escodegen, accessed May 2017

[14] PEG.js. http://pegjs.majda.cz/, accessed May 2017

[15] Richards, G., Lebresne, S., Burg, B., *et al.*: 'An analysis of the dynamic behavior of JavaScript programs'. Int. Conf. Programming Language Design and Implementation, 2010, pp. 1–12

[16] Cypher, A., Halbert, D.C., Kurlander, D., *et al.*: '*Watch what I do: programming by demonstration*' (MIT Press, 1993)

[17] Mandelin, D., Xu, L., Bodk, R., *et al.*: 'Jungloid mining: helping to navigate the API jungle'. Int. Conf. Programming Language Design and Implementation, 2005, pp. 48–61

[18] Galenson, J., Reames, P., Bodik, R., *et al.*: 'Codehint: dynamic and interactive synthesis of code snippets'. Int. Conf. Software Engineering ACM, 2014, pp. 653–663

[19] Varró, D.: ‘Model transformation by example’. Int. Conf. Model Driven Engineering Languages and Systems, 2006, pp. 410–424
[20] Varró, D., Balogh, Z.: ‘Automating model transformation by example using inductive logic programming’. Int. Conf. Symp. Applied Computing, 2007, pp. 978–984
[21] Wimmer, M., Strommer, M., Kargl, H., *et al.*: ‘Towards model transformation generation by-example’. Int. Conf. Annual Hawaii, 2007
[22] Kappel, G., Langer, P., Retschitzegger, W., *et al.*: ‘Model transformation by-example: a survey of the first wave’, in Düsterhöft, A., Klettke, M., Schewe, K.-D. (EDs.): ‘*Conceptual modelling and its theoretical foundations*’ (Springer, 2012), pp. 197–215
[23] Strommer, M., Murzek, M., Wimmer, M.: ‘Applying model transformation by-example on business process modeling languages’. Int. Conf. Conceptual Modeling, 2007, pp. 116–125
[24] Alves, E.L., Song, M., Massoni, T., *et al.*: ‘Refactoring inspection support for manual refactoring edits’, *IEEE Trans. Softw. Eng.*, 2017, (accepted)
[25] Cohen, T., Gil, J.Y., Maman, I.: ‘JTL: the java tools language’. Int. Conf. Object-oriented Programming, Systems, Languages, and Applications, 2006, pp. 89–108
[26] Markstrum, S., Marino, D., Esquivel, M., *et al.*: ‘JavaCOP: declarative pluggable types for java’, *ACM Trans. Prog. Lang. Syst.*, 2010, **32**, (2), p. 4
[27] Necula, G.C., McPeak, S., Rahul, S.P., *et al.*: ‘CIL: intermediate language and tools for analysis and transformation of C programs’. Int. Conf. Compiler Construction, 2002, pp. 213–228
[28] Córdoba-Sánchez, I., de Lara, J.: ‘Ann: a domain-specific language for the effective design and validation of java annotations’, *Comput. Lang., Syst. Struct.*, 2016, **45**, pp. 164–190
[29] Fluri, B., Wuersch, M., PInzger, M., *et al.*: ‘Change distilling: tree differencing for fine-grained source code change extraction’, *IEEE Trans. Softw. Eng.*, 2007, **33**, (11), pp. 725–743
[30] Falleri, J.-R., Morandat, F., Blanc, X., *et al.*: ‘Fine-grained and accurate source code differencing’. Int. Conf. Automated Software Engineering ACM, 2014, pp. 313–324
[31] Cobena, G., Abiteboul, S., Marian, A.: ‘Detecting changes in XML documents’. Int. Conf. Data Engineering, 2002, pp. 41–52
[32] Martin, E.: ‘Toward the automatic derivation of XML transformations’, in Jeusfeld, M.A. and Pastor, O. (Eds) *Conceptual Modeling for Novel Application Domains*, 2003, pp. 342–354
[33] Königs, A., Schürr, A.: ‘MDI – a rule-based multi-document and tool integration approach’, *Int. J Softw. Syst. Model.*, 2006, **5**, (4), pp. 349–368
[34] Washizaki, H., Kubo, A., Mizumachi, T., *et al.*: ‘AOJS: aspect-oriented JavaScript programming framework for web development’. Int. Conf. Aspects, Components, and Patterns for Infrastructure Software, 2009, pp. 31–36
[35] Ofuonye, E., Miller, J.: ‘Securing web-clients with instrumented code and dynamic runtime monitoring’, *J. Syst. Softw.*, 2013, **86**, (6), pp. 1689–1711
[36] Leger, P., Tanter, É., Fukuda, H.: ‘An expressive stateful aspect language’, *Sci. Comput. Prog.*, 2015, **102**, pp. 108–141
[37] Kiciman, E., Livshits, B.: ‘Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications’. Int. Conf. Operating Systems Review, 2007, pp. 17–30
[38] Toledo, R., Leger, P., Tanter, É.: ‘Aspectscript: expressive aspects for the web’. Int. Conf. Aspect-oriented Software Development, 2010, pp. 13–24
[39] Lerner, B.S., Venter, H., Grossman, D.: ‘Supporting dynamic, third-party code customizations in JavaScript using aspects’. Int. Conf. Object-oriented Programming Systems, Language and Applications, 2010, pp. 361–376
[40] Reis, C., Dunagan, J., Wang, H.J., *et al.*: ‘Browsershield: vulnerability-driven filtering of dynamic HTML’, *ACM Trans. Web*, 2007, **1**, (3), pp. 11
[41] Yu, D., Chander, A., Islam, N., *et al.*: ‘JavaScript instrumentation for browser security’. Int. Conf. Principles of Programming Languages, 2007, pp. 237–249