Towards Automatic Client-side Feature Reuse

Josip Maras[1], Jan Carlson[2], and Ivica Crnković[2]

[1] University of Split, Croatia,
josip.maras@fesb.hr
[2] Mälardalen University, Sweden,
jan.carlson@mdh.se, ivica.crnkovic@mdh.se

Abstract. Client-side applications often contain similar features and facilitating reuse could offer considerable benefits in terms of faster development. Unfortunately, due to the specifics of prevailing technologies, the techniques and tools used to support reuse are not as advanced as in other software engineering disciplines and the main method of reuse is still copy-pasting code. Copy-paste reuse can introduce a number of different types of errors that are time-consuming to detect and fix. In this paper we present an automatic method for feature reuse in client-side web applications. We identify problems that occur when introducing code from one application into another, present a set of algorithms that detect and fix those problems and perform the actual code merging. We have evaluated the approach on four case study applications, and the results show that the method is capable of performing feature reuse.

Keywords: Web applications, Reuse, Client-side Analysis

1 Introduction

From the user's perspective, the behavior of a client-side application is composed of distinguishable parts, i.e. features, that manifest at runtime. Similar features are often used in a large number of applications, and facilitating their reuse offers significant benefits in terms of easier development. However, the client-side domain does not offer any widely used feature-reuse method, and code is usually copy-pasted to the new application. Copy-paste reuse can be complex and error-prone. Usually it is hard to identify code for reuse and introduce it into the new application without errors, and there is need for systematic reuse.

A feature is an abstract notion representing a distinguishable part of the system behavior that manifests at runtime triggered by the user [2]. Since client-side web applications are highly dynamic event-driven applications where the majority of code is executed as a response to user-generated events, identifying the exact implementation details of a certain feature is difficult and time-consuming. In our previous work [4] we have introduced a client-side dependency graph that captures dependencies that exist in a client-side web application. The dependency graph is built during the feature identification process [4] by analyzing an execution of a particular scenario demonstrated by the user. By using the

feature identification process, we are able to identify the implementation details of individual features.

In this paper we present a method for code-level feature reuse in client-side web development. We have specified how to reuse features, and when can that reuse be considered successful. Naturally, when merging two code bases a number of problems can arise. We have identified those problems and have developed algorithms capable of detecting and fixing them. The approach has been evaluated by performing the reuse process on four case study web applications, and the evaluation shows that the method is capable of performing feature reuse.

2 Reuse process overview

The goal of the method is to enable code-level feature reuse from one client-side application into an already existing application. Let A and B be two client-side applications, each defined with HTML, CSS, and JavaScript code and resources: $\langle H_A, C_A, J_A, R_A \rangle$ for application A, and $\langle H_B, C_B, J_B, R_B \rangle$ for application B.

An application offers a set of features F, and when a user performs a certain scenario s_i, a feature f manifests. A feature is implemented by a subset of the application's code and resources. However, identifying the exact subset is a challenging task: code responsible for the desired feature is often intermixed with irrelevant code, and there is no trivial mapping between the code and the application running in the browser. In our previous work [4], we have developed a method that can, by analyzing the execution of a scenario causing the manifestation of a feature f_a, identify the subset of the application $\langle h_a, c_a, j_a, r_a \rangle$ that implements the feature. The goal of the reuse method is to enable the inclusion of code and resources $\langle h_a, c_a, j_a, r_a \rangle$ of f_a from application A into the application B, thereby obtaining a new application B' that offers both the feature f_a from A, and the features F_B from B. We consider **reuse successful** if, in the final B' application, the scenario s_a causing the manifestation of f_a can be repeated with the same presentational and behavioral characteristics on h_a, and all scenarios S_B can be repeated with the same presentational and behavioral characteristics on H_B. This means that, in order for the reuse to be correct, there should not be any feature "spilling" – the feature f_a, in the context of B', should not operate on parts of application originating from application B (nor should features from B operate on parts of the application originating from A). With regard to presentational characteristics this means that CSS rules c_a, when included in $C_{B'}$, should only be applied to HTML nodes h_a included in $H_{B'}$ (similarly, C_B should only be applied to H_B). For the preservation of behavioral characteristics, code j_a should only interact with h_a, c_a, r_a, and J_B with H_B, C_B, R_B.

2.1 Process Description

Due to the fact that client-side web applications are highly dynamic event-driven UI applications, we have based the process on the dynamic analysis of application execution while performing scenarios that capture the behavior of individual

applications. As input, the process (Figure 1) receives the whole code of the application A from which a feature f_a will be extracted, a scenario s_a that invokes the feature f_a, a selector that specifies the part of the web page where f_a manifests; the whole code of the web application B where the feature will be reused into, a set of scenarios S_B that capture the behavior of the application, and a reuse position that specifies where the feature will be reused.

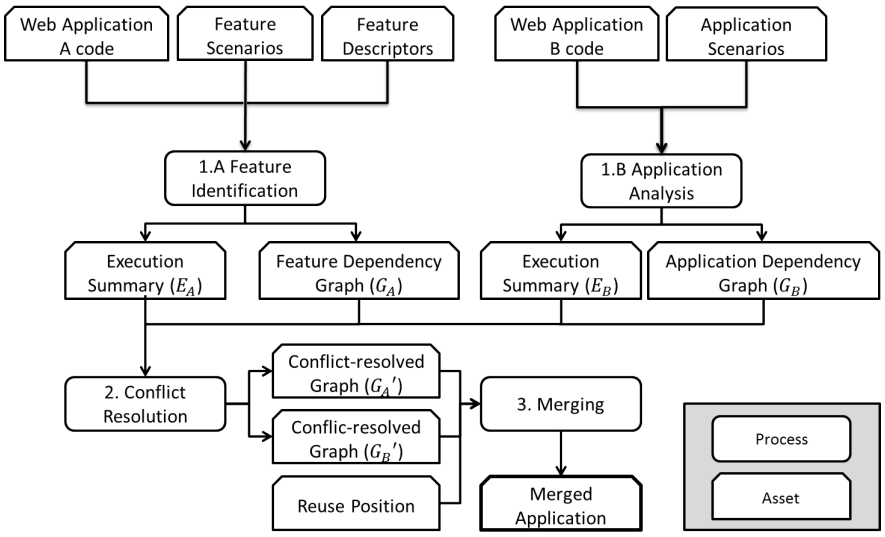

Fig. 1. The process of extracting and reusing features

The reuse process starts by invoking the *feature identification* process [4] (1.A, Figure 1) for application A, and by analyzing the execution of application B (1.B, Figure 1). The feature identification process executes the application with the scenario event trace as a guideline, logs an execution summary, builds a dependency graph, and automatically identifies the subset of the application A that implements the feature f_a. Similarly, the application analysis process analyzes the execution of application B, logs an execution summary and builds a dependency graph. When merging two code bases a number of conflicts can occur. For this reason, all artifacts generated in the previous phases are used as input to the *conflict resolution* phase which automatically fixes the conflicts (or notifies that the fix can not be applied). Next, the modified, conflict-resolved dependency graphs, along with the reuse position are used as inputs to the merging phase where the code of the two applications is merged, and reuse is achieved.

3 Conflict Resolution

Including code and resources of f_a into B changes the situation in both the feature code and the application code, primarily because merging code creates a new page whose DOM is different from what is expected by the code of each individual application. This difference can create a number of problems that are complicated by the fact that the web application code is heavily interdependent (the final result displayed in the browser is an interplay of HTML code, CSS code, JavaScript code and resources), and that a change in one section can propagate to a number of different places. On top of that, due to the dynamicity of JavaScript, both the positions on which the problems arise, and the positions to where they are propagated to can not be accurately determined statically.

3.1 Conflicts

There are two broad types of conflicts: DOM conflicts and JavaScript conflicts.

DOM conflicts – From the DOM perspective, the merging of HTML code can lead to conflicts in naming attributes of HTML nodes (class, id, and name). Since HTML is an error tolerant language, this won't lead to any problems in the DOM itself. However, the naming attributes are used in CSS and JavaScript code, and the main problem with DOM conflicts is that they propagate to CSS and JavaScript code. CSS rules are applied to HTML nodes based on CSS selectors, and if CSS conflicts occur, CSS rules designed to target HTML nodes of one application could, in the final application, be applied to HTML nodes of the other application. This can lead to a number of problems: from not preserving the visual properties, to changing the values of code expressions that access the element's visual properties. JavaScript code interacts with the DOM and accesses HTML nodes by using queries similar to CSS selectors. This means that if there are conflicts in HTML, JavaScript expressions that query the DOM of the page can return different results in the context of the final application than in the original contexts. These differences can lead to a number of errors.

JavaScript conflicts – Apart from conflicts that propagate from HTML and CSS, JavaScript code can introduce a number of errors, mostly because of the use of global variables. JavaScript has different types of global variables, and from the perspective of conflict-handling they can be divided into three groups: *i) standard global variables* created by declaring variables in the global scope, or by extending the global window object (writing to a non-registered identifier, or adding a new property to the window object); *ii) Built-in object extensions* – extending built-in objects (e.g. the Math object, String, Array prototypes); and *iii) Event-handling variables* used by the browser to register event handlers (e.g. onload, onmousemove properties of the global window object). *Standard global variables* can cause naming conflicts; *Built-in object extensions* can cause naming conflicts within the extended objects, and errors can be introduced when iterating over object properties; and *Event-handling variables* can cause problems with property overriding. Similar to CSS type selectors, JavaScript type DOM

queries can come into conflict and a different set of elements, compared to the originating applications, can be returned in the context of the final application.

Conflicts can also occur between resources (images, fonts, videos, files, etc.) if there exist resources with the same identifiers in both applications. These types of conflicts can propagate to HTML, CSS, and JavaScript code, and have to be tracked and handled.

3.2 Fixing Conflicts

The following sections describe algorithms for fixing conflicts in different parts of the application. The process is composed of two steps: *i)* fixing conflicts that arise due to changes in the DOM structures of both applications, and *ii)* fixing problems that happen when merging two JavaScript code bases. Since conflicts can occur both statically and dynamically, all possible conflicts can not be accurately detected with static analysis, and fixes performed with simple string renamings, without taking into consideration the semantics of the changed expressions, can only handle a subset of possible problems (and even then, we can not be sure if they are applied to correct expressions). This is why we heavily rely on client-side dependency graphs and execution summaries to identify the exact conflict positions, and the positions to where these conflicts propagate.

Fixing DOM conflicts – Algorithm 1 describes the process of detecting and fixing conflicts that arise when merging HTML code of two different applications, but that also propagate to CSS and JavaScript. The main idea of the algorithm is to identify all static or dynamic code positions that can cause conflicts due to the fact that a new page will be created by merging the DOMs of both applications. This means replacing conflicted HTML attributes, expanding HTML nodes, and modifying both the CSS rules and JavaScript DOM queries in order to localize them in a way that they only interact with nodes from their respective applications. As input the algorithm receives the dependency graph *fGraph* and execution summary *fExe* of A for scenario u_a, and the dependency graph *bGraph* and the execution summary *bExe* of B for U_B.

Algorithm 1 handleDOM(*fGraph*, *fExe*, *bGraph*, *bExe*)

```
1: attrConflicts ← getHtmlAttrsConflicts(fExe, bExe)
2: resources ← getResources(fGraph)
3: for all item : merge(attrConflicts, resources) do
4:     new ← getName(item, fExe, bExe)
5:     for all pos : getUsagePos(item, fExe) do
6:         if isInHtml(pos) OR isInCss(pos) then
7:             replaceVal(pos, item, new)
8:         else if isInJs(pos) then
9:             replaceDomStrLit(pos, item, new, fGraph)
10: expandNds(getName('f', fExe, bExe), fGraph, fExe)
11: expandNds(getName('b', fExe, bExe), bGraph, bExe)
```

The algorithm finds all conflicts in HTML node named attributes and all used resources, and generates a new, unconflicting name for each item. An item can be used in a number of different positions: in HTML code as node attributes, in CSS code as selectors or key values, and in JavaScript code as assignment or call expressions. (e.g. assignment expressions that modify node attributes, or DOM querying call expressions). If the usage position is in HTML or CSS code then the old value in the feature code is simply replaced with the new, unconflicting value. If the usage position is in JavaScript code, the process traverses the dependency graph and attempts to find the string literal that matches the old value and replace it with the new value. If the string literal can not be found (e.g. is constructed by concatenating strings), then a comment notifying that a conflict was not handled is added to the access position.

The process of handling DOM conflicts continues in line 10, Algorithm 1, by handling conflicts with selectors in CSS and JavaScript. The main idea is to make the selectors more specific by limiting them only to parts of the DOM that match the originating application. Unconflicting names are generated with calls to the *getName* function and are added as attributes to enable differentiation between nodes originating from different applications. Type selectors are expanded so they target only nodes they have targeted in the originating applications (both for CSS selectors, and JavaScript DOM queries).

Algorithm 2 handleJs(*fGraph*, *fExe*, *bGraph*, *bExe*)

```
 1: for all cnflct : getStandardGlobalConflicts(fExe, bExe) do
 2:     renameIdDeps(getDecl(cnflct, fGraph), getNewName(cnflct, fExe, bExe))
 3: for all objExt : getBuiltInObjExtns(fExe) do
 4:     addSkipIterationToAllPropertyIters(getPropertyIters(objExt, bExe))
 5:     if hasConflicts(objExt, fExe, bExe) then
 6:         renameIdDeps(getDecl(objExt, fGraph), getNewName(objExt, fExe, bExe))
 7: for all objExt : getBuiltInObjExtns(bExe) do
 8:     addSkipIterationToAllPropertyIters(getPropertyIters(objExt, fExe))
 9: conflicts ← getConflictedHandlers(fExe, bExe)
10: if sizeOf(conflicts) != 0 then
11:     addInitConflictHandlerObjectAsTopNode(bGraph)
12:     expandWithConflictHandlerCode(conflicts)
13:     addHandlerInvokerCodeAsLastBodyNode(bGraph)
```

Fixing JavaScript conflicts – The main goal of Algorithm 2 is to detect and fix JavaScript conflicts that arise due to global variable naming conflicts, and due to the modifications of the globally accessible objects that can change the behavior of additionally included code. Since conflicts can occur both dynamically and statically, as input the algorithm receives the dependency graphs and execution summaries from both applications. The algorithm starts by finding conflicts regarding standard global variables: for each conflicted variable, a new unconflicting identifier is generated, and all usage positions of that identifier in

the feature code are replaced (if it is not possible, a warning is added). Next, the algorithm is dealing with conflicts that arise by extending built-in objects. For the object extensions in feature code, the algorithm traverses all code positions in application B that iterate over the extended objects and adds a statement that will skip the iteration over the properties extended by the other application. The algorithm proceeds by checking if there are any conflicts with the object extensions done in application B, and if there are, the property names in the feature code are replaced with unconflicting names. The process goes similarly for the application B with the exception that there is no need for handling naming conflicts. Finally, event handler conflicts are handled by inserting code that creates an event-handler-tracker object that keeps track of all registered handlers, replacing conflicting code expressions in both applications with code that reroutes the handler registration and deregistration to the event-handler-tracker, and inserting code that invokes the necessary handlers.

4 Merging code

Once all conflicts have been detected and fixed or reported, the process is finished by merging the code of the feature and the application (Algorithm 3). The main idea is to perform the merge of both the header and body nodes of each application, and then to move the feature nodes to the designated position, without introducing errors. Algorithm 3 works by taking the head children and the body children of the feature graph from application A and appending them to the head and the body node of the application B. Next the HTML nodes that define the feature are selected from the graph with the goal of moving them to a new position defined with the *rSlctr* selector. Since not only feature HTML nodes are selected in the feature identification process (others might be included due to dependencies) it is not always possible to perform the moving of nodes without introducing errors. Some CSS selectors that apply styles to feature nodes, or JavaScript DOM queries, could be structurally dependent on the position of the feature nodes in the page hierarchy, and by moving the feature nodes errors are introduced. In this case we detect and report the error positions (lines 7, 8). Also, due to DOM queries, when moving feature nodes it is necessary to maintain the relative position of the feature script nodes towards the feature nodes (line 11).

5 Case studies

The evaluation of the approach is based on four case study applications divided into two groups: *i)* Group 1, applications 1 and 2 that use the most wide-spread third-party JavaScript library – jQuery; and *ii)* Group 2, applications 3 and 4 developed with the second most-wide spread JavaScript library – MooTools[1]. With these four applications we have created a set of four case studies with a goal to test whether our method is capable of performing automatic feature

[1] http://w3techs.com/technologies/history_overview/javascript_library/all

Algorithm 3 merge(*fGraph*, *bGraph*, *fSlctr*, *rSlctr*)

```
1: for all hChild : getHeadChildren(fGraph) do
2:     if isLinkScriptStyle(hChild) then
3:         appendToHeadNode(hChild, bGraph)
4: for all bodyChild : getBodyChildren(fGraph) do
5:     appendToBodyNode(bodyChild, bGraph)
6: featrNds ← getFeatureNodes(fSlctr, fGraph)
7: for all pos : getStructSlctrsPos(featrNds, fGraph) do
8:     addComment(pos, "Error - struct query")
9: moveNodes(rSlctr, featrNds)
10: scripts ← getFeatureScriptsAffctdPos(fSlctr, rSlctr, bGraph)
11: updatePosition(rSlctr, scripts, featrNds)
```

reuse in different situations (e.g. is the process able to include a feature developed with the jQuery library into the application developed with the MooTools library, and vice versa). Based on the features from each application we have specified Selenium tests[1] that test the correctness of the features in the final application. The case study applications, their tests, and the results are available at: *www.fesb.hr/~jomaras/download/reuseCaseStudies.zip*.

Table 1. Case studies: Lines of code (LOC); Feature (F); HTML modifications (H), CSS modifications (C), JavaScript modifications (J) ; Time – process execution time in seconds;

#	App A	App B	A-LOC	B-LOC	F-LOC	H (A;B)	C (A;B)	J (A;B;B')	Time	Success
1	App 1	App 2	10554	12031	1083	30;0	12;201	9;0;0	179	Y
2	App 1	App 3	10554	5112	1083	29;0	12;11	9;0;0	159	Y
3	App 4	App 2	9083	12031	1258	46;0	23;201	6;0;0	195	Y
4	App 4	App 3	9083	5112	1258	44;0	23;22	N/A	168	N

Table 1 shows the results of 4 reuse case studies. For each reuse we present the total number of lines of code in the application from which a feature was extracted (A-LOC), lines of code of the application where the feature will be reused into (B-LOC), total LOC of the feature extracted with the feature identification process (F-LOC), number of changes done to the HTML code (H), CSS code (C), and JavaScript code that were performed by the process to resolve conflicts; and the total running time of the reuse process[2]. All experiments have been performed with a plugin to the Firefox browser – Firecrow[3], which implements all algorithms described in the paper.

[1] http://docs.seleniumhq.org/
[2] Firefox 20, Core i7, 1.73GHz, 6 GB RAM
[3] https://github.com/jomaras/Firecrow

In all cases but one, the method was able to introduce a feature from one application into another. However, in order to achieve this, some modifications of the application source code were necessary. As can be seen from the Table 1 the majority of these modifications was concerned with resolving HTML naming attributes conflicts, and conflicts that arise due to overriding CSS styles. As far as JavaScript conflicts go, conflicts in the library methods were found and fixed.

The failing Case 4, related to reusing a feature from an application with the MooTools library into another application that also uses the MooTools library, has a number of problems that have occurred in the JavaScript code. The problems are related to extensions of built-in prototypes – the method was able to identify the conflicting positions, but the tool was unable to perform the necessary corrections, due to the particularities of how these extensions were implemented.

6 Related work

There are a number of approaches that support reuse: HunterGatherer [7], Internet Scrapbook [8], HTMLviewPad [9], and Web Mashups [6] in the web application domain; and G&P [3] and Jigsaw [1] in the Java domain.

HunterGatherer [7], Internet Scrapbook [8], and HTMLviewPad [9] are similar approaches mostly related to clipping and reusing fragments of Web pages. Since these approaches were developed in 1990's and early 2000, when web page development was not so dynamic on the client side, their usability in the current web development is quite limited. These approaches are mostly limited to reusing HTML element such as text fragments and forms, and make no attempts to also include CSS and JavaScript. Web mashups [6] are web applications that combine information and services from multiple sources on the web. The main advantage of mashups is that they enable the creation of new applications by integrating services offered by third-party providers. The main difference between mashups and our approach is that mashups foster reuse on a service-level, while we specifically target reuse on the code level.

In the more general domain of Java applications, G&P [3] is a reuse environment composed of two tools: Gilligan and Procrustes, that facilitates pragmatic reuse tasks. Gilligan allows the developer to investigate dependencies from a desired functionality and to construct a plan about their reuse, while Procrustes automatically extracts the relevant code from the originating system, transforms it to minimize the compilation errors and inserts it into the developer's system. In this domain there is also a tool called Jigsaw [1] which facilitates small-scale reuse of source code. The main difference between our approaches is that G&P and Jigsaw are approaches that statically analyze Java applications – while the ideas and end goals are similar, their methods can not be used in the highly dynamic, multi-paradigm environment of client-side web applications.

This work is a continuation of our previous work [5] where we have described an approach for extracting and reusing web application UI controls by combining simple analysis and profiling information. Due to not having a way of precisely

capturing dependencies between different parts of the application, the reuse process was primitive, and was more focused on extraction than on handling reuse problems. This is why in this work, by providing a detail description of possible conflicts, and by basing the process on the client-side dependency graph [4], we are able to handle conflicts that arise both statically and dynamically.

7 Conclusion

In this work we have shown how to achieve code-level feature reuse in client-side web application development. We have defined what exactly feature reuse is, and when can it be considered successful. Naturally, when reusing code from one application into another application a number of problems can occur – we have identified those problems and have developed algorithms both for their handling and for merging code bases. Finally, by testing the method on four non-trivial case-study applications, we have shown that the method is capable of identifying and handling conflicts, and performing actual reuse.

For future work, we recognize that providing user specified scenarios is time-consuming, and we plan to include a method for automatic generation of usage scenarios. We also plan to expand the reuse process to include the server-side – the client-side and the server-side are parts of the same whole, and should be treated as such. Also, for this research we have considered a particular kind of client-side feature reuse – the reuse of behavior on a structure. However, one might argue that support for global application behavior reuse should also be provided. This would require more advanced behavior analysis and code modification because a wider range of problems can occur when allowing code originating from one application to operate on the DOM of the other application.

References

1. R. Cottrell, R. Walker, and J. Denzinger. Jigsaw: a tool for the small-scale reuse of source code. In *ICSE*, pages 933–934. ACM, 2008.
2. T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.
3. R. Holmes and R. J. Walker. Semi-Automating Pragmatic Reuse Tasks. ASE, pages 481–482. IEEE Computer Society, 2008.
4. J. Maras, J. Carlson, and I. Crnkovic. Extracting client-side web application code. WWW '12, pages 819–828, New York, NY, USA, 2012. ACM.
5. J. Maras, M. Štula, and J. Carlson. Reusing web application user-interface controls. *Web Engineering*, pages 228–242, 2011.
6. S. Murugesan. Understanding web 2.0. *IT professional*, 9(4):34–41, 2007.
7. M. Schraefel, Y. Zhu, D. Modjeska, D. Wigdor, and S. Zhao. Hunter Gatherer: Interaction Support for the Creation and Management of Within-Web-Page Collections. WWW, pages 172–181, 2002.
8. A. Sugiura and Y. Koseki. Internet scrapbook: creating personalized world wide web pages. HCI, pages 343–344. ACM, 1997.
9. Y. Tanaka, K. Ito, and J. Fujima. Meme Media for Clipping and Combining Web Resources. *World Wide Web*, 9:117–142, 2006.