Understanding Web Applications Using Component Based Visual Patterns

Dan C. Cosma and Petru F. Mihancea
LOOSE Research Group
Politehnica University of Timişoara and Institute e-Austria Timişoara, Romania
Email: {dan.cosma, petru.mihancea}@cs.upt.ro

***Abstract*—This paper introduces our approach for high-level system understanding that uses software visualization to analyze the presentation layer of Web applications. The technique is driven by static analysis, relies on state-of-the art concepts, and is technology-aware, so that it focuses on those precise particularities of the application's presentation layer that define its Web presence. By combining an approach initially developed for software testing with visualization, the essential structural dependencies between and within the Web components are extracted and reviewed. Initial evaluation shows that the technique is able to provide a comprehensive view that is very useful in spotting new and interesting visual patterns that give significant insight for software comprehension.**

I. Introduction

This paper introduces our approach of reverse engineering Web applications, specifically targeting program comprehension. This new approach extends a technique developed by [1] for software testing, and combines it with software visualization in order to identify and analyze the relevant structural patterns that provide significant understanding on the application. Static analysis is used for extracting structural information about the Web components, their interdependencies, and the way they generate the Web content during the interaction with the user. The approach focuses on the application's *presentation layer* as the most significant source of information, gathering most of the elements that define the application's Web presence: technology-dependent code, user interface interaction, dynamic content.

II. Background

Web applications provide user interfaces consisting of dynamically generated HTML pages sent from the server to the Web browser. When aiming to represent the structure of the generated Web pages, one needs a way of identifying the source code fragments that deal with the dynamic generation of each significant portion of the HTML content.

For this purpose, a very useful set of concepts were introduced by [1] in the context of modeling Web applications for software testing. The main concept is the *atomic section*, defined as *"a section of HTML [...] that has the property that if part of the section is sent to the client, the entire section is"* [1]. The atomic section is in a way similar to the concept of basic blocks in programs, although it focuses on the way HTML responses are generated. A dynamically generated HTML page consists of a certain combination of atomic sections, put together at runtime by a server-side Web component. Building on this concept, [1] introduces the *Web application transition graph*, with the aim of modeling the entire application. Nodes are Web components (the *A_jsp.java* and *B_jsp.java* rectangles in Figure 1), and edges model HTML link elements and other similar transitions between the components. Each web component is represented in turn by using a *Component Interaction Model*, a graph showing how the atomic sections (the circles in Figure 1) are combined by the software to generate all the possible versions of Web pages that fall in the respective component's responsibility.

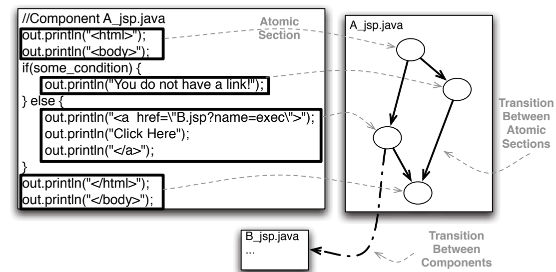

Fig. 1: Atomic Sections and Transitions

While [1] introduces this model to further define testing criteria, we use the model for an altogether different purpose: reverse engineering existing applications in order to extract highly relevant information for program comprehension.

III. Towards a Reverse Engineering Approach

Our work focuses on *i)* using static analysis to build a model of a *JSP/Servlet* application, similar to the transition graph described in Section II, and *ii)* using visualization to comprehend the system. While this is work in progress, we have already built an interactive tool to apply the approach.

A. Model Representation and Extraction

To represent the Web application, we have defined a specific meta-model shown in Figure 2. The main concept is the Web *component*, corresponding to a *JSP page*, and the other entities the atomic sections, the transitions between atomic sections and several types of transitions between components.

The model is built in two steps by processing the Abstract Syntax Tree of each Web component. The first step uses an

978-1-4673-8159-8/15 $31.00 © 2015 IEEE

DOI 10.1109/ICPC.2015.39

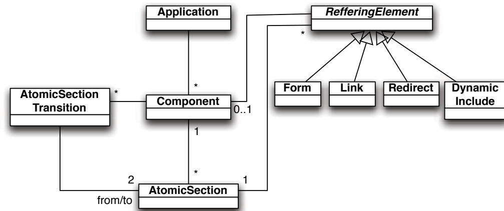

Fig. 2: The Meta-Model

inter-procedural analysis of the control flow to create the *intra-component atomic section graph*, where the nodes are atomic sections and the transitions represent compositional links between them. The second step creates the *inter-component transition graph*. It captures all elements that can cause the user interface to “jump” from the page generated by the current component to a page generated by another component (HTML links, forms and HTTP redirections), as well as the statements that let Web components include each other at runtime. The purpose is to extract the ”intentional flow” of the Web interface, *i.e.*, how are the components activated when users interact with the pages generated by each component.

B. Model Visualization

An example of the main view rendered by our visualization is presented in Figure 4, and the individual visual elements of the representation are explained in Figure 3.

Components are drawn as rectangles, linked together by color-coded arrows depicting the inter-component dependencies. There are, at this point, three types of arrows drawn at the inter-component level: i) **red arrows** – the transitions that show the ”jump” dependencies: links, form actions, and HTTP redirects. Each arrow starts in the atomic section that generates the jump, and points to the target component; ii) **blue arrows** – the runtime include dependencies. The arrow starts at the atomic section containing the runtime include statement and points to the included component; iii) **gray arrows** – dependencies that either refer to destinations outside the application, or link to components whose names are computed at runtime (and cannot be resolved by static analysis)

For each component in the view, the interior of the rectangle contains the graph that describes the component interaction model (as described in Section II). Any path from a component entry point (the smaller grey disks) to its exit point (the black bullets) represents a specific combination of atomic sections, forming one of the several possible HTML pages that can be generated by the current component.

C. Visual Patterns of Interaction

In order to present the details of our visual approach, while also providing an initial evaluation of the involved techniques, we have applied it to a real web application, *Online Bookstore*, part of the *GotoCode Applications* suite [2]. It is made of 28 Web components, totaling about 9500 lines of JSP code.

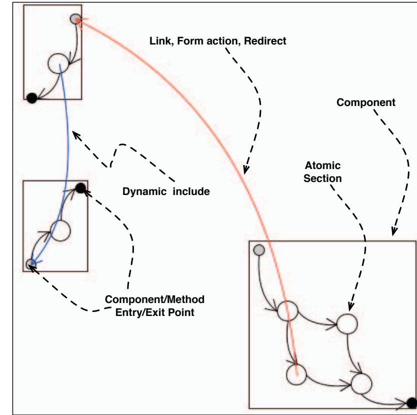

Fig. 3: The Main View Explained

The Bookstore’s presentation layer, as depicted by our visualization, is presented in Figure 4. The image is annotated to indicate the visual patterns we discuss below. A first visual clue provided by the view is that larger rectangles usually depict more important components: they contain a larger number of atomic sections, possibly connected by more complex compositional paths. While helpful, this clue is by no means sufficient by itself. There are several other interesting visual patterns we have identified, which provide a more sophisticated insight on the application’s traits relevant to system comprehension. They are presented as follows.

a) Red Target: A visual Red Target is a Web component pointed to by a lot of red arrows that originate in other components of the application (Figure 4 A). As the red arrows depict HTML links, form actions or redirects between components, this shows that a Red Target is heavily used by its counterparts, which is a strong indicator that it provides a significant, if not essential, functionality of the system.

In Bookstore we have identified one prominent Red Target component, which was easily visible. We found that its name was ”Login.jsp”, and provided an essential functionality all other main components relied on: user authentication. We must point out that the view was more than helpful in this case because, regardless of the component name, it quickly showed that Login was indeed a central component, with many other large components depending on it. If we were to only look at the code manually, we would have probably found out that a Login page was there, but there would have been no quick way to actually confirm that its was indeed used by many other important components. Moreover, interpreting the pattern from a system evolution perspective, we quickly understood that any new component adding functionality will probably have to use Login.jsp, too, which is significant insight.

b) Blue Target: A visual Blue Target is a Web component that is pointed by a lot of blue arrows originating in different components (Figure 4 B). This means the Blue Target component is in fact a JSP page that is heavily included at runtime by other JSP pages, a trait that may show that the component does

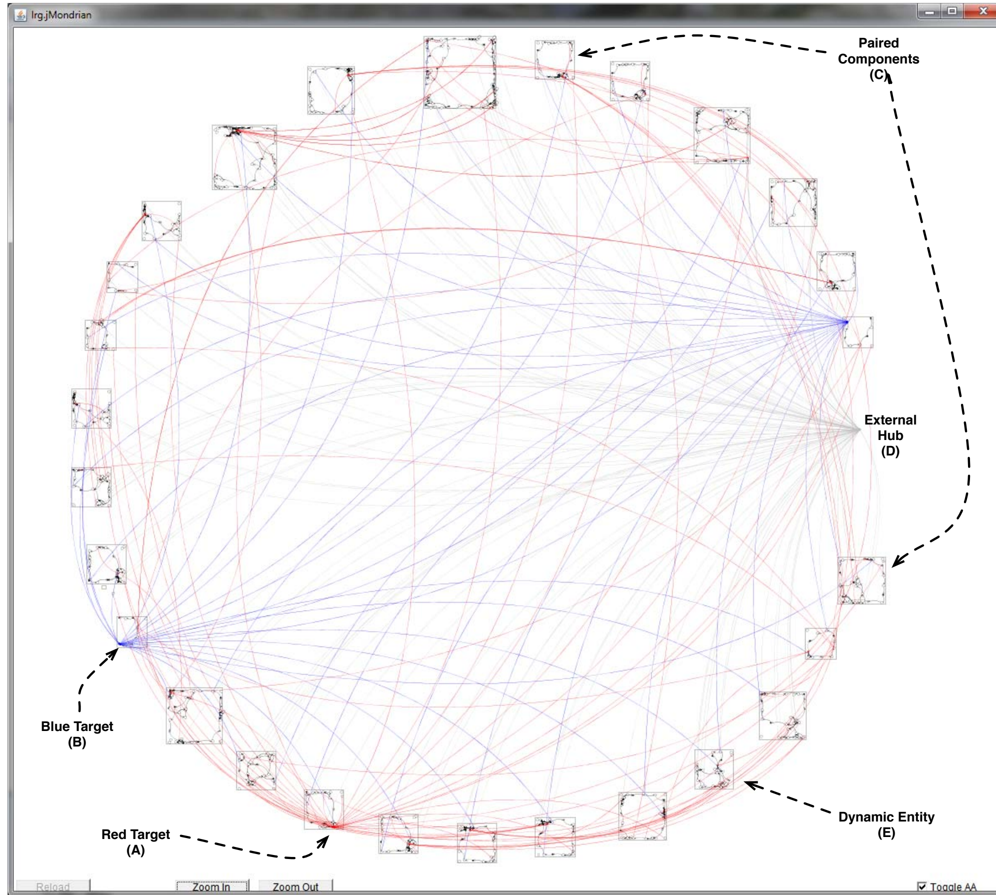

Fig. 4: The Complete View of the Case Study

not generate a complete Web page by itself, but it is instead used by others to generate parts of their own Web pages. This is useful when trying to discriminate between the main Web components and the helper JSP pages in the system. For Bookstore, we have visually identified two such components, handling the header and footer sections of the dynamic Web pages, which helped us quickly understand they are not core functional components. From the system evolution perspective, the pattern provides us again with relevant clues: any new system component (page) will probably have to include these ”blue target” components as well.

c) Paired Components: This pattern describes two components that are linked together by a significant number of red arrows. This means that the two refer each other in one or both directions through links, form actions or redirections. The pattern is significant as it shows that the user interface actions often jump from one component to another, which suggests that the two work together for a common goal. Visually, the pattern is identified by finding ”streams” of arrows that only connect the two entities, standing out among the other inter-component transitions. In Bookstore, we have found several such pairs, one of them being (MembersRecords, MembersGrid) – Figure 4 C. An analysis of their code showed that they indeed worked together, enabling the user to browse a list of registered users (”members”). MembersRecord handled the editing of user information, while MembersGrid displayed the member list by generating HTML tables.

d) External Hub: This is not a pattern *per se*, but a rather useful particularity of the visualization. Our visualization gathers all inter-component transitions that point to external or unresolved destinations (the gray arrows) in a single node (visible in Figure 4 D on the right). It can be used as a starting point for analysis in at least two cases: *i)* to search for the components that refer external entities, such as other parts of the same Web portal, external sites, entities belonging to the Web development frameworks or tools used, etc.; *ii)* to help locate the *dynamic* parts of the code, *i.e.*, those that use runtime constructs to compute jumps to other components: they are identifiable by looking at the atomic sections with outgoing gray arrows not labeled with (statically-resolved) URLs.

In Bookstore, about 35% of the External Hub links could be resolved statically and provided technology-related clues: they pointed to the site of a software tool used when developing the application. The rest were references computed at runtime.

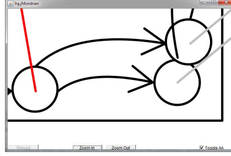

Fig. 5: Conditional Jump

e) Dynamic Entity: This pattern highlights the most prominent components related to the External Hub: those that originate a visibly large number of outgoing gray arrows (Figure 4 E). They either depend strongly on external entities, or they heavily rely on runtime values for computing the links. In Bookstore most such components were using runtime database values for linking to other pages.

f) Conditional Jump: Unlike the above patterns, this is an *intra*-component visual pattern, visible when looking inside a component, at the graph connecting atomic sections. It depicts two intra-component edges starting in a same atomic section, and pointing to at least one atomic section that is the origin of an *inter*-component arrow. For example, this happens when the code contains an *if* statement, and only one of its branches generates a HTML link to another component. Finding such cases is important, because they show an inter-component reference which only activates in specific cases, and identifying the condition will provide essential insight on the code. We found several such cases in Bookstore, one of them in the EditorialGrid component (Figure 5). One of its atomic sections was connected to two atomic sections originating gray arrows. We found that the corresponding code controlled the behavior of a button for navigating in a row of inter-related pages. When at the first page the button simply pointed to it, while for the rest the EditorialGrid was called again, with a different page value as parameter. Helped by the visualization, the process of understanding this case only took us 5 minutes, including the parts where we read the code.

IV. Related Work

Analyzing Web applications is a constant concern for the software engineering community. Aiming software testing, Offutt and Wu [1] introduce the *atomic section model* to represent Web applications by focusing on the HTML output generated by their components. We use this idea in our approach, adapting it to our goal of reverse engineering the source code to achieve program comprehension.

There are various efforts of reverse engineering Web applications. Early achievements targeted the transformation of legacy static Web sites into dynamic applications. A reference example is the work done by Ricca and Tonella [3], where similarity measures are used to group pages together, as candidates for migration into dynamic components. Full-fledged dynamic Web applications were addressed by Di Lucca et al. [4], [5], who developed a tool-supported reverse engineering process for understanding Web applications. While also targeting system comprehension, their approach differ from ours as their aim is the recovering of UML diagrams for the system, it uses a hybrid static-dynamic process, and it is not driven by visualization. Amalfitano et al. [6] describe what they call an 'agile' reverse engineering process for Web applications, that uses dynamic analysis to model the application's presentation layer. Besides using static analysis, our approach is different as it aims at finding structural patterns that can be directly used for quick yet relevant assessments in an source-code driven interactive environment. Techniques that reverse engineer Web applications can be entirely based on dynamic analysis, rather than static as our own. An example is Revangie [7], where crawling and HTTP communication monitoring are the employed techniques. An interesting way of analyzing Web applications is also presented in [8], where the authors use instrumentation to dynamically capture interaction between the application and its database. Rather than applying the analysis on the presentation layer, it finds the SQL-related behavior in PHP-based applications to assess security aspects.

V. Conclusions

This paper has presented our ongoing work in reverse engineering the presentation layer of JSP/Servlets Web applications by combining static analysis and visualization to achieve software comprehension. The identified visual patterns describe several essential interactions that help us easily find system traits relevant for its understanding, as shown by our initial evaluation in a real-world case study.

Future work aims the development of a full reverse engineering process for comprehending modern Web applications. The first steps imply refining the visual patterns, finding new ones, and validating them on a larger number of applications. Next, the process will have to use them in *strategies* targeted to assess the various high-level concerns implied by the goal of software understanding.

Acknowledgment

This work has been partially supported by the European FP7-ICT- 2009-5 project no. 257876, SPaCIoS Secure Provision and Consumption in the Internet of Services.

References

[1] J. Offutt and Y. Wu, "Modeling presentation layers of web applications for testing," *Software&Systems Modeling*, vol. 9, no. 2, pp. 257–280, 2010.

[2] Yes Software, "Gotocode applications," http://web.archive.org/web/20110430192101/http://gotocode.com/.

[3] F. Ricca and P. Tonella, "Using clustering to support the migration from static to dynamic web pages," in *IWPC '03*. IEEE CS Press, 2003.

[4] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "Ware: a tool for the reverse engineering of web applications," in *CSMR 2002*.

[5] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Reverse engineering web applications: the ware approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 71–101, 2004.

[6] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "An iterative approach for the reverse engineering of rich internet application user interfaces," in *ICIW 2010*.

[7] D. Draheim, C. Lutteroth, and G. Weber, "A source code independent reverse engineering tool for dynamic web sites," in *CSMR 2011*.

[8] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Wafa: Fine-grained dynamic analysis of web applications," in *WSE 2009*.