Columbus: A Tool for Discovering User Interface Models in Component-based Web Applications

Adrian Hernandez-Mendez, Andreas Tielitz and Forian Matthes
Software Engineering for Business Information Systems, Department of Informatics, Technische Universität München, München, Germany

Keywords: Component-based Web Applications, Model-based User Interfaces, UIML, Model Discovering.

Abstract: The processes of replacing, maintaining or adapting the existing User Interfaces in Component-based Web Applications to new conditions requires a significative amount of efforts and resources for coordinating their different stakeholders. Additionally, there are many design alternatives, which can vary according to the context of use. Therefore, understanding the structure and composition of UIs and their contained elements can provide valuable insights for future adaptations. In this paper, we present a tool for discovering UI models in the source code of Component-based Web Applications, which could be used to support the reverse engineering process. Subsequently, we evaluated its capabilities of User Interface model extractions using open-source project TodoMVC. The evaluation process shows the main limitations of the JavaScript frameworks for creating an abstract UI model (i.e. technology independent model) for Web Applications.

1 INTRODUCTION

Web Applications have become ubiquitous. They are an essential element of any modern service, and examples are found in online shops or booking systems. Therefore, different users and stakeholders demand continuous improvements of them that can be adapted to the natural changes in Web technologies. However, these demands put the developers to face diverse challenges during the development and maintenance of Web Applications.

One of the modern architectures used in the development of Web Applications is the Single-Page Applications (SPA). This architecture divides Web Applications into two main components: User Interface (UI) and Server component. The Server component which remains constant across multiple environments is most supported and understood due to the extensive research in the field of service-oriented architecture. However, there is no dominant technology for creating the UI component, contrariwise there are many alternatives, which can vary according to the context of use. Therefore, understanding the structure and composition of UIs and their contained elements can provide valuable insights for future adaptations.

In this paper, we propose a tool, called Columbus, for discovering UI models in Component-based Web Applications. The design and development of Columbus is based on the first iteration of the Hevner's three cycle view of the design science research framework (Hevner, 2007). The three cycles within this research framework correspond to:

- **Relevance cycle**: We establish the need for analyzing the most popular open-source JavaScript Libraries in Github (e.g. AngularJS, PolymerJS, and ReactJS).
- **Design cycle**: We present the platform and its architecture design process as a research artifact.
- **Rigor cycle**: We evaluate the capabilities of the existing UI modeling languages to describe Web User Interfaces, and establish a small contribution of our research findings to the model-based User Interface knowledge base.

The remainder of this paper is organized as follows: In Section 2 we discuss the use of declarative UI models and describe UI Modelling Language (UIML). In Section 3, the model discovered process is presented. The architecture of Columbus is discussed in Section 4 and it is evaluated in Section 5. In Section 6, we present the related work and finally conclude with Section 7.

Hernandez-Mendez, A., Tielitz, A. and Matthes, F.
Columbus: A Tool for Discovering User Interface Models in Component-basedWeb Applications.
DOI: 10.5220/0006307803240331
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 324-331
ISBN: 978-989-758-246-2

2 USER INTERFACE MODELS

Declarative UI models were introduced to reduce repetition in developing User Interfaces. By explicitly defining the UI's information in an abstract model, concrete UI implementations can be generated. Over the time, many different declarative UI modeling formalisms have emerged for various purposes. For example, companies like Microsoft introduced the language Extensible Application Markup Language (XAML[1]) and Oracle the XML-based User Interface markup language (FXML[2]). These languages allow the developers to model the static parts (e.g. structure) of the UI and complementing the development of the dynamic components using C# and Java, respectively. Additionally, in the academic and standardization environment formalisms such the User Interface Modelling Language (UIML)(OASIS, 2008), the User Interface eXtensible Markup Language (UsiXML)(Limbourg et al., 2004) and the Web Modelling Language (WebML) (Ceri et al., 2000). Unlike the formalisms created at the industry context, the formers allow modeling of UI independently from any technology or platform. These models themselves rely on predefined sets of elements which can be displayed, connected, and reused in whichever way is necessary for the desired UI. Different groups of elements can also be split up into distinctive sets with similar functionalities or goals. We choose UIML as a reference model for discovering the UI models in the existing Component-based Web Applications.

2.1 UIML

The UIML describes any User Interface using a canonical User Interface model (Abrams et al., 1999). The composition of the User Interface is defined as a collection of interface elements which can be used independently in different environments, including the web technologies.

UIML establishes a clear separation of concerns in the UI models, with four sections as shown in Figure 1.

The structure section specifies how the template of the interface looks and which parts are visible at a given moment. The child elements of a structure are composed of parts which represent individual elements or can contain nested interfaces.

The content holds multiple versions of the displayed information in different languages. This section is used for internationalization and separation of language specific information.

[1]https://tinyurl.com/7wzsufu

[2]https://tinyurl.com/j4cnysx

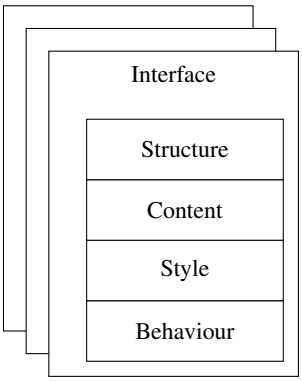

Figure 1: Separation of concerns in UIML.

The style contains a list of all properties of the interface. It holds information regarding the styling of the template, the association of content elements to their respective parts, or variables that are shared with other interfaces.

The behavior describes the user interactions with the particular interface. Interactions like button clicks or form submits are declared here. Additionally, the behavior also contains entries for internal method invocations that occur in the User Interface.

2.2 Simplification of UIML

The focus of UIML lies on supporting and describing a wide variety of interfaces. Reused elements are redefined – rather than referenced – for each occurrence. Due to those limitations, a more streamlined and simplified adaptation of the standard was required to properly express modern web-based User Interfaces. Support for representations of non-browser based visualizations was omitted from the model.

Instead of referring to interfaces, the more fitting term component was used instead, which is described by a structure, behavior, content and style section. The parts in the structure of UIML were supplemented with a Reference entity, which can point to another component of the User Interface.

Properties are used to model the content in the User Interface, as well as all attributes of the JavaScript components. The content is ignored in this case because the model discovery process cannot distinguish between different language sets.

The behavior remains unchanged and still contains all user interactions. However, the life cycle methods of a JavaScript components are part of the model and they are described in the behavior.

3 MODEL DISCOVERY PROCESS

The proposed model discovery process in Columbus converts the source code of a web application into a corresponding view model. The process itself is divided into three steps: Semantic analysis, information extraction, and model generation. This division is based on a fundamental architecture of reverse engineering tools (Chikofsky and Cross, 1990). The in- and outputs of each step are shown in Figure 2.

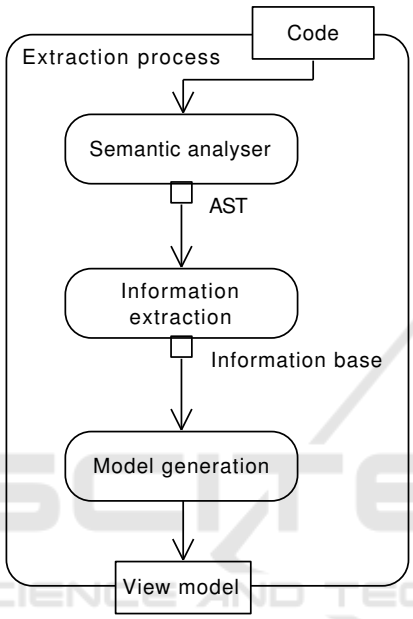

Figure 2: Extraction process implemented in Columbus.

The application source code is automatically retrieved from a GitHub repository. The optional commit hash and a regular expression can be supplied to filter the retrieved source code.

3.1 Semantic Analyser

During the semantic analysis, a syntactical parser is applied to the supplied source code and transforms it into an abstract syntax tree. Relevant information for the view model generation is encoded in the tree nodes and can be accessed in a structured way.

In addition to the syntactical parser, pre- and post-processing steps, as shown in Figure 3, allow alterations to the in- and output.

As part of the pre-processing step, template expressions, like JSX, are translated into JavaScript code before the syntactical analyzer processes the input.

Differences in declarations of JavaScript source code are unified by using a JavaScript compiler. Different syntax constructs with the same semantics are enriched with additional information to align the possible input scope.

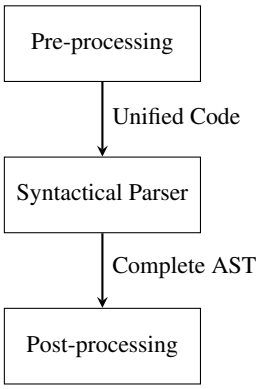

Figure 3: Workflow of the semantic analyser with in- and outputs.

3.2 Information Extraction

The information extraction receives the abstract syntax tree (AST) generated in the previous step as an input. The step is designed as a rule-based system as shown in Figure 4. Each rule is specialized in independently extracting a certain piece of information from the AST and storing any retrieved information under a unique identifier in the information base.

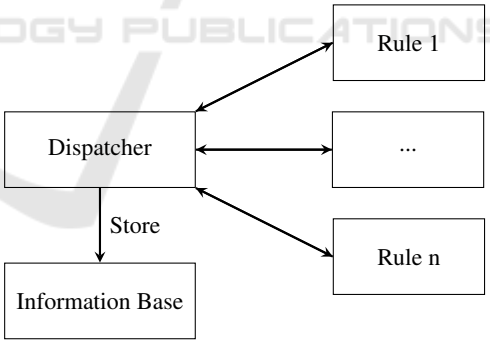

Figure 4: Rule-based system for information extraction.

A typical rule initially queries the abstract syntax tree for a certain pattern. The pattern corresponds to a certain structure in the tree, and the query returns the parent nodes of the matches. The contained information in the returned subtrees can be extracted either by further elimination or flattening of the nodes and their children.

3.3 Model Generation

The model generation operates on the aggregated set of information in the information base. The view

model is constructed by selectively retrieving and processing one or more sets of information and mapping them to the entities in the model.

4 ARCHITECTURE

Columbus is an AngularJS web application which implements the previously described extraction process. The implementation of the process is shown in Figure 5. The content of the GitHub repository is retrieved, and the first two steps of the extraction process are performed on each file. During each iteration, additional information is added to the information base. The model generation uses the aggregated information to create the model.

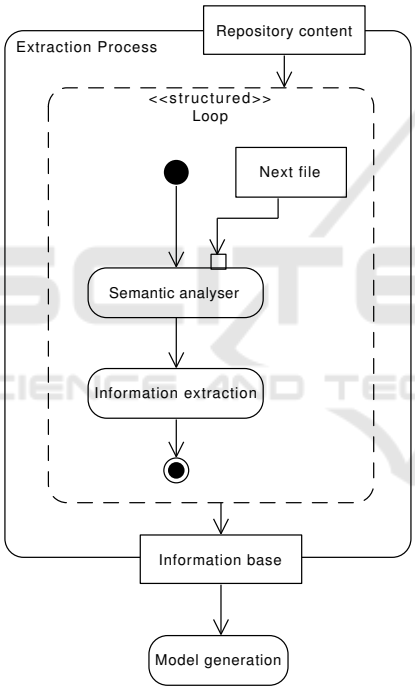

Figure 5: Extraction process implemented in Columbus.

The architecture of Columbus is centred around the main controller *AppCtrl* as shown in Figure 6. The controller directs the program flow and is responsible for displaying the output generated by each extraction step in the user interface. Responsibilities are divided up into the respective parts of the extraction process.

The semantic analyser classes translate the fetched source code from GitHub into an abstract syntax tree. Each subclass of *Ast* implements helper methods to provide easier access to framework specific constructs. The library ESQuery is used to extract information from the tree.

The information extraction is implemented as a rule-based system with each rule extending the *AbstractExtractor*. Every rule must be registered in the *SharedModelExtractor*.

Model generation is implemented in the *AbstractModelGenerator*, which stores information from the information base in the corresponding *ComponentModel*.

The application uses the external libraries Esprima[3] and Babel[4] to transform and compile the source code. The library ESQuery[5] is used to query the abstract syntax tree generated by Esprima.

5 EVALUATION

The TodoMVC project[6] provides an example implementations for a simple ToDo manager. The project contains a TodoMVC application, which is implemented in multiple JavaScript frameworks and libraries. We evaluated Columbus by comparing the model extraction process with available implementations of the TodoMVC for React, AngularJS, and Polymer.

The TodoMVC is a Web Application which can be used to manage tasks. The tasks are stored in a remote database or the local storage of the browser. The architecture of TodoMVC is composed of multiple components. The components used to build the interface differ between each implementation, but they all share the design aspect of having two main components: *TodoApp* and *TodoItem*, shown in Figure 7, although their naming differs between implementations. However, the HTML structure is similar the selected frameworks.

Initially, we analyzed each implementation and refactored the existing code. The implementation for React[7] could be used without any alterations since the source files contain pure JavaScript in conjunction with JSX template syntax. The Polymer[8] and AngularJS[9] implementations had to be stripped of all non-JavaScript content first. The AngularJS implementation was additionally refactored and migrated to a component-based web application.

The evaluation of Columbus follows with the creation of two UI models. The first model was generated manually and the second, using the tool itself.

[3]http://esprima.org/
[4]https://babeljs.io/
[5]https://github.com/estools/esquery
[6]http://todomvc.com/
[7]https://git.io/vykNG
[8]https://git.io/vykNZ
[9]https://git.io/vykNc

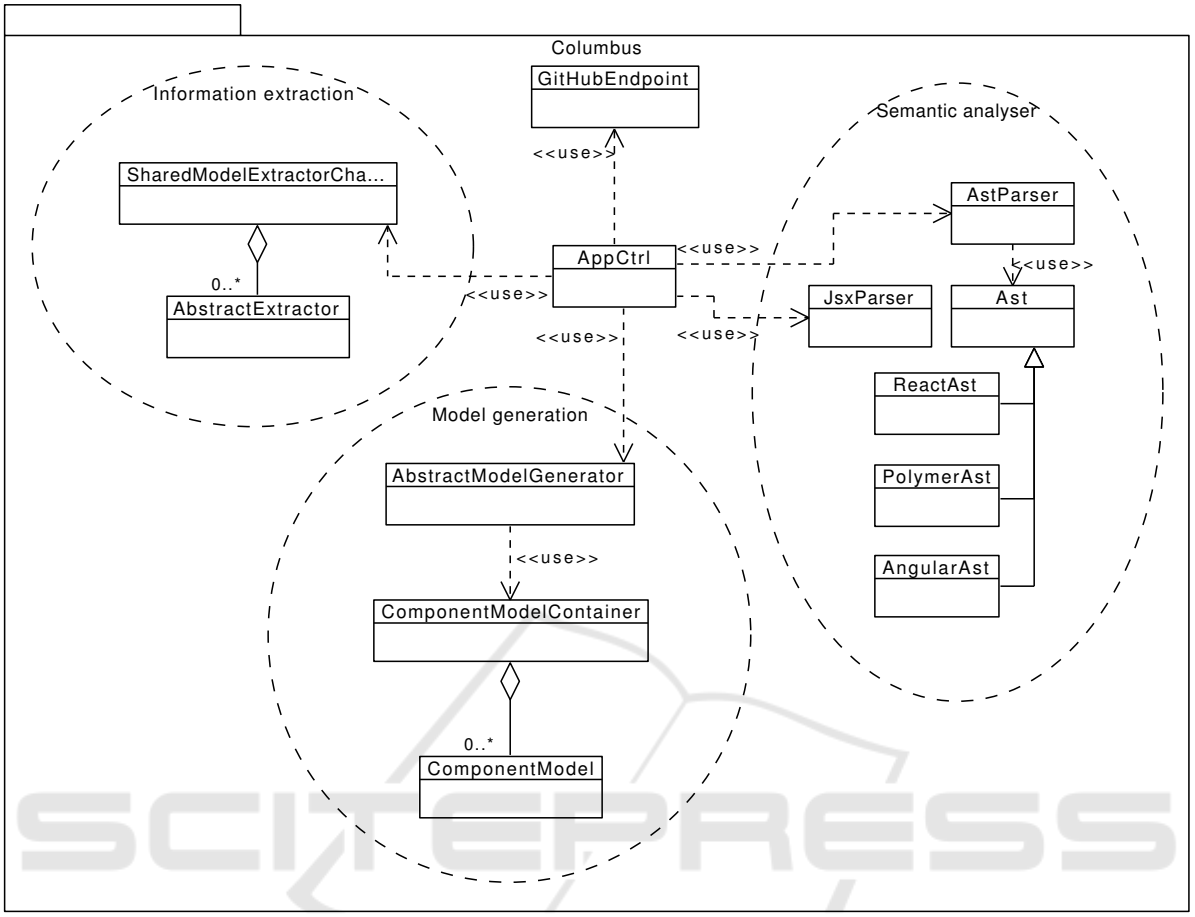

Figure 6: Architecture of Columbus.

Figure 7: Component composition of TodoMVCs user interface.

Both models were compared and their deviations regarding structure, style and behavior were analyzed in each implementation. The results are summarized in the Table 1 and described in details in the following subsections.

Table 1: Evaluation of correctly extracted entities of the TodoMVC application with Columbus.

	React	Polymer	Angular
Structure	84%	0%	0%
Style	88%	55%	50%
Behaviour	100%	11%	0%
Life cycle	100%	100%	100%

5.1 AngularJS

Without the possibility to analyze the template, the extraction process could not extract most of the properties and any behavioral rules. AngularJS relies heavily on annotations on template tags to define custom event listeners who cannot be processed by Columbus.

5.1.1 Limitations Identified in the Structure Element

Unable to Process the Template.
The proprietary template syntax of AngularJS is not

supported by Columbus, which means that all information from the template was unavailable to the extraction process.

5.1.2 Limitations Identified in the Style Element

Missing Properties.
Without the template, any property defined in the DOM is inevitably lost to the view model generation. This is critical to ordinary text fragments in HTML elements since these are not defined anywhere inside the component definition.

Property Usage without Declaration.
Columbus requires the component to list all properties in either the bindings section or as a variable declaration in the constructor function. If properties are used without declaration, Columbus attempts to detect them by tracking this context in conjunction with variable usage, but it cannot guarantee that all variables are detected.

Function Properties.
Some of the properties are functions which are passed down from parent components. Currently, the view model does not differentiate between a property which is an attribute or a function. It is necessary to alter the model to correctly reflect the different properties.

5.1.3 Limitations Identified in the Behavior Element

Rules from Missing Parts.
Without the template, any behavioral rules defined in the DOM are inevitably lost.

5.2 ReactJS

The possibility to analyze the template in React resulted in a high extraction rate of properties and behavioral rules. The missing and incorrect entries are mostly due to the misinterpreted part constructs which are not supported by Columbus.

5.2.1 Limitations Identified in the Structure Element

Multiple Return Statements in Render Function.
Columbus uses static code analysis to extract information and is not capable of determining which return statement should be evaluated. Due to this limitation, the last return statement of the render function is used per default. All template parts that are not declared in the last return statement are not part of the view model.

Building the DOM Piece by Piece with Variables.
If part of the DOM is built beforehand and printed out as a variable in the final return statement, Columbus cannot extract the information contained in the variable. The complete DOM must be constructed inside the last return statement of the render function.

Misinterpreted References / Missing Parts.
Columbus traverses through the template structure and tries to identify the nodes visited. Depending on the type of node, different subsequent actions are taken, like differentiating between an HTML element and a component or determining if the current node contains any children. If the tool encounters an unexpected construct, like a variable output which contains further template tags, the traversal can not determine how to proceed. The fallback routine for these cases replaces unknown constructs with an empty part entity. No further traversal of children is possible for these misinterpreted nodes.

5.2.2 Limitations Identified in the Style Element

Property Usage without Declaration.
Columbus requires the component to list all properties in the propTypes section. If a property is used without a preceding declaration, Columbus attempts to detect it by tracking this context in conjunction with variable usage. Special objects like this.props and this.state simplify the search process to some degree. Variables that start with one of these constructs can always be treated as component properties. The properties detection is currently limited to the render statement which means that some properties might not be detected.

Properties from Missing Parts.
If a property was never used in the JavaScript part of the component, the detection is limited to all parts of the template detected during the corresponding extraction process. If a part was not correctly processed by the template extraction rule, any property is inevitably lost, and therefore unavailable during the view model generation.

Detecting Default Values.
Columbus can only detect simple default values as long as they are declared in the propTypes and getDefaultProps section of the component. More sophisticated values, like the output of a function or assignments of global variables, are not detected. In these cases, the default value of the property remained blank.

5.2.3 Limitations Identified in the Behavior Element

Rules from Missing Parts.
Columbus is only capable of extracting behavior rules which are either declared in the JavaScript configuration or the template. The extraction of the behavior is not limited to a specific return statement in the render function, but instead, tries to capture all usages. If the corresponding source part was not extracted due to the limitations listed before, the source id of the behavior event does not point to a valid part of the view model. Nonetheless, each rule contains a unique part id which hints at an existing behavior.

5.3 PolymerJS

Without the possibility to analyze the template, the extraction process could not extract most of the properties and behavioral rules. Only two of the behavior rules were defined according to the conventions for Polymer application required by Columbus.

5.3.1 Limitations Identified in the Structure Element

Unable to Parse the Template.
The proprietary template syntax of Polymer is not supported by Columbus, which means that all information from the template was unavailable to the extraction process.

5.3.2 Limitations Identified in the Style Element

Missing Properties.
Without the template, any property defined in the DOM is inevitably lost to the view model generation. This is critical to ordinary text fragments in HTML elements since these are not defined anywhere inside the component definition.

Property Usage without Declaration.
Columbus requires the component to list all properties in the properties section. If properties are used without declaration, Columbus attempts to detect them by tracking this context in conjunction with a variable usage, but it cannot guarantee that all variables are detected.

5.3.3 Limitations Identified in the Behavior Element

Rules from Missing Parts.
Without the template, any behavioral rules defined in the DOM are inevitably lost. The requirements for view model extraction of Polymer projects requires the usage of the listener's section to define event listeners.

6 RELATED WORK

Considerable research has been conducted in the domain of reverse engineering of Web Applications. However, the permanent changes in the technologies, frameworks, tools and architectures in their domain makes very challenging the comparison process of Columbus with existing tools. However, we describe selective relevant research in the domain which can be either source of evaluation or inspiration for Columbus.

Di Lucca et al. introduced their WARE tool which allows users to visualize user interfaces in conjunction with the frontend and backend architecture (Di Lucca et al., 2002). However, the details of the extraction process are not described deathly, yet they state that the source code is transformed into an abstract representation. Their tool is capable of extracting the user interface of a web application and provide visualizations for structure and page flow. The reverse engineering was focused on providing post-development documentation in the form of UML diagrams. We encounter the main differences with our approach in the UI model definition, which makes challenging the incorporation of all functionality that can be found in modern JavaScript frameworks.

Laurent Bouillon compares different reverse engineering tools and also introduces approaches to reverse engineer HTML documents (Bouillon, 2006). He proposes a translation of HTML elements into an extensible user interface markup model to gather an abstract representation of the user interface. The tools listed in the dissertation use static reverse engineering to translate HTML files into the defined markup language. The process itself shares certain similarities, like transforming the input into a more accessible data structure, like a tree or in that case, as an XML document. Rules are used to extract relevant information from a knowledge base. The research itself is limited to plain HTML documents, without any JavaScript integration. Columbus' focus lies on reverse engineering JavaScript components, which most of the time contain templates with HTML.

Morgado et al. elaborate on different reverse engineering approaches to reduce the necessity of creating view models for testing purposes. They discuss rationales for static and dynamic reverse engineering and introduce their tool called ReGUI (Morgado et al., 2011). However, the approach using in

this paper differs in the targeted environment and the used process of extracting information. ReGUI uses dynamic reverse engineering with try-and-error interactions to obtain a graph of the different windows of a user interface. Columbus tries to create a view model of JavaScript components and their templates through static reverse engineering. This work is extended with the use of Machine Learning tools in (Morgado et al., 2012), which is a relevant aspect to consider in further research on using Columbus.

An interesting framework is proposed by Carlos Eduardo Silva in his Ph.D. thesis (e Marques da Silva, 2015), which is capable of analyzing existing Web Applications and generate the structure and behavior models from them. This approach differs from the approach used in Columbus because the inspections are realized on the running applications instead to the source code of the application.

7 CONCLUSIONS

In this paper, we present a tool for discovering UI models in the source code of Component-based Web Applications, which could be used to support the reverse engineering process. Subsequently, we evaluated its capabilities of User Interface model extractions using open-source project TodoMVC. The evaluation process shows the main limitations of the tools for creating an abstract UI model from the modern JavaScript frameworks (i.e. technology independent model) in Web Applications.

We plan to extend in the Columbus tool: first, adding new rules to support other web technologies to evaluate the complexity of the integration of different presentation techniques. Additionally, we continue researching on the model-based UI formalisms to facilitate the integration of the UI components and evaluate the possible combinations of the current process with Machine Learning techniques.

REFERENCES

Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M., and Shuster, J. E. (1999). Uiml: an appliance-independent xml user interface language. *Computer Networks*, 31(11):1695–1708.

Bouillon, L. (2006). *Reverse Engineering of Declarative User Interfaces*. PhD thesis, Université de Valenciennes et du Hainaut-Cambrésis.

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1):137–157.

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17.

Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P., and De Carlini, U. (2002). Ware: a tool for the reverse engineering of web applications. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, pages 241–250. IEEE.

e Marques da Silva, C. E. B. (2015). *Reverse engineering of web applications*. PhD thesis, Universidade do Minho.

Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M., and Trevisan, D. (2004). Usixml: A user interface description language for context-sensitive user interfaces. In *In Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages"*, pages 55–62. Press.

Morgado, I. C., Paiva, A., and Faria, J. P. (2011). Reverse engineering of graphical user interfaces. In *The Sixth International Conference on Software Engineering Advances, ICSEA*, pages 293–298.

Morgado, I. C., Paiva, A. C. R., Faria, J. P., and Camacho, R. (2012). GUI reverse engineering with machine learning. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 27–31, Zurich, Switzerland. IEEE.

OASIS (2008). User Interface Markup Language (UIML) Version 4.0. Committee draft.