Automatic Mapping of Business Web Applications

Adrian Sterca[a], Virginia Niculescu[b], Alexandru Kiraly and Darius Bufnea[c]
Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Keywords: RPA, Web automation, Web Application Mapping, DOM, CRUD Operations.

Abstract: We present an automated tool that can be used to construct conceptual maps of business web applications. This conceptual map depicts in an abstract, hierarchical way the possible navigation and operational paths in the UI (i.e. User Interface) of the web application. Our tool discovers this conceptual map by navigating automatically through the UI of the target business web application and by mapping UI operations to conceptual operations in a database. The output product of our tool is this conceptual map represented in a graphical or serialized form. This conceptual map can be used for documenting a business web application so that new users of the web application quickly gain the necessary knowledge to navigate through the UI screens of the application and to operate the application. It can also be used for developing RPA (i.e. Robotic Process Automation) solutions that automate process execution on that business web application. This tool comes in the form of a browser extension.

1 INTRODUCTION

We consider in this paper the idea of mapping a business web application. That is to create a conceptual map for all the paths in the UI of a business web application. In the initial days of the world wide web, static websites used to have a tree-like map of the website, describing all of the website's static Html documents. Current business web application don't have such maps because most of the Html content is generated dynamically from a database. And so, it would be difficult to create such maps due to the possible infinite amount of generated Html content.

Our study targets business web applications like CRM (i.e. Customer Relation Management), ERP (i.e. Enterprise Resource Planning) or project management web applications like Microsoft Dynamics, Atlassian Jira, but generally, any business web application that uses a relational database in the backend and exposes the data in this database through the functionality of the UI (i.e. User Interface).

If we consider all the Html content generated by a business web application, we can see that there are two types of Html content in a business web application:

- Html content used purely for the visual look of the application and for navigation (e.g. menus and menu items, tabs and panels, headers and footers, cover images etc.); this Html content does not use, expose or manipulate the data in the underlying database of the application
- Html content that represents an interface to the data in the underlying database; this Html content is used for showing various entities from the database or for allowing the user to add new content or modify existing content from the database; most Html content generated by a business web application belongs to this second type.

In many ways, we can consider a business web application to be just a user-friendly interface to the data in a database. Most UI operations of such an application can be abstracted/reduced to operations on the database (i.e. most UI operations translate to read or write operations in the database or in SQL language, they translate to Select, Insert, Update or Delete operations of various entities in the database) - these are the business functionalities of the application.

We introduce a tool that is able to navigate through the screens/paths of the target web application and is able to detect the conceptual operations facilitated by the Html content of the application. And it does all of this automatically, without the intervention of the human user. The tool handles all the understanding of the UI and the mapping of UI controls to various properties of entities from the backend database of the

[a] https://orcid.org/0000-0002-5911-0269
[b] https://orcid.org/0000-0002-9981-0139
[c] https://orcid.org/0000-0003-0935-3243

Sterca, A., Niculescu, V., Kiraly, A. and Bufnea, D.
Automatic Mapping of Business Web Applications.
DOI: 10.5220/0012212700003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 437-445
ISBN: 978-989-758-672-9; ISSN: 2184-3252

application. In this sense, our mapping tool automatically translates human user operations on the UI onto conceptual operations in the database. So the central idea of the tool is to map UI operations on conceptual operations in the database.

The tool comes in the form of a browser extension. The output of running our tool on a target business web application is a conceptual map which consists of nodes inter-linked in a tree-like structure, these nodes representing either *generic Html content* or *Html content representing conceptual operations*. A *generic Html content* is just an Html document that contains menus, documentation or any other generic info about the application and does not expose any entity from the database. While an *Html content representing conceptual operations* is an Html code like text inputs, buttons, paragraphs, tables either for displaying properties of an entity from the database (i.e. 'SELECT' the entity) or for allowing the user to add or remove an entity from the database or to alter its properties (i.e. 'INSERT' or 'UPDATE' or 'DELETE' the entity). This conceptual map is produced in a serialized form as JSON and depicted graphically in Html canvas.

Such a conceptual map of a business web application can have many usages. We mention two of them. Such a conceptual map can be useful as documentation for new users, for allowing new users of the application to see an abstract, birds-eye view of the web application. This can facilitate an easy learning of the functionality of the business application by new users. Such a map can also be useful for advanced RPA (i.e. Robotic Process Automation) robots. Instead of an RPA developer coding various operations on UI controls, an RPA agent can be guided to execute various conceptual operations on the conceptual map. The browser extension tool automatically maps UI controls to properties of entities in the database, i.e. it 'understands' the UI of the application, so it can also be programmed to manipulate those UI controls (i.e. adding a value to a text input control in the UI, clicking a submit button in the UI etc.) in order to perform the actual conceptual operation from the map (e.g. inserting an 'Account' entity, updating a 'Contact' entity, deleting a 'Resource' entity etc.).

In this paper, the term *concept* refers to the data type stored in a database table, while *entity* refers to a row/record of a database table. A *concept* always describes a set of *entities*; e.g. an 'Account concept' is stored in an 'Account' table in the database, while each entry/record from this table represents an 'Account entity'. In order to avoid a node explosion problem in the generated conceptual map, for all entities of a specific database concept and each conceptual operation (i.e. Select, Insert, Update, Delete) we only store one node in the map; i.e. we don't have one map node for each entity, only a single map node for the whole concept. For example, if we have an 'Account' table in the database and in this table there are three entries, 'Account1', 'Account2' and 'Account3', we only store one single node in our conceptual map for the conceptual operation of 'Update Account'; we don't have three different map nodes for 'Update Account1', 'Update Account2' and 'Update Account3'. This is true for the other conceptual operations: Select, Insert and Delete.

The rest of the paper is structured as follows. Section 2 presents studies related to our work. Section 3 presents the structure of the conceptual map, while Section 4 details the algorithm used for building the conceptual map. Following, in Section 5 we discuss the method used for detecting conceptual operations in the DOM. Section 6 is devoted to tests and experiments and Section 7 discusses some limitations of our mapping tool. The paper ends with conclusions and future work.

2 RELATED WORK

A related topic is UI screen parsing(Wu et al., 2021) where a screenshot of a UI is analyzed in order to determine the components it's made from: buttons, menus, labels etc. These components are represented in an hierarchical structure. Screen2Vec (Li et al., 2021) uses machine learning autoencoders to represent the layout and UI components of a mobile application screen in a vector of numbers. These studies are performed on a single screen of an applications and consider only mobile applications, not web applications. Our study considers all the screens of a business web application.

(Feiz et al., 2022) uses transformer models Faster RCNN to determine the similarity of various screens of a mobile application. Another approach for determining the similarity between screens of web applications, not only mobile, is taken in (Wu et al., 2023). Authors construct a dataset of web UI screenshots, WebUI and then they use machine learning models with transfer learning to perform screen element detection, screen similarity and screen classification on these screenshots. Similar approaches are described in (Nguyen and Csallner, 2015) where authors try to detect UI elements (i.e. buttons, text inputs etc.) from a screenshot of a mobile application in order to generate de UI code for that specific UI.

Another related topic is that of web segmentation which refers to dividing a web page (including the

UI of a web application) into constituent segments like: menu items, paragraphs, tables, panels, cards etc. Web segmentation can be useful for information retrieval, archiving, topic extraction, focused crawling, and improving the accessibility of web content in non-visual environments. Generally, there are three approaches for web segmentation: a) a pixel-based one that uses computer vision techniques to determine segments from a screenshot of a web page, b) a DOM-based approach which uses features extracted from the DOM hierarchical structure of the web page, and c) a hybrid approach which combines the previous two.

The VIPS (VIsion-based Page Segmentation) algorithm for segmenting a web document is presented in (Cai et al., 2003). The starting point of the VIPS algorithm is separator tags. VIPS uses separator tags in the web document like
and <hr>in order to identify segments. Then it uses heuristics based on the <table>, <tbody>, <tr>, <td>, <p>, and tags in order to further divide the Html content into segments. Another paper, (Mantratzis and Cassidy, 2005), introduces a method for recognizing significant structures, such as table-like or list-like structures within a web page. This approach operates at different levels within the DOM tree. Initially, the DOM tree undergoes a cleanup process where unimportant tags are removed, while empirically determined important tags, such as <table>, <tr>, <td>, <ul>, or , are retained. Then, the algorithm proceeds to identify the table-like and list-like structures. The paper by Jeyafreeda et al. (Andrew et al., 2019) examines web page segmentation from the perspective of a clustering problem involving visual elements. The objective is to cluster all visual elements while discovering a predetermined number of clusters, where the elements within each cluster should be visually connected. The study evaluates three clustering algorithms: K-means, F-K-means, and Guided Expansion.

Our present study is connected to RPA (Robotic Process Automation). Robotic Process Automation (RPA) is defined as the application of specific methodologies and technologies that aim to automate repetitive tasks achieved usually by human users (IRPA, 2015), (Hofmann et al., 2020). RPA frameworks (e.g., UiPath[1] , Automation Anywhere[2] , Blue Prism[3] , Microsoft Power Automate[4] , etc.) are designed to develop software robots that improve the business environment in various ways. RPA refers to those tools that operate on the user interface (UI) aiming to perform automation tasks using an "outside-in" approach. The information systems are kept unchanged, compared to the traditional workflow technology, that allows the improvement using an "inside-out" approach (Van-der Aalst et al., 2018). In order to describe business processes in a way that can later be executed automatically by programs, RPA platforms like UiPath, Power Automate, Automation Anywhere etc. operate (i.e. create automated business processes) in the following way: RPA developers identify UI (i.e. User Interface) components of a software application like buttons, text input controls, dropdown lists and tables, and then customize activities by writing code snippets in a programming language in order to act on these UI controls (e.g. click the selected button, write data in the text input, select all data from a table or a dropdown list etc.); this code that references the selected UI controls forms the automated business process which can be executed many times later with different input parameters. Other alternative to the aforementioned commercial RPA platform do exist and they are usually based on PBD (i.e. Program-By-Demonstration) methodologies (Li et al., 2017), (Leno et al., 2020), (Agostinelli et al., 2020).

A work somewhat similar to ours is (Liu and Yang, 2005). In this paper, authors build a topic hierarchy of a website starting from the homepage of the website and continuing in a tree-like structure in which the vertices and edges correspond to web pages and hyperlinks. But our work deals with a much complex topic of mapping the web content generated by a business web application and the map is a conceptual map (it does not include actual web pages).

3 THE STRUCTURE OF THE MAP

The map of the business web application will be a tree-like structure and will describe in an abstract, conceptual way, the use cases (functionalities) of the web application. The map will be a navigational map meaning that it will describe in a compact way all possible navigation paths in the UI of the web application. The elements of the map are the following:

- *Clickable elements*
- *Conceptual operation DOM parts*
- *Web pages*
- *Links*

Conceptual operation DOM parts and *Web pages* are the nodes of the map, *clickable elements* are at-

[1]https://www.uipath.com/

[2]https://www.automationanywhere.com/

[3]https://www.blueprism.com/

[4]https://powerautomate.microsoft.com/en-us/

tributes of the nodes and *links* are the edges of our map.

3.1 Clickable Elements

These are Html elements which can be clicked by the user in the browser, thus triggering a change in the UI of the web application (by sending a complete HTTP request or an XHR/Fetch API HTTP request whose answer would replace totally or partially the currently loaded Html content in the browser window/tab). These *clickable elements* create *links* in our conceptual map of the application between two *Web pages* of the application. From a technical point of view, *clickable elements* are the following Html elements: <button>, <input type="submit">, <a>or any other Html element that has an *onclick* attribute or has a click event listener added on it. In order to discover the Html elements that have a click event listener from the current document, we had to overwrite the *addEventListener()* method on the *Element.prototype* object in the browser.

We identify each *clickable element* in the browser by assigning an unique signature to it (i.e. unique to the current Html document) - the XPATH of that *clickable element* in the current DOM (i.e. Document Object Model).

3.2 Conceptual Operation DOM Parts

A *conceptual operation DOM part* is just a part of the DOM in a business web application that allows the human user to execute a *conceptual operation* on the database. A *conceptual operation* is just a CRUD operation (i.e. Create, Read, Update, Delete) on a concept or entity from the underlying database of the web application. For example, for an *Update* operation on an *Account* entity, usually the web application would provide an Html form with labels and inputs for each field of the *Account* entity. Similarly, for a *Create* operation on an *Account* concept, the web application will usually construct also a form with labels and inputs for all the fields of the *Account* concept. For a *Read* operation on an *Account* entity, a similar DOM part is shown, but this time the text inputs are read-only. And for a *Delete* operation usually just a key of the entity (like the name of the entity) is usually required from the user.

Conceptual operation DOM parts are always the leaves of the tree-like, conceptual map of the web application.

3.3 Web Pages

The *web pages* are the nodes of the map. A *Web page* contains *clickable elements*. Actually, the set of all *clickable elements* from a *web page* form a signature of that *web page*; as mentioned before, each *clickable element* is identified by its XPATH in the current document. From a technical point of view, an *web page* represents an Html document that is loaded in the browser. In the map, *web pages* may lead to other *web pages* or to *conceptual operation DOM part* nodes. The *web pages* are linked together in the map by *links*.

Web pages are actually container nodes which contain a set of *clickable elements*, each *clickable element* connecting the current *web page* with another *web page* or with a *conceptual operation DOM part*.

Even if an actual Html document may contain an Html section corresponding to a *conceptual operation DOM part*, this *conceptual operation DOM part* is not considered to be part of the *web page* corresponding to this Html document. This *conceptual operation DOM part* will form a separate node in the conceptual map (separate from the original *web page*).

3.4 Links

The *links* are just connections between *web pages* or between a *web page* and a *conceptual operation DOM part* in the conceptual map. They form the backbone of the map of the web application. A *link* unites two *web pages* in the map if by clicking on a *clickable element* of the first *web page* we navigate to the second *web page*. A *link* of the map always has directionality.

4 BUILDING THE MAP

The map is built by running a depth-first traversal algorithm on the entire web application. The various web pages of the application are traversed through clickable elements. The algorithm for constructing the map of the web application is described in code listing 1. In the first lines of the algorithm, lines 1-5, we perform initializations of various structures used in the algorithm: the stack, the root of the map and the current *WebPage* node. The algorithm assumes that the starting Html document of the web application is already loaded in the browser. The algorithm initializes a new *WebPage* node from this Html document in line 4, node that will also be the root of the conceptual map. As mentioned before, a *WebPage* node of the map is identified by a list of *clickable elements* and also it has a set of *links* pointing towards other

WebPage nodes or *Conceptual operation DOM parts* (in line 4, *links* are initialized with an empty set).

Following, lines 6-25 define the main cycle of the iterative depth-first traversal algorithm. The *Stack* stores all the *WebPages* that need to be traversed/visited. A *WebPage* is considered visited and removed from the stack when all its *clickable elements* were visited by our tool. On lines 7-10 we take the first *WebPage* from the top of the stack, get it's first unvisited (i.e. unclicked) *clickable element* and navigate through that path by clicking on this unvisited *clickable element* (on line 10). Following, if all *clickable elements* from the *WebPage* node from the top of the stack were already visited, we remove this *WebPage* from the top of the stack (line 12).

Next, on line 14, we try to determine if this new Html document that was loaded in the browser (after executing line 10) represents a generic *WebPage* or a *conceptual operation DOM part* leaf node. If we have found a conceptual operation in the currently loaded Html document (this is performed by the $DetectConceptualOperation()$ algorithm which is detailed in a subsequent code listing), we add this *conceptual operation DOM part* to the map on lines 15-17. Otherwise, in lines 19-23 we add a new, generic *WebPage*, *newWebPage* to the map. We don't forget to link this *newWebPage* to the previous *WebPage* and to add it to the stack in order to be visited later.

5 DETECTING CONCEPTS AND CONCEPTUAL OPERATIONS IN THE DOM

As we said in section 1, in order to avoid a node explosion problem in the generated conceptual map, we only store one node in the map for all entities of a specific database concept and each conceptual operation (i.e. Select, Insert, Update, Delete); i.e. we don't have one map node for each entity, only a single map node for the whole concept. As conceptual operations we consider the 4 basic SQL operations: Select, Insert, Update, Delete, but we also consider a 5-th one, *SelectAll* which represents the operation of selecting many entities, not necessary all, of the same concept. This is because a popular UI pattern for CRM and ERP web applications is to show a list of entities (i.e. a SelectAll operation) from where the human user can choose which entity to update or delete.

The detection of an operation on a concept in the current DOM (i.e. Document Object Model) happens according to the algorithm depicted in

The MapBuilding algorithm is:

```
1:  Stack = []
2:  MapRoot = []
3:  clickableElements = GetClickableElements()
4:  MapRoot = {WebPage : clickableElements, Links : []}
5:  Stack.push(MapRoot)
6:  while (Stack is not empty) do
7:      WebPage = Stack.top()
8:      ClickElem = GetFirstUnvisitedClickableElement(WebPage)
9:      ClickElem.state = visited
10:     Navigate(ClickElem)
11:     if VisitedAllClickableElements(WebPage) == True then
12:         Stack.pop()
13:     end if
14:     conceptualOperation = DetectConceptualOperation()
15:     if conceptualOperation != NULL then
16:         WebPage.Links.add({ClickableElement :
            ClickElem, WebPage : conceptualOperation})
17:         { This is a leaf node of the map }
18:     else
19:         { This is a new WebPage node of the map }
20:         clickableElements = GetClickableElements()
21:         newWebPage = {WebPage : clickableElements, Links : []}
22:         WebPage.Links.add({ClickableElement :
            ClickElem, WebPage : newWebPage})
23:         Stack.push(newWebPage)
24:     end if
25: end while
```

Algorithm 1: The algorithm that constructs the map of the web application.

listing 2. After each UI change generated by a click event, the *diffDOM* (i.e. the difference DOM) is computed. The *diffDOM* is just the part of the DOM that changed after the click event and the $DetectConceptualOperation$ algorithm is run on this difference DOM, not on the whole DOM loaded in the browser, for efficiency reasons.

The algorithm begins with the function $TableOfEntitiesDetected(diffDOM)$ which tries to detect any tables in the *diff DOM*. Tables are important because the may indicate a SelectAll operation on a concept. We can not present here the body of this function due to space constraints, but in essence, the algorithm for detecting a table tries to horizontally align leaf html tags containing only text into horizontal row clusters and then tries to vertically align these horizontal rows into a table. After detecting a table, the algorithm tries to detect if the entities of a concept are rendered in that table through function $DetectConceptInTable(diffDOM)$. Again, we omit the details here, but simply said, the concept is determined by searching its attributes in the table cells of the table header or, if these are not found (e.g. the table may not have a header), the concept is determined by inspecting (i.e. triggering click events) the entities from the first two rows of the table; detecting an entity is done similarly to the way function

$FindOneConcept(diffDOM)$ determines that an entity (of that concept) is rendered in the $diffDOM$: several text attributes (labels) of a concept from the Data Model are found in the $diffDOM$ - more than 75% of the labels of a concept are found.

Next, if there was no *SelectAll* or *Delete* operation on a concept/entity detected, the algorithm tries to find text input elements/tags (i.e. "<input type=text >,<select >,<textarea >") associated to the text labels (i.e. the text attributes of the found entity). We do this in the $FindInputFieldsForTextLabels()$ function. The associated text input tag is searched in the neighborhood of the label, more specifically, the associated input tag is searched in the South-East quarter of a circle with the center in the middle of the label - the closest such text input tag from this South-East quarter is associated to the label. If no such input html tags were detected, that the operation is *Select* on an entity of the detected concept. If, on the other hand, there was text input html elements found in the *diffDOM*, but they are empty (i.e. don't contain values), the conceptual operation would be *Insert*. Otherwise, the conceptual operation is *Update* on an entity of the detected concept.

5.1 Configuration Settings for the Plugin

The initial settings of the plugin for a target web application are the following:

- URL of the target web application together with access credentials
- the Data Model of the database used by the target web application

The Data Model configuration does not need to match exactly the structure of the database used by the application, but it should match the text labels used for each concept of the database on the UI of the application. This is why, primary access to the database used by the target web application is not required in order to use the plugin.

A small snippet from a data model example used for the Microsoft Dynamics 2016 CRM application is given below in JSON format.

The data model describes the concept *Account* with its associated attributes and the concept *Contact* with its associated attributes. It also describes that the *Company* attribute from *Contact* is a foreign key and refers to the primary key *Account Name* from the *Account* concept. Again, we emphasize that these attributes don't have to match exactly the attributes of the database tables, but instead they have to match the text labels for the respective entities on the UI of

The DetectConceptualOperation algorithm is:

```
1:  if TableOfEntitiesDetected(diffDOM) == TRUE then
2:      DetectedConcept = DetectConceptInTable(diffDOM)
3:      return {Concept : DetectedConcept, Operation : SELECTAll}
4:  else
5:      if
        (DetectedConcept = FindOneConcept(diffDOM))! = NULL)
        then
6:          {Several text attributes (labels) of a Concept from the Data
            Model are found in the DOM}
7:          if (The previous click event was triggered on a tag with the
            caption "DELETE") then
8:              return
                {Concept : DetectedConcept, Operation : DELETE}
9:          end if
10:
11:         InputFieldsFound = FindInputFieldsForTextLabels()
12:         if InputFieldsFound == FALSE then
13:             return
                {Concept : DetectedConcept, Operation : SELECT}
14:         else
15:             if (Found Text Input elements are empty (i.e. don't contain
                values)) then
16:                 return
                    {Concept : DetectedConcept, Operation : INSERT}
17:             else
18:                 return
                    {Concept : DetectedConcept, Operation : UPDATE}
19:             end if
20:         end if
21:     end if
22: end if
```

Algorithm 2: The conceptual operation detection in DOM algorithm.

the target web application. The foreign key relation is used by the plugin to identify entities that are linked with the currently identified entity (i.e. entities that are values for a property of the currently identified entity).

6 EXPERIMENTS

We have tested our tool on two commercial, business web application, Microsoft Dynamics 2016 CRM [5] and Atlassian Jira [6]. Although we tested our web automation tool only on these two business web application, we based our plugin implementation on common web design principles (like the fact that an input field is always placed in the web UI either on the right or below or in the south-east part of its corresponding text label), so it should work correctly on other business web applications. We used general templates for

[5]https://learn.microsoft.com/en-us/lifecycle/products/dynamics-crm-2016-dynamics-365

[6]https://www.atlassian.com/software/jira

```
DataModel = {
    "Account" : ["Account Name", "Phone", "Fax", "Website", "Parent Account", "Ticker Symbol",
    "Address", "Primary Contact", "Description", "Industry", "SIC Code", "Ownership"],

    "Contact" : ["Full Name", "Job Title", "Account Name", "Email", "Business Phone", "Mobile Phone",
    "Fax", "Preferred Method of Contact", "Address", "Gender", "Marital Status", "Birthday",
    "Spouse/Partner Name", "Anniversary", "Personal Notes", "Company", "Originating Lead",
    "Last Campaign Date", "Marketing Materials", "Contact Method", "Email", "Bulk Email", "Phone",
    "Fax", "Mail"],

    "ForeignKeys" : [ { ForeignKey : "Company", ForeignTable : "Contact", PrimaryKey : "Account Name",
    PrimaryTable : "Account" } ]
        ...
};
```

the conceptual operations which are not specific to a particular web application, so our web automation tool should, in principle, work with any typical business web application that uses a relational database.

We evaluated the ability of our web automation tool to automatically discover conceptual operations on the UI by going through all use cases/functionalities of the aforementioned business applications. We considered all use cases that involve (i.e. whose outcome is) a CRUD operation in the backend database. So, we ignored use cases that involve only UI modifications, without reading from or changing the data in the database. We ignored use cases that do not operate on the database (they operate just on the data that is shown in HTML documents in the browser) like for example: print current HTML document, draw reports, email a link/current document, import from Excel into the browser window, etc. On all these considered use cases, we tested whether our plugin is able to automatically detect the main conceptual operation. The results are summarized in Table 1.

The second column in the table presents the number of use cases (i.e. potential conceptual operations) we considered/tested in each of the 2 web applications. The third column presents the absolute number of conceptual operations detected and the fourth column presents the percentage of this number of automatically detected conceptual operations in the total number of potential conceptual operations considered (i.e. the second column value). We can see in this table that our web plugin tool was able to detect 79% and respectively 67% of the available conceptual operations in the considered business web applications. The rest of non-discovered conceptual operations involve partial CRUD operations, i.e. operations that only change a single field or a couple of fields from a database entity (e.g. the operation ”Fulfill Order” which only changes a field of the ”Order” entity and does not perform a full-fledged Update operation of all fields).

In Fig. 1 we can see a partial conceptual map generated for the business web application Microsoft Dynamics 2016 CRM. We could not place the complete map here due to space constraints.

7 LIMITATIONS

Our mapping tool should work with any business web application that uses a relational database in the backend because our plugin uses general templates for the conceptual operations which are not specific to a particular web application, and we based our plugin implementation on common web design principles (like the fact that an input field is always placed in the web UI either on the right or below or in the south-east part of its corresponding text label). However, there are some limitations to our approach.
Because our mapping tool detects a conceptual operation in the DOM by mapping text labels in the DOM (that represent properties of a concept from the underlying database of the application) to text inputs, for operations like partial CRUD operations that only modify a single or just a few properties of a concept, our mapping tool would not be able to detect a conceptual operation in the DOM (just a ‘Generic DOM‘ would be detected). The operation will still be labeled by default by our tool with the text found on the *clickable element* that triggered that operation on the UI, but the effect on the UI would still be a ‘Generic DOM‘. Our mapping tool allows the human user to change the ‘Generic DOM‘ node in the map to some other *conceptual operation DOM part* node.

8 CONCLUSIONS

We presented an automated tool that can be used to construct conceptual maps of business web applications. This conceptual map depicts in an abstract, hierarchical way the possible navigation and opera-

Table 1: Automatic discovery of conceptual operations on the User Interface.

Target application	Total number of use cases (i.e. conceptual operations)	Number of conceptual operations automatically discovered	Percentage of conceptual operations automatically discovered
Microsoft Dynamics 2016 CRM	102	81	79%
Atlassian Jira	39	26	67%

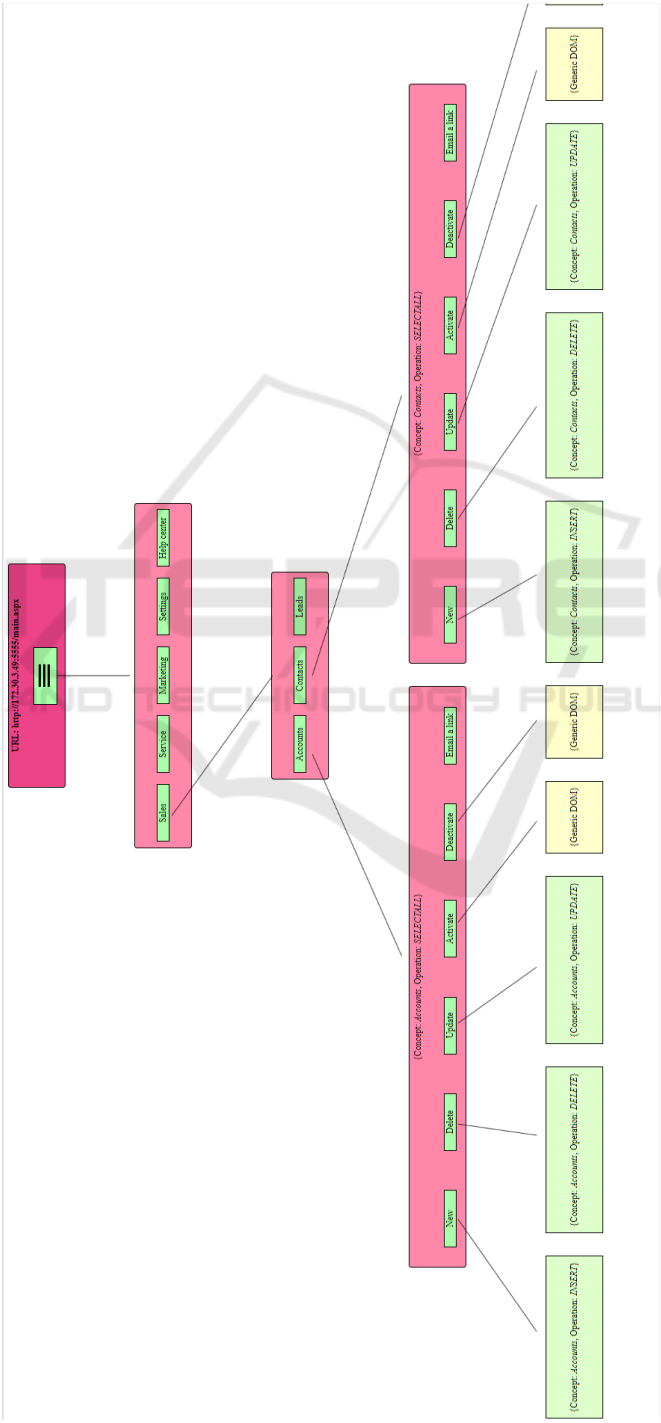

Figure 1: Partial map of Microsoft Dynamics 2016 CRM.

tional paths in the UI (i.e. User Interface) of the web application. The map is produced in either serialized, JSON format or in a graphical form in Html. We have tested our tool on two commercial web applications and found that our tool is able to construct a conceptual map for those applications.

Obtaining such a map offers important advantages especially in the RPA tools' development. The map could be used as a foundation for automatizing the operations allowed to be executed through the web UI of an application.

As future work, we plan to improve our algorithms of conceptual operation detection so that it works also on partial CRUD operations and we plan to add an RPA execution capability to our tool so that it behaves also as RPA software.

ACKNOWLEDGEMENTS

The present work has received financial support through the project: *Integrated system for automating business processes using artificial intelligence*, POC/163/1/3/121075 - a Project Cofinanced by the European Regional Development Fund (ERDF) through the Competitiveness Operational Programme 2014-2020.

REFERENCES

Agostinelli, S., Lupia, M., Marrella, A., and Mecella, M. (2020). *Automated Generation of Executable RPA Scripts from User Interface Logs*, pages 116–131.

Andrew, J. J., Ferrari, S., Maurel, F., Dias, G., and Giguet, E. (2019). Web page segmentation for non visual skimming. In *The 33rd Pacific Asia Conference on Language*, Japan. hal-02309625. Information and Computation (PACLIC 33) Hakodate.

Cai, D., Yu, S., Wen, J.-R., and Ma, W.-Y. (2003). Extracting content structure for web pages based on visual representation. In *Web Technologies and Applications: 5th Asia-Pacific Web Conference, APWeb 2003, Xian, China, April 23–25, 2003 Proceedings 5*, pages 406–417. Springer.

Feiz, S., Wu, J., Zhang, X., Swearngin, A., Barik, T., and Nichols, J. (2022). Understanding screen relationships from screenshots of smartphone applications. In *27th International Conference on Intelligent User Interfaces*, IUI '22, page 447–458, New York, NY, USA. Association for Computing Machinery.

Hofmann, P., Samp, C., and Urbach, N. (2020). Robotic process automation. *Electronic Markets*, 30(1):99–106.

IRPA (2015). Introduction to robotic process automation. a primer.

Leno, V., Deviatykh, S., Polyvyanyy, A., Rosa, M. L., Dumas, M., and Maggi, F. M. (2020). Robidium: Automated synthesis of robotic process automation scripts from UI logs. In *Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2020 co-located with the 18th International Conference on Business Process Management (BPM 2020), Sevilla, Spain, Sept. 13-18, 2020*, volume 2673, pages 102–106. CEUR-WS.org.

Li, T. J.-J., Azaria, A., and Myers, B. A. (2017). Sugilite: Creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 6038–6049, New York, NY, USA. Association for Computing Machinery.

Li, T. J.-J., Popowski, L., Mitchell, T., and Myers, B. A. (2021). Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA. Association for Computing Machinery.

Liu, N. and Yang, C. (2005). Mining web site's topic hierarchy. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, page 980–981, New York, NY, USA. Association for Computing Machinery.

Mantratzis, C. and Cassidy, S. (2005). Dom-based xhtml document structure analysis separating content from navigation elements. In *International Conference on Computational Intelligence for Modelling*, pages 632–637, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06), Vienna, Austria. Control and Automation and International Conference on Intelligent Agents.

Nguyen, T. A. and Csallner, C. (2015). Reverse engineering mobile application user interfaces with remaui. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 248–259. IEEE Press.

Van-der Aalst, W. M. P., Bichler, M., and Heinzl, A. (2018). Robotic process automation. *Business and Information Systems Engineering*, 60:269–272.

Wu, J., Wang, S., Shen, S., Peng, Y.-H., Nichols, J., and Bigham, J. P. (2023). Webui: A dataset for enhancing visual ui understanding with web semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA. Association for Computing Machinery.

Wu, J., Zhang, X., Nichols, J., and Bigham, J. P. (2021). Screen parsing: Towards reverse engineering of ui models from screenshots. In *34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*, pages 470–483, New York, NY, USA.