Appstract: Towards Application-wide Locators for the Web

ABSTRACT

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:

. 2021. Appstract: Towards Application-wide Locators for the Web. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Web applications are often perceived by end users as online systems exposing a limited set of views, concepts, and related actions. From a user perspective, navigating through an online shop, like Amazon, mainly offers to search and list products, while clicking on a given product brings us to its details view. From a machine perspective, like a bot, navigating the same web application, will be perceived as crawling thousands of unique pages, exposing unrelated content and proposing unique actions.

This inability for a machine to understand the navigation model of a web application, like a user intuitively does, makes it very hard to automate interactions with web applications. Typically, research topics benefiting from more intuitive navigation include: *(a) web data mining* to automatically extract, de-noise, and structure data from web pages, *(b) web testing* to generate, maintain, repair, and augment tests on web applications, and *(c) web analytics* to report on how users interact with the web application.

While there is a considerable amount of existing literature, notably in the fields of data mining [**?** **?** **?** **?** **?** **?**], most of existing works are highly specific and do not provide adequate insights to build an application-wide understanding of a navigation model.

In particular, state-of-the-art approaches focus on either extracting data within a page (so-called, *intra-page* extraction) or across two pages (*inter-page* extractions).

For example, Miao *et al.* [**?**] clusters full tag paths (i.e. /html/body/div[2]/div[1]/span) to detect repeating occurrences of a template within a page. This allows their approach to extract what is usually called *records* in a web page (*e.g.*, a single product card in a product list). This is an example of what we categorize as *intra-page abstraction.*

Inter-page abstraction solutions usually try to detect templates of a webpage or to learn a wrapper by studying two different instances of the same template. That is what is done by the algorithms like EXALG [**?**] and RoadRunner [**?**]. These approaches do not offer any solution to also take into account the intra-page variability.

In this paper, we thus propose an unsupervised approach to infer the navigation model of a whole web application that we call an *appstraction.* The *appstraction* of a web application aims at delivering actionable insights: it allows a machine to understand application states (*e.g.*, product pages, blog page) as well as elements within states (*e.g.*, product title, price), hence guiding the information extraction process, as well as identifying navigation actions. The *appstraction* process aims to abstract away the natural variability of web pages into a canonical model composed of as a compact tree of *template pages* and *template elements.* Our unsupervised approach assumes that even web applications with billions of pages will build on a limited set of template pages, thus making it possible to infer these generative templates from a dataset of visited pages. To achieve this, our approach—named Appstract—builds on three stages: (1) *web page clustering* to group instances of related web pages, (2) *intra-page abstraction* to extract repeating patterns within each cluster of pages, and (3) *inter-page abstraction* to capture repeating patterns across clusters.

We empirically demonstrate that our appstraction succeeds to generate application-wide locators that can be used to support semantic guidance across multiple pages of any web application.

The remainder of this paper is therefore organized as follows. Section 2 introduces the required background and related works in this area. Section 3 presents the design and implementation of Appstract, while Section **??** reports on an evaluation of the perceived accuracy of Appstract. Finally, Section 5 concludes.

2 BACKGROUND & RELATED WORKS

Throughout this paper, we consider a subset of the Amazon web application as a running example. In particular, we consider two related page templates: the product details and product list pages (cf. Figure 1).

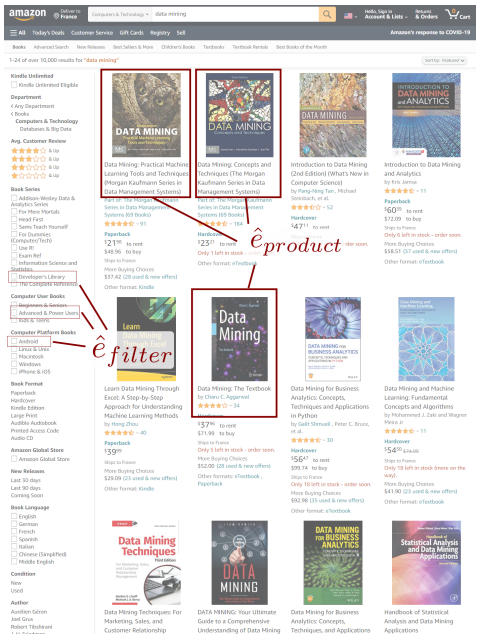
Data Extraction. Understanding the navigation model of a web applications implies to extract enclosed data pattern. All existing literature on data extraction relies on one of two approaches:

- *supervised approaches* where the user inputs more or less detailed directions to describe to how to extract data [**?**],
- *unsupervised approaches* where data is automatically extracted by analysing recurring patterns [**?**].

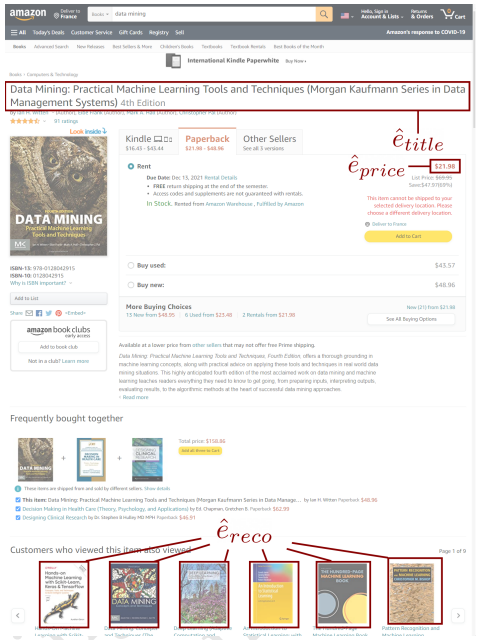
In this paper, we focus solely on *unsupervised approaches.* These approaches rely on the assumption that, if there is a lot of different pieces of information exposed by an application, the same type of information will always be presented in the same way. For example, on the product list page of an online shop, each product will be presented in a similar fashion (e.g. product name, price,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Sample page from cluster *list*.



(b) Sample page from cluster *product*.

Figure 1: Screenshots illustrating some elements $\hat{e}_\lambda$ included in the template pages $\hat{p}_{list}$ and $\hat{p}_{product}$ of 2 distinct clusters.

description). In this context, we distinguish two families of unsupervised approaches: *intra-page* and *inter-page* data extraction. One should note that *intra-page* data extraction is also referred as *record extraction* [?].

*** Related work on record extraction

Similarly, different product pages detailing different products share a similar structure since they present the same *type* of information. Therefore, we distinguish two types of variability:

- *intra-page variability* for repeating pattern within one page (e.g. list of products on Amazon),
- *inter-page variability* for repeating pattern across several pages (e.g. product page on Amazon).

We use this idea to draw a model of the application as a whole (site-level).

2.1 Related Works

2021 Modeling Web Browsing Behavior across Tabs and Websites with Tracking and Prediction on the Client Side [?]

- Web behavior analysis using clickstream
- Novelty: Client Side only / Multi-tab analysis
- End results: Classify navigation types / Predict user next click
- How: url encoding - weird RNN fed with actions and timestamps
- Eval: Give missions to users on different websites (132 action paths / 19 test)
- Results: classification -> 100%, prediction -> not amazing

What did they do? Understanding clickstreams with the WebQuilt visualization system [?]

3 APPSTRACT

3.1 Abstracting a Web Application

This section starts by defining the notions of application and application abstraction we assume, before discussing the challenges of inferring such defined abstractions.

Definition 3.1 (Application)**.** An web application A is a set of web pages $\{p \in A\}$, where every page p is captured by a *Document Object Model* (DOM).

The number of web pages $\|A\|$ can be very high and keep rising with time (*e.g.*, Amazon products or blog posts). In addition, we consider that any mutation of a web page (even if the URL does not change) is considered as a new page. Obviously, from a human perspective, a web application is much more than a collection of pages (*e.g.*, the pages are linked, the application has different features usable by different users), but we choose to take the point of view of a machine, for which an application is perceived as a set of visited web pages. As a DOM of a page p is a tree, we represent it as a tuple $\langle N(p), par\rangle$ where $N(p)$ is the set of elements (or nodes) in p and $par : N(p) \rightarrow N(p)$ is the parent function that associates a DOM node with its parent. To lighten the notations, we write $e \in p$ to describe an element e in a page p, instead of writing $e \in N(p)$.

Definition 3.2 (Application abstraction)**.** The abstraction of a web application A is a tuple $\langle \hat{A}, T_{\hat{A}}\rangle$ where *i)* $\hat{A}$ is the set of template pages and each page $\hat{p} \in \hat{A}$ contains template elements $\hat{e} \in \hat{p}$ and *ii)*:

$$\begin{aligned} &\forall p \in A, T_{\hat{A}}(p) = \langle \hat{p}, T_{\hat{p}}\rangle \\ &\text{and } \forall e \in p, T_{\hat{p}}(e) = \langle \hat{e}\rangle \end{aligned} \tag{1}$$

In other words, an abstraction of a web application A is a set of template pages (*e.g.*, product page, list page) and a function that allows to map any page from A to its corresponding template, and every element within the page to the corresponding element in the template page. This mapping is completed by two functions:

(1) $T_{\hat{A}} : A \rightarrow \hat{A}$ is the template function that takes any page $p \in A$ and returns the matched *template page* $\hat{p} \in \hat{A}$, and
(2) $T_{\hat{p}} : N(p) \rightarrow N(\hat{p})$ is a function that takes any element $e \in \hat{p}$ and returns the matched element in the template $\hat{e} \in \hat{p}$.

Additionally, we also use:

- $T^{-1}_{\hat{A}}(\hat{p}) \subset A$ as the set of pages $p \in A$, such that $T_{\hat{A}}(p) = \hat{p}$, and
- $T^{-1}_{\hat{p}}$ as the inverse function of $T_{\hat{p}}$.

These notations allow us to easily refer to the instance pages/elements of a given template page/element.

Figure 1 depicts a theoretical application of the appstraction to 2 page web pages crawled from Amazon. For each related clusters of similar web pages, the $T_{\hat{A}}$ function will return either template page $\hat{p}_{list}$ or $\hat{p}_{product}$. Then, for each clustered web page, each web element can be mapped to its corresponding template element (*e.g.*, $\hat{e}_{title}$, $\hat{e}_{product}$) using the $T_{\hat{p}}$ function.

Please note that our appstraction process does not intend to label template pages or elements. Thus, our references to $\hat{p}_{list}$ or $\hat{e}_{product}$ should be understood as a *global unique identifier* (GUID) capturing a class of pages and elements within and across clusters.

3.2 Building an Abstraction

To abstract the web page p into its appstraction A_p, we operate at two levels:

(1) *intra-page abstraction* to extract repeating patterns of elements within a cluster of web pages, and
(2) *inter-page abstraction* to group repeating patterns of elements across template pages.

This bi-level abstraction process combines two steps: *learning* and *prediction*, as depicted in Figure 2. In this section, we first overview of the abstraction process by considering intra- and inter-page abstractions as black boxes. In the following sections, we provide am in-depth description of individual level abstraction techniques.

Learning. The objective of the learning phase is to build a model of the application that will deliver accurate predictions. The upper part of Figure 2 describes how intra- and inter-page abstractions are used to build this model.

Before any abstraction, the application is perceived as a set of related pages. The first step is thus to organise these pages into clusters, each cluster representing a template page (*e.g.*, product page). In order to do this, any clustering algorithm could be used and, in this paper, we consider the clustering part as out of scope as we focus on the actual abstraction process. The learning approach thus starts with the user picking one example page per template.

Each template page goes through intra-page abstraction. The intra-page abstraction takes a DOM tree p as input and returns a *template* as output. A *template* is a tuple $t = \langle \hat{p}, T_p \rangle$ where T_p is the identifier map that maps every element from the original tree p to a global identifier. After the intra-page abstraction, $\hat{p}$ typically contains much less elements than p since all repeated patterns have been abstracted away. The T_p map is a way to keep track of the elements in the original page p after abstraction. For example, in Figure 3 illustrating intra-page abstraction, the identifier map would contain 11 entries, one for each original element in p and 5 distinct values corresponding to the five distinct nodes of $\hat{p}$.

In practice, the values of T_p are *Global Unique Identifiers* (GUID). Before starting the abstraction process, we transform the page p into a *template* tuple, where the identifier mapping maps every node to a randomly generated GUID. Each abstraction step will thus transform the input tree and update the associated identifier map.

Once each page has been abstracted at the intra-page granularity, we build a model allowing to achieve two aspects of inter-page abstraction:

- *Template abstraction:* same elements within different instances of a template must have the same identifiers (*e.g.*, title of a product on different product pages)
- *Cross-template abstraction:* same elements within different pages (regardless of templates) should have the same identifiers. (*e.g.*, menu link that appears on all template pages)

To achieve cross-template abstraction, we arbitrarily select one template to represent the whole application ($\langle \hat{p}_a, T_{p_a} \rangle$ in Figure 2) we call this template the *mother template*. All other template pages are then inter-page abstracted against the mother template. The inter-page abstraction function takes two templates: a reference template and a query template. It then returns one template in which the values of the identifier map have been updated to match the reference template.

The final model is obtained after applying intra-page abstraction between the mother template and all other pages. The final model is simply a set of n templates where each template is a tuple $\langle \hat{p}, T_p \rangle$.

Prediction. During the prediction phase described in the bottom part of Figure 2, the user sends a previously unseen page and Appstract returns the mapping between each element e of the page and the *id* of the associated template element $\hat{e}$.

To do so, Appstract first applies intra-page abstraction to create an abstracted DOM tree $\hat{p}$, then applies the tree matching algorithm at the core of the inter-page abstraction between $\hat{p}$ and all templates in the model. The template page from the model that matches best with $\hat{p}$ is retained (*e.g.*, $\hat{p_b}$ in Figure 2).

The matching between $\hat{p}$ and the corresponding template is then used to compute the final mapping $T_{\hat{p}}$.

Overall, the user sent a page and received the mapping between each element of the page and the corresponding template element in the corresponding template page.

In the following sections, we describe both intra-page abstraction and inter-page abstraction in more details.

3.3 Intra-Page Abstraction

Intra-page abstraction relies on the detection of repeating patterns within a page.

Building the intra page abstraction corresponds to creating the T_e function defined above. The intra-page abstraction deals with

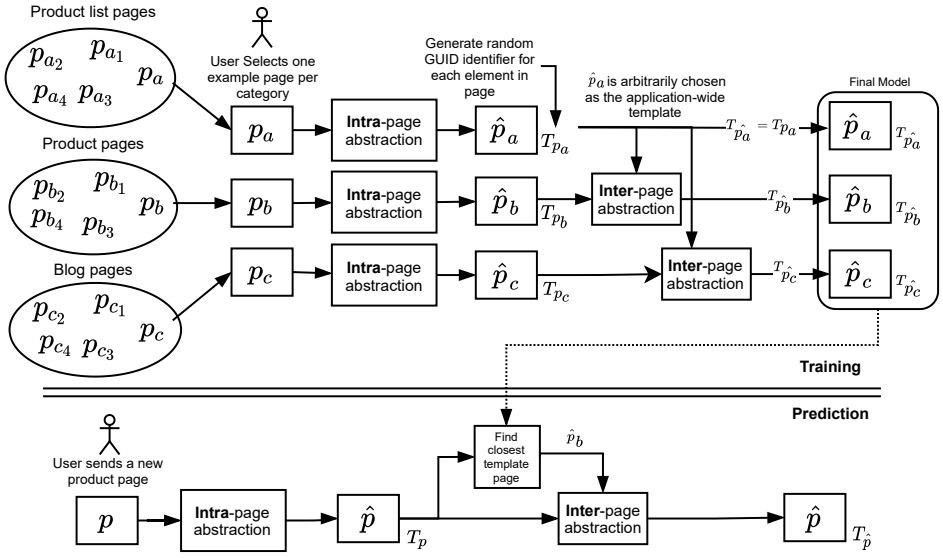


Figure 2: Overview of the two stages of the appstraction process: *learning* and *prediction*. The identifier map of p noted T_p associates an identifier to each element of p. The identifiers are first randomly generated, then merged to existing maps during inter-page abstraction. The prediction phase produces an identifier map $T_{\hat{p}}$ in which each original element of p is associated to an application-wide identifier.

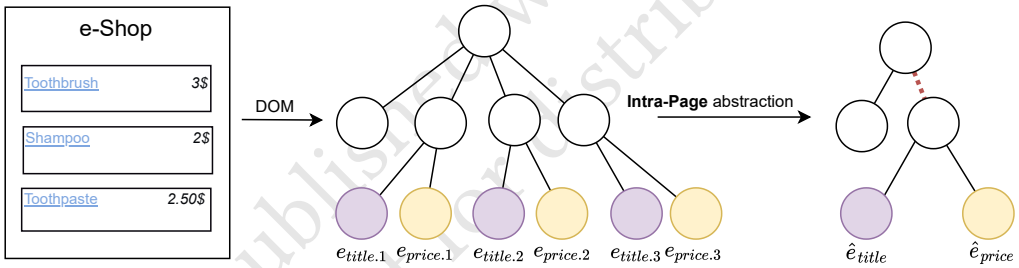


Figure 3: Illustration of Intra-Page abstraction. DOM leaves at the end of repeating branches are tagged and then recursively merged.

a single page: given an input page p, we want to build a template page $\hat{p}$ and a function T_e that maps all the elements $e \in p$ to their corresponding elements $\hat{e} \in \hat{p}$ Overall, the Intra-Page abstraction is done in three steps:

(1) *Leaf Clustering*: cluster repeating DOM leaves together
(2) *Node Tagging*: propagate information about leaf-clusters in all ancestor nodes to prepare for the final step
(3) *Recursive Leaf-Group Merging*: build the intra-page abstraction by recursively merging branches

3.3.1 Leaf Grouping. In the first step of Intra-Page abstraction, we attempt to identify groups of leaf nodes that present data of the same type. To do so, we: (1) Build all root-to-leaf paths from the DOM tree (2) Group all same paths together (3) Filter out groups of leaves containing less than a fixed threshold k of elements. At the end of this phase, we have a set of leaf groups (LG), each containing at least k elements.

What we call root-to-leaf path of a node is the formatted sequence of tags from the root to the node. For example, on the web page from figure 4, the root-to-leaf path of the element containing the text *price2* is //html/body/div/span/

To find leaf-groups, we simply extract all root-to-leaf paths from the DOM tree, group the same ones together and only keep the groups that contain more than a fixed threshold k of elements. In example 4, assuming $k = 2$, it means we have two groups:

```
Leaf Group 1: //html/body/div/span/
Containing: price1 and price2

Leaf Group 2: //html/body/div/a/
Containing: Item1 and Item2
```

```
<html>
  <Head></Head>
  <body>
      <div></div>
      <div>
          <a>Item1 </a>
          <span>price1 </span>
      </div>
      <div>
          <a>Item2 </a>
          <span>price2 </span>
      </div>
  </body>
</html>
```

Figure 4: Web page example to illustrate intra-page abstraction with no nested records

In our evaluation however, this threshold is set to $k = 4$.

Limits. The grouping method described here is very simple. It may fail to cluster items correctly in two ways:

- *Sur-abstraction*: Items can be clustered together even if they are not of the same type.
- *Sub-abstraction*: Items can be put in different clusters even though a human would put them in the same cluster.

To a certain extent, *Sub-abstraction* can be compensated afterwards. For example, when extracting information, it is common for websites to structure data differently according to the type of product presented (e.g. regular products or "sponsored" products). In this case, using the approach we presented, the two types of products will be classified in different clusters, meaning that if the data is extracted as a table, the prices will be spread in two different columns that the user will be able to easily merge.

In the cases of *Sur-abstraction* are more problematic since it will mean that the associated data has been lost when abstracting the page.

3.3.2 Node Tagging. At this stage, we have a list of all leaf-groups (LG_1, LG_2...). In order to be able to recursively merge all leaf-groups, we propagate the information that a leaf belongs to a certain group to all ancestors of the leaf using the node-tagging algorithm 1.

Algorithm 1 iterates through all the leaves that belong to a leaf group LG and for each leaf, it recursively tags all the ancestors of the leaf. The *tag* of an element e is the tuple $\langle LG, depth \rangle$ where LG is a leaf-group and $depth$ is the number of nodes between e and the leaf from its offspring that belongs to LG. The tag is used to identify nodes that should be merged together.

When the algorithm ends, each node e of the DOM tree contains a map *tags* whose keys are tags and values are integers count. Figure 5 shows an example result of the node-tagging algorithm on a simple DOM tree.

An important implementation detail: the algorithm defined above assumes that the *tag* tuple's hash code will be generated from the value of its components (and not the object's reference), i.e. two

Algorithm 1 Intra-Page abstraction: Node Tagging

```
1:  function createTags(LGs = [LG1, LG2...])
2:      function tagBranch(depth, LG, e)
3:          tag ← ⟨LG, depth⟩
4:          e.tags[tag]+ = 1 // Inc or init tag count of node e
5:          tagBranch(depth + 1, LG, e.parent)
6:      end function
7:      for all LG ∈ LGs do
8:          for all e_leaf ∈ LG do
9:              tagBranch(0, LG, e_leaf)
10:         end for
11:     end for
12: end function
```

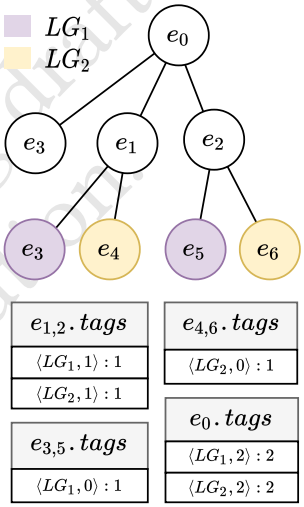


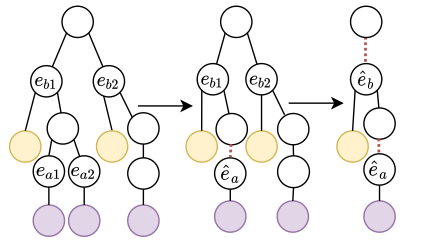
Figure 5: Output of the node-tagging algorithm. Each node is assigned a *tags* map keeping track of the number of leaf-groups in its offspring.

tag tuples containing the same leaf group LG and the same $depth$ should have the same hash code when inserted into the $e.tags$ map (even though the tags will have different addresses in memory since they were created at a different stage of the algorithm).

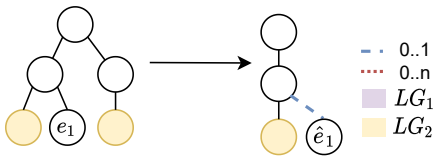
3.3.3 Recursive Branch merging.

Overview. The last step of the intra-page abstraction consists in merging all required nodes such that the resulting DOM contains only one node instance of each leaf group. Figure 3 shows an example of the result obtained after this last step of the intra-page abstraction. In the final abstract tree of figure3, the red dotted edge connecting the template element $\hat{e}$ to its parent indicates that $\hat{e}$ has a special relationship with its parent, in this case: a 1 to many relationship.

The algorithm must be recursive because it is possible (and likely) that some leaf groups will have to be merged at different levels of the tree. Figure 6 illustrates this use case: e_{a1} and e_{a2} must be merged first before their ancestor e_{b1} can merge with e_{b2}.



(a) Nested Records



(b) Optional Elements

Figure 6: Illustrating two important cases our merging algorithm must cover: nested records and optional elements

The second case illustrated in figure 6 b) occurs when merging two nodes even though not all their respective children are merged. In this case, the inferred abstract node $\hat{e}_1$ is said to have an *optional* relationship with its parent template element i.e. it only exists as a child node in certain instances of $\hat{e}_1$. In general, a template element $\hat{e}$ can have many types of relationship with its parent. In this paper, we distinguish two:

- A zero to many relationship: at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ have 0 child element that is an instance of $\hat{e}$ and at least one other element has more than 1.
- An optional (or zero to one) relationship: at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ have 0 child element that is an instance of $\hat{e}$ and at least one other element has more than 1.

Algorithm. Algorithm 2 gives a high level view of the algorithm. The *abstractTree* function recursively traverses the tree. For each node e considered, it checks if e is already abstract using the *isAbstract* function (line 2). A node is abstract if none of its children need to be merged. If e is already abstract then it is returned, otherwise, we:

(1) Recursively call the *abstractTree* function on all children of e
(2) Group all abstract children by group of tags using *assocGroup* (see paragraph 3.3.3)
(3) Merge each group of children using *mergeGroup* (see paragraph **??**)

We describe in details the three functions used in algorithm 2: (1) *isAbstract* (2) *assocGroup* (3) *mergeGroup*

isAbstract. The *isAbstract* function checks if an element e is already abstract. An element e is abstract if none of its offsprings need to be merged. This information is obtained using the tags computed in previous steps (see section 3.3.2.

Each tag *tag* associated to a node e is associated to its tag count $|tag|$. Using the tag counts, we can then easily detect if e is abstract:

Algorithm 2 Intra-page abstraction: recursive merge

```
1:  function abstractTree(e : Element)
2:      if isAbstract(e) then
3:          return e
4:      else
5:          children ← e.children.map(abstractTree)
6:          children ← assocGroup(children).map(mergeGroup)
7:          ê ← { e with e.children = children }
8:          return ê
9:      end if
10: end function
```

a node e is abstract iff all tag counts are equal to 1. In this case, it means no node in the offspring will have to be merged. For example, in figure 5, all nodes are abstract except e_0 since two tags in $e_0.tags$ have a tag count of 2. In practice, it means that some children of e_0 (in this case e_1 and e_2) will have to be merged.

assocGroup. The *assocGroup* (for associative grouping) function groups all nodes that need to be merged together. In order to know if two nodes need to be merged, we look at their associated *tags* map. The merging condition is simple: two nodes should be merged iff they share one same tag.

Below is an example of input/output pair of the *assocGroup* function.

```
Input:
A has t1, t2
B has t2, t3
C has t3
D has t4
E has t4

Output:
[A, B, C] -> [t1, t2, t3]
[D, E] -> [t4]
```

In our case, the letters A to E are nodes and $t1$ to $t4$ are tags.

mergeGroup. **??** The *mergeGroup* function is the core of the recursive merging algorithm. As an input, it takes a list of abstract nodes (i.e. abstract subtrees) that need to be merged and the set of common tags between the groups of nodes. The output is the abstract node $\hat{e}$. The inputs come from the output of the *assocGroup* function described above and the abstract node output replaces all children that were merged in the tree as shown in the *abstractTree* algorithm 2

The *mergeGroup* function reduces the group list taken as input by repeatedly applying the function *mergeAbstractTrees*. Algorithm 3 describes how the function merges two abstract trees into one.

Before diving into the details of the algorithm, it is important to highlight that the inputs the *mergeAbstractTrees* function recieves are already abstract. At this stage of the algorithm we are assured that along the whole branch starting at the root of both nodes sent as parameters, there is never two children belonging to the same cluster (i.e. that need to be merged). It means that the algorithm's

Algorithm 3 Intra-page abstraction: merge two abstract trees

```
1:  function MERGEABSTRACTTREES(ê1, ê2)
2:      // Group pairs of children containing the same tags
3:      pairs, orphans ← groupPairs(ê1, ê2)
4:      mergedChildren ← pairs.map(mergeAbstractTrees)
5:      for e in orphans do
6:          e.rel ← relType.Optional
7:      end for
8:      ê ← new Node()
9:      ê.tag = ê.tag
10:     ê.attrs = mergeAttrs(ê1.attrs, ê2.attrs)
11:     ê.rel = mergeRel(ê1.rel, ê1.rel)
12:     ê.children = mergedChildren + orphans
13:     ê.tags = {ê1 ∪ ê2 with tag counts set to 1}
14:     return ê
15: end function
```

sole purpose is to recursively merge the two trees between them (and not within).

The function starts by grouping the children of the two nodes into pairs and orphans. The *groupPairs* returns two lists: the pairs of nodes that must be merged and the orphan nodes. The orphan nodes are the children in $e_{1/2}^{\hat{}}$ that have no corresponding element to be merged with in $e_{2/1}^{\hat{}}$, these elements are set as optional using the *rel* (as in relationship) property.

Before merging, we recursively call the *mergeAbstractTrees* function on each pair of nodes returned by *groupPairs*. Intuitively, the algorithm will stack the calls to *mergeAbstractTrees* until it reaches the leaves of the trees, then it will merge the groups of leaves and merge their ancestors as the function calls unstack.

Most of the steps described above help computing the *children* property of the abstract node returned by the *mergeAbstractTrees* function. Other properties of the nodes are also merged:

rel: In case the relationships of the nodes to merge are different, they are merged using the following pattern matching:

```
mergeRelTypes :: (RelType, RelType) -> RelType
mergeRelTypes (t1, t2)
    | (t1, t2) when t1 = t2 -> t1
    | (t1, Normal) -> t1
    | (_, ZeroToMany) -> ZeroToMany
    | (Optional, OneToMany) -> ZeroToMany
    | (t1, t2) -> mergeRelTypes (t2, t1)
```

attrs: In case the attributes of the nodes to merge are different, they are merged. To merge the attributes, we select the attributes that are present in both nodes and merge their values using the Longest Common Subsequence (LCS) algorithm. For example, given three nodes having the following class values: *class* attribute: "link nav–link", "link active nav–link" and "link nav–link", the *mergeAttrs* function will return the following value: "link nav–link".

Stack Diagram. Since the algorithm has several levels of recursivity, it may be hard to understand how a given tree will be abstracted.

In figure **??** we describe the different steps of the algorithm. At each step, we show the output of the current function that is called. Below each tree, we show the current stack of functions called.

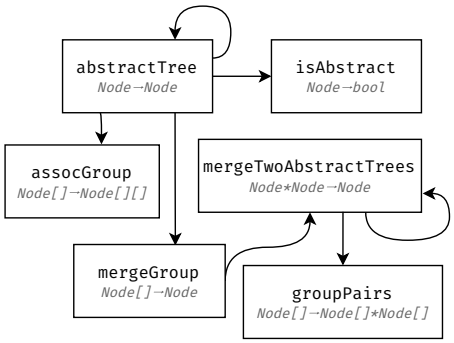


Figure 7: Key functions involved in the *abstractTree* algorithm. A directed arrow from function f to g indicates that f calls g.

The functions we mention are all described earlier:

Figure 8 describes an example application of the *abstractTree* function. At each step, the current stack of functions is shown at the bottom. All functions shown have been described in the previous sections. The figure 7 summarizes all these functions and how they interact. For simplicity however, the *mergeTwoAbstractTrees* is not explicitly mentioned, it is considered as part of the *mergeGroup* function.

3.3.4 *Complexity.*

3.4 Inter-Page Abstraction

We described how to detect and abstract the repeating patterns contained within one page. In the second part of the Appstract approach, we detect and abstract repeating patterns across pages of the web application.

The inter-page abstraction relies on tree-matching. A tree matching solution allows to match two web page DOM trees p and p'. The matching $M_{p,p'}$ obtained is a subset of $p \times p'$ such that each tuple $(e, e') \in M_{p,p'}$ represents the fact that the element $e \in p$ matches with the element $e' \in p'$ (e.g. e and e' contain the names of two different products on two different product pages p and p').

Intuitively, the idea of the inter-page abstraction step is to iteratively match all pages with their template page

4 LIMITS

Modeling a whole application is a highly ambitious task. We hope our work can help progress toward this goal but we cannot claim that it already does. Indeed, our current work has several limits, mainly:

Template topology Real-life templates may have a much more complex structure than the one we assume:

(1) There can be a tree-like structure with deeply nested templates
(2) There could be graph-like template structure (e.g. components)

Our current appstraction method do not allow to infer such template topology.

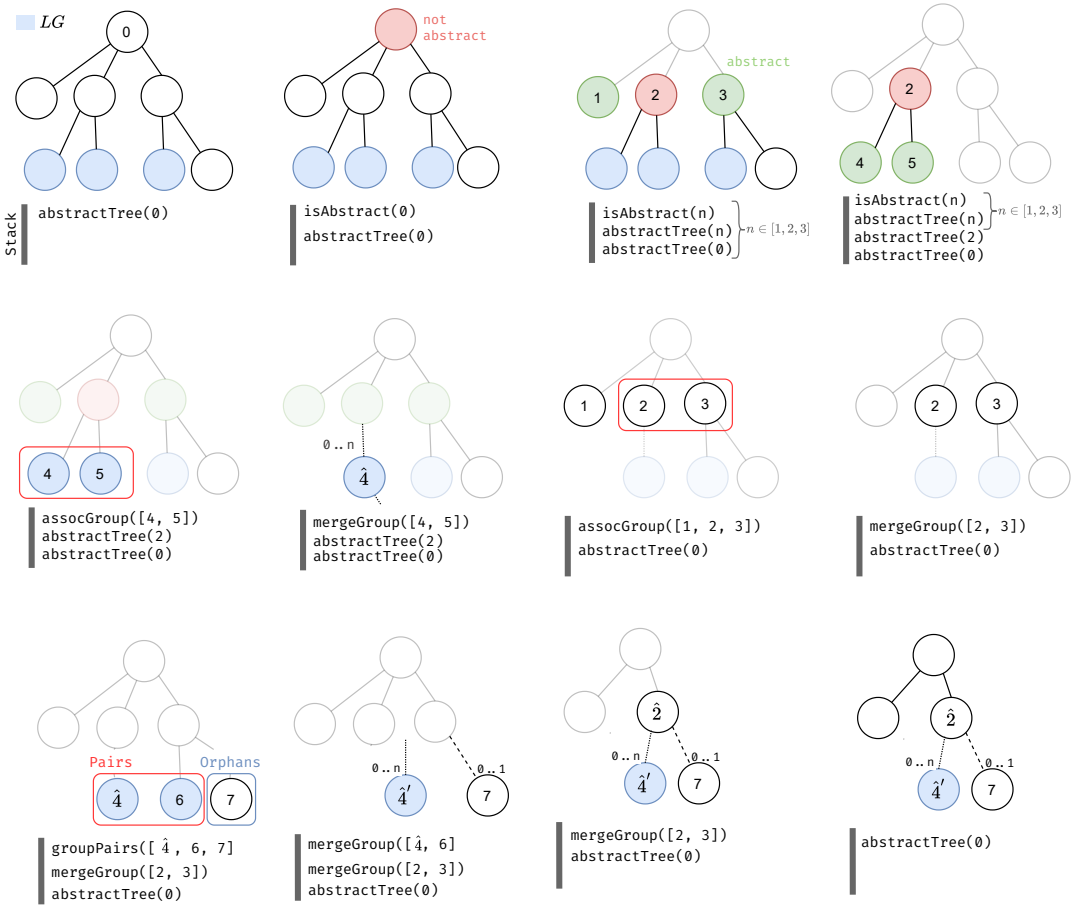


Figure 8: An example application of the *abstractTree* function. For each figure, the stack of the current step is shown below.

5 CONCLUSION