Detecting Email Components with Constraints: Expressive and Extensible Models in Answer Set Programming

Gromit Yeuk-Yin Chan
ychan@adobe.com
Adobe Research
San Jose, USA

Shunan Guo
sguo@adobe.com
Adobe Research
San Jose, USA

Caroline Kim
carolinekim.j@gmail.com
Adobe Inc.
San Francisco, USA

Cole Connelly
cconnell@adobe.com
Adobe Inc.
New York, USA

Michelle Lee
mlee@adobe.com
Adobe Inc.
New York, USA

Andrew Thomson
anthomso@adobe.com
Adobe Inc.
San Francisco, USA

Eunyee Koh
eunyee@adobe.com
Adobe Research
San Jose, USA

ABSTRACT

Detecting structures and components in business emails is vital for editor software to convert third-party emails so that designers can edit them without needing to know how HTML works. In a production environment, the challenge is to make the model easy to be understood and maintained by different stakeholders. We propose detecting email components with a collection of constraints written in Answer Set Programming (ASP). Hard constraints can detect well-defined components like email layouts, and soft constraints can incorporate ML to detect custom components like buttons and titles in emails. Using constraints, developers can apply their domain knowledge to the model and express them in a concrete, extensible, and deterministic form. We demonstrate the effectiveness with a prototype and evaluations from real datasets.

CCS CONCEPTS

• **Human-centered computing**;

KEYWORDS

Email Component Detection, Constraints, Answer Set Programming

ACM Reference Format:
Gromit Yeuk-Yin Chan, Shunan Guo, Caroline Kim, Cole Connelly, Michelle Lee, Andrew Thomson, and Eunyee Koh. 2023. Detecting Email Components with Constraints: Expressive and Extensible Models in Answer Set Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3544549.3585714

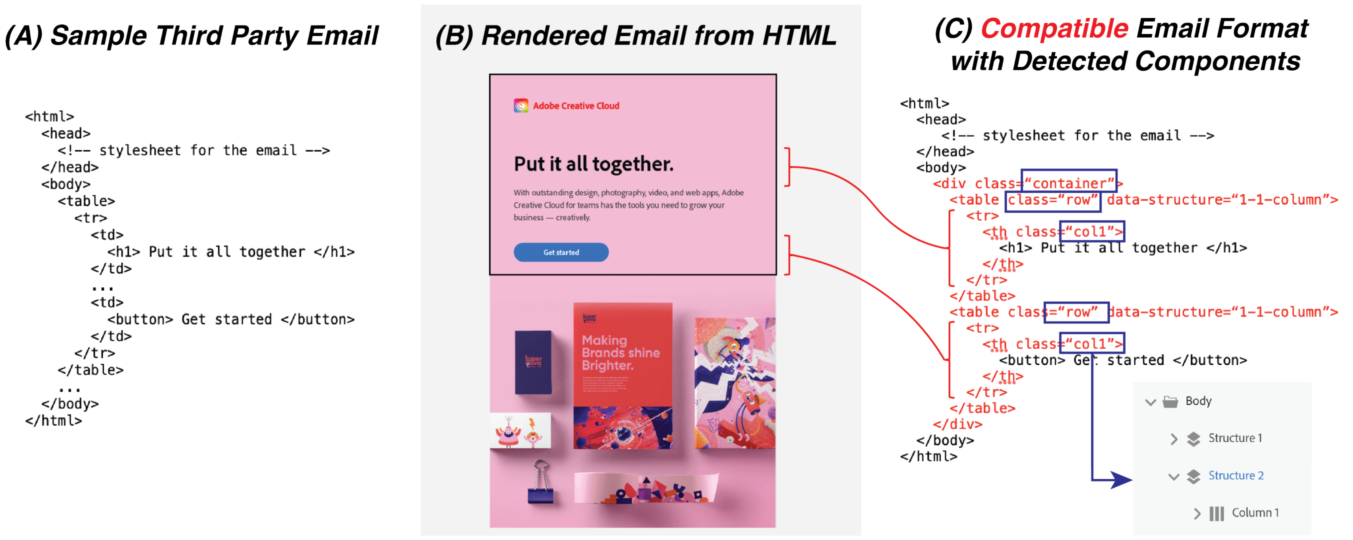
1 INTRODUCTION

Nowadays, we receive many business emails from different companies for purposes related to purchases and news. These emails are often designed by editors that follow different HTML formatting guidelines [1–6]. Although the rendered emails can be easily shared among different browsers and mailboxes (Figure 1(B)), they cannot be edited by other editors due to different formats to construct the HTMLs. Most of them contain predefined components, such as rows, columns, text, images, and buttons (Section 2.1) so that designers can create and edit emails without coding the HTML. Therefore, to facilitate the usage of email editors on third-party emails, we need to detect these components to wrap them with the editor-specific HTMLs (Figure 1(C)).

Developing a model in an industrial setting, we encountered a human-centered challenge. To integrate the detection and conversion of third-party emails as a feature in a commercial editor, a team of researchers, engineers, and product managers need to collaborate together. Since different stakeholders have different backgrounds, expertise, and requirements on the model, we discovered that a "collaborative" model development needs to be *accessible*, *extensible*, and *scalable*. Accessible means everyone should understand how the model works. Extensible means a developer can add rationale to the detection easily. And scalable means it can leverage ML to improve accuracy on many emails.

As a result, we propose an email component detection model using Answer Set Programming (ASP) [8]. We represents HTML emails as a set of logical facts and detects the components with hard and soft constraints over these facts. As we will show in Section 3.1, these constraints are concise and human-readable sentences. Thus, we view them as a human-readable model with *deterministic* and *probabilistic* objectives. Hard constraints must be satisfied (e.g., rows and columns cannot overlap), whereas soft constraints express a preference (e.g., an image HTML tag looks like an image component). Our contribution is not to propose a more accurate model to detect email components. Instead, we propose ASP to motivate an expressive and extensible email component detection model development. We believe it could potentially apply to other use cases that involve HTML [16–19].

In general, ASP has an extensive use in various applications, ranging from GUI layout [14, 15], bioinformatics [12, 13], robotics [10], information integration [7, 20], to scheduling [22]. Prior research has also broadly explored the use of ASP in classification problems. In this work, we apply ASP for detecting the roles of HTML components in business emails as human interpretable


(A) Sample Third Party Email
(B) Rendered Email from HTML
(C) Compatible Email Format with Detected Components
Adobe Creative Cloud
Put it all together.
With outstanding design, photography, video, and web apps, Adobe Creative Cloud for teams has the tools you need to grow your business — creatively.
Get started
Making Brands shine Brighter.
Body
Structure 1
Structure 2
Column 1

(A)

```
<html>
  <head>
    <!-- stylesheet for the email -->
  </head>
  <body>
    <table>
      <tr>
        <td>
          <h1> Put it all together </h1>
        </td>
        ...
        <td>
          <button> Get started </button>
        </td>
      </tr>
    </table>
    ...
  </body>
</html>
```

(C)

```
<html>
  <head>
    <!-- stylesheet for the email -->
  </head>
  <body>
    <div class="container">
      <table class="row" data-structure="1-1-column">
        <tr>
          <th class="col1">
            <h1> Put it all together </h1>
          </th>
        </tr>
      </table>
      <table class="row" data-structure="1-1-column">
        <tr>
          <th class="col1">
            <button> Get started </button>
          </th>
        </tr>
      </table>
    </div>
  </body>
</html>
```

Figure 1: Goal of Email Detection: Given a HTML email (A), we detect the components (e.g., rows, titles, buttons) to inject formatted HTML (C) and make it editable in an email designer with the same display (B). Example format in (C) includes putting each text and button into separate `<table>` with class names `row` and `col`.

logic-based AI [11]. Our contributions focus on the readability and expressiveness of the framework that allows hand tuned constraints.

2 BACKGROUND

2.1 Email Components

A business email editor usually enables the design through "drag-and-drop" functionalities. Based on email editors from Adobe [1], Bee [2], Chamaileon [3], Mailchimp [4], Salesforce [5] and Stripo [6], we summarize two types of components provided to design HTML emails without needing to code:

- *Layout Components*: To place the text and images into desired regions in the email, the user first need to specify the "grid". To do so, the editors often provide a "row" component to segment the vertical regions and a "column" component to segment the horizontal regions.
- *Design Components*: In each of the cells in the grid, the user can drag a specific email design. The typical options could be summarized as follows: *Text*: for inserting titles or paragraphs; *Multimedia*: for inserting images or videos; *Button*: for inserting buttons to external websites; *Divider*: for inserting page breaks in the email; *Social*: for inserting links to social media websites, and; *Offer*: for inserting personalized links to product websites.

Originally, to identify these components in an imported emails, each email editor will assign related `<class>` names to the related HTML in the email (Figure 1(C)). Therefore, it is challenging to identify these structures from an email not designed by its native editor since different editors have different names and elements assignments. In addition, the same layout and visual designs could have different HTML specifications. For example, a row with three columns can be achieved by either a `<table>` or `<div>` tag, and a button can be designed with either a `<button>` or `<a>` tag. As long as some minimal HTML rules are obeyed, different HTML emails can provide the same visuals.

2.2 Model Development Process

To deploy a detection model in a commercial email editing platform, our team consists of researchers, developers, and product managers. The development is a holistics and collaborative process and heavily human-in-the-loop. To begin with, we started with a small set of business emails. The whole team brainstormed the characteristics of the components to verbally describe them. Then, we translated them into ASP rules and tested the emails by inspecting the results in the email editor. After several iterations, the product manager introduced more emails from different customers. Since different customers had different ways to create emails, we brainstormed again for any issues raised from the new data and implemented new rules to address them. Thus, ASP played an useful role due to the following reasons:

- *Easy mapping between rules syntax and human descriptions.* During the discussion, we often had ideas like "*large fonts must be titles*" or "*company A likes to put links into images*". As we will show later (Section 3.1), these comments could be easily mapped to concise one-line syntax in ASP.
- *Version Control.* Since each line of the ASP often represent one rule, we can easily see different versions using file comparison tools to compare the performances between two versions of models from different considerations.

3 ASP EMAIL COMPONENT DETECTION

3.1 Answer Set Programming

Readers might refer to the literature for a more detailed review of ASP [8]. We only describe the functionality of ASP that was used in our work. To understand how ASP searches answers over constraint-based problems as a modeling language, we need to understand the three building blocks of the program: *atoms*, *literals*, and *rules*. *Atoms* are factual statements that may be true or false.

Literals are atoms a and their negations *not* a. *Rules* are the expressions of the logic. For example, in the rule: `a :- b, not c`. The definition states that `a`, `b`, and `c` are all atoms. Also, it justifies that `a` (the so-called head) is true if all literals to the right (the so-called body) are true if a non-negated literal `b` has a derivation and the negated literal `c` does not have one. Noted that the negation will result in a true condition when either the atom is true or the atom is not declared. This is useful for detecting email elements without complete specification of the whole feature space. For example, consider a following rule that determines whether a HTML is a text fragment:

```
fragment(text) :- hasText, not backgroundColor(white).
```

Suppose when we extract atoms from a HTML fragment and there are no background colors declared, the head `fragment(text)` will still be true if we can generate an atom `hasText` from the HTML. The atoms with brackets are the ones that could contain variables [1], which is useful for us to get answers from constraints and optimization results with repeatedly defined atoms.

The next important property for ASP is that the rules do not need a head (i.e. bodiless). For example, a rule can be stated as `:- a, b`. In this case, such a rule is called *integrity constraint*, in which `a` and `b` cannot be satisfied together. This is useful when we want to prevent some atoms from happening at the same time. For example, `:- isElement, tag(tr).` allows us to set the `isElement` flag to false when the tag of the current HTML fragment's tag is `tr`.

In addition to the constraints inside rules, the constraints can be reached to level of multiple rules. In other words, the rules `a0`, `a1`, ... `an` can be *aggregated* as j `{a0, a1, ... an}` k.: meaning at least j and at most k atoms must be true. For example, if we want to restrict the answer (e.g. fragment type) to one only, we can use such a syntax `1 { fragment(F), fragmentType(F) } 1.` Also, there are other aggregation available such as `#sum` and `#count`. For example, if a fragment has three children's widths as `childrenWidth(1,10)`, `childrenWidth(2,30)`, and `childrenWidth(1,40)`, we can define a rule to find out their total width as:

```
totalWidth(W) :- W = #sum{ CW : childrenWidth(_,CW) }.
```

Noted that the underscore symbol `_` implies an anonymous variable so that the rule will not constraint its value.

Until this point of discussion, we might treat ASP as a neat expression for "`IF-THEN-ELSE`" statements to arrive at answers through elimination. Yet, another useful feature from ASP is the *soft constraint*, which we can incur *preferences* when searching for answers. *Soft constraints* are in the syntax of `:~ a, b. [`w`]`. The rule is interpreted as follows: if the literals `a`, `b` are satisfied, we will impose a penalty equal to the weight w. Across a set of soft constraints in a program, we look for an optimal answer set that minimizes the sum of weighted costs of all satisfied soft constraints.

[1]The variable that starts with a capital letter is a numerical variable.

3.2 Pipeline

The whole pipeline of the detection model is as follows. First, we view the HTML email as a hierarchical DOM tree. We recursively parse the children's elements along the tree and see whether they belong to the defined structures. We first detect the **rows** in the email, then followed by **columns**, and **elements** at the end. The parsing includes an extraction of ASP atoms from the HTML (Section 3.3) and running our pre-defined rules with the atoms (Section 3.4). Once the parsing is finished, the detected HTML fragments will be extracted and fit into the templates that are recognized by our own email editor.

3.3 Mapping HTML Specifications to Facts

An HTML specification of an email fragment describes the arrangements and styles of a set of web contents. The tags usually show some characteristics of the web content, such as whether it contains a website link, text, or image. Also, the whole HTML is accompanied by an embedded style sheet that defines the styles like background color and positions of the web components. Noted that for business emails, they work as standalone files that do not need external files such as CSS or Javascript to display the contents.

In addition, we can also extract the features of the emails in their *rendered* format. By using a browser to render the HTML, we can acquire the statistics of the appearance of the business email, such as the width, height, and position. Also, we can format the appearance as pixels so that we can use Computer Vision libraries to extract colors and fonts. Together, we summarize the features extracted from a HTML fragment in Table 1.

3.4 Detecting Email Components with Constraints

With the defined features and facts, we now describe the rules and constraints for different detection purposes in the pipeline. Our goal is to have a *single* ASP program handling most of the routines in the detection tasks so that we do not need to manage multiple programs through external tools. Thus, our constraints can be categorized into three categories: (1) task and configurations, (2) email structure detection, and; (3) email element detection.

3.4.1 *Constraints for setup and configurations.*

The first set of constraints aims to define the detection tasks so that we can turn off some rules that are not related to the current task. It is important since some rules might affect the cost function and, consequently the optimization results. Also, there are requirements on the results' formats for each task. To begin with, we can define the task constraints:

```
taskType(detectRows;detectColumns;detectElements).
1 {task(T): taskType(T) } 1.
```

By inputing a `task` atom to declare the task, we can control the output of the ASP program. For example, if we have designed a set of rules to determine email elements with syntax "`element(button) :- ...`", we can control the activations of these rules by associating the atom `element` with the atom `task(detectElements)`:

```
0 {element(T): elementType(T) } 1.
:- element(_), not task(detectElements).
```

Table 1: The table shows some basic extracted ASP atoms from the emails.

Source	Atom's Definition	Description	Usage Example
HTML Parsing	`tag(<string>)`	The tag contained in the email fragment.	`tag(img).`
	`href(<string>)`	The domain of the website link inside the fragment.	`href(google).`
	`nText(<int>)`	The number of texts inside the fragment.	`nText(10).`
	`nParagraph(<int>)`	The number of text paragraph inside the fragment.	`nParagraph(2).`
Browser Rendering	`size(<int>,<int>)`	The size (width and height) of the fragment.	`size(100,50).`
	`pos(<int>,<int>)`	The position (x and y) of the fragment.	`pos(0,0).`
	`childrenSize(<ID>,<int>,<int>)`	The `i`-th child element's size in the fragment.	`childrenSize(1,50,20).`
	`childrenPos(<ID>,<int>,<int>)`	The `i`-th child element's position in the fragment.	`childrenPos(1,0,0).`
Pixel-level Extraction	`bgColor(<int>,<int>,<int>)`	The RGB values of the fragment's background color.	`bgColor(255,255,255).`
	`fontSize(<int>)`	The font size of the text in the fragment.	`fontSize(12).`

The first rule tells us the number of `element` heads in the answer set is either one or none, and the second rule tells us any atoms related to `element` should not appear if `task(detectElements)` is not declared, which implies all optimization rules and costs related to `element` will not be realized.

In addition, for email conversion, we also declare some HTML-specific conditions that violate the detection tasks. Recall our goal is to extract an HTML fragment and copy it inside our editor-specific structures. Such a strategy will violate the email formatting if the fragment begins with a tag that requires its parent tag to be a specific one. For example, a `<th>` fragment must have a `<tr>` tag as its parent. To address the violation, we can declare a `invalidTag` head when the tag falls into a set of definitions:

```
invalidTag :- tag(tr;th;...).
:- element(_), invalidTag.
```

Notice that the second rule does not need to concatenate to the rule related to the similar one with `task`. Such an extensible representation allows us to read the constraints in separate lines, which is also useful for version control when multiple developers work on the same Git repository.

3.4.2 *Constraints to Detect Rows and Columns.*

To detect structures in the HTML email, we can divide the tasks into the detection of rows followed by the detection of columns. Both tasks follow the same routine, and the only difference is that the row detection tries to split the HTML fragments vertically, and the column detection tries to split them horizontally. We use row detection as the illustration. To start with, we can treat the email structure as a set of hierarchical rectangular partitions. The challenges are that the row structures might exhibit at different levels in the hierarchy, and the formulation of detecting non-overlapping row partitions can easily turn into codes and models that are hard to interpret among different developers.

We formulate the row detection routine into the detection of row assignments of children nodes under the same parent HTML node. Among the children nodes, we use ASP to optimize the rows assignment. Afterward, we recursively replace the nodes in the assignment with their descending nodes if the node can be further split into *more than one row*. Algorithmically, it implies the supply of $n-1$ more row partitions to the parent node where n is the number of row partitions in the current node. As a result, the final row assignment will be available on the root level.

Recall that each children's node's ID, size, and position are defined in `childrenSize` and `childrenPos` in Table 1. We define the atoms that represent the labels of nodes and row partitions:

```
children(1..CID) :- CID = #max{ID: childrenPos(ID,_,_)}.
row(1..RID) :- RID = #count{ ID: children(ID) }.
```

Noted that we declare a constraint related to row assignment in the second rule above, where the number of row partitions cannot exceed the number of children nodes. To collect the row assignment, we can define an atom `rowAssignment(<CID>,<RID>)` to represent the answers in the program output:

```
1 { rowAssignment(CID,RID): row(RID)} 1 :- children(CID).
```

This also provides a constraint for the program that each children node can only belong to one row.

After the constraints related to the answers and variables are defined, we can proceed to the constraints related to the row detection. In a "constraint" mindset, we determine the row assignment by telling the program *what could not happen for a row assignment.* Since our goal is to determine non-overlapping partitions and their dimensions are determined by the children inside the partitions, the two cases that the partitions will overlap are when (1) the *upper row's bottom Y* coordinates are larger than the *lower row's top Y* coordinates, and; (2) the *upper row's bottom Y* coordinates are larger than the *lower row's bottom Y* coordinates, assuming the Y coordinates increases from top to bottom. To express these into ASP, we can define rules that align with the reasoning below.

The following condition cannot happen:

```
:-
```

For any two different children in different rows,

```
rowAssignment(CID1,RID1), rowAssignment(CID2,RID2),
CID1 != CID2, RID1 != RID2,
```

regarding their heights and starting Y coordinates,

```
childrenSize(CID1,_,H1), childrenPos(CID1,_,Y1),
childrenSize(CID2,_,H2), childrenPos(CID2,_,Y2),
```

the upper child's bottom Y is larger than the bottom child's top Y.

```
(H1+Y1)>Y2, Y2>Y1.
```

< Similar syntax for the second row related constraint >

...

the upper child's bottom Y is larger than the bottom child's bottom Y.

```
(H1+Y1)>(H2+Y2), Y2>Y1.
```

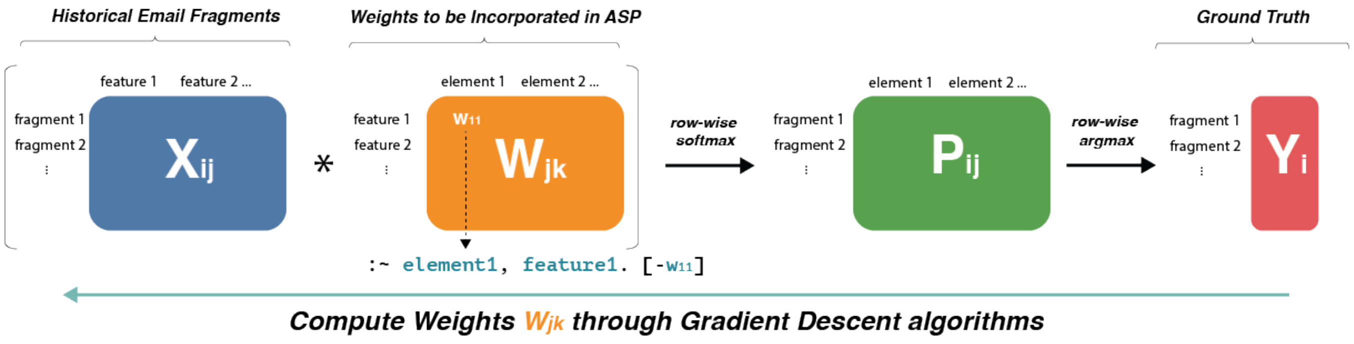


Figure 2: An illustration of learning weights for soft constraints using historical email fragments.

The expressiveness of ASP allows describing a constraint with a similar ordering of atoms in the literal. With the four constraints defined, the program arrives at a set of legitimate row partition results. To arrive with a final answer that divides most vertical spaces, we can define a head atom that records the number of row partitions used and define the *maximization the number of row partitions as the objective function*:

```
numberOfRows(N) :- N = #count{RID: rowAssignment(_,RID)}.
#maximize { N: numberOfRows(N) }.
```

The first rule counts the number of unique row IDs in the row assignment, and the second rule optimizes the row assignment that leads to the largest number of rows defined in the head.

After the row detection of children nodes is completed, we can prepare for the same routine for the descendants among the children nodes. Noted that if a row has accommodated more than one children nodes, we do not need to continue the row detection routine on these children since they already overlap with each other. We can detect this behavior with one rule based on our current atoms:

```
continue(CID) :- N = #count{I: rowAssignment(I,RID)}, rowAssignment(CID,RID),
N <= 1.
```

The `i`-th children will appear an answer `continue(i)` only if the number of `rowAssignment` that contains the same `RID` as `i` is greater than one.

3.4.3 Email Element Detection with Soft Constraints. While the row and column detection tasks are deterministic since we understand our goals to partition the emails, we now define a scoring mechanism to map a custom email element's characteristics into an integer representing its preference level. This mechanism allows us to rank the email element answer without defining the scoring as a procedure. Instead, we can use a set of soft constraints to implicitly define it. The weights of the constraints reveal their favor and strengths: the sign of weight affects the favor (i.e., negative weights represent favor towards the characteristics and vice versa), and the magnitudes represent the importance.

The syntax of soft constraints is similar to a bodiless rule, except the beginning starts with `:~` instead of `:-`. For example, `:~ element(image), hasImage. [-3]` states that defining the element as an image component while the HTML fragment contains an image will decrease the cost of the optimization function by three. Therefore, we do not need to set a hard threshold to have every HTML fragment with an image as an image component. It is sometimes reasonable to define the fragment as something else.

To extend our ASP program to support new email components, a developer can add soft constraints to capture the desired preferences. For example, if the email editor supports a carousel element with multiple image containers, we can add the soft constraints to "encourage" the choice of carousel and "discourage" the choice of image in the answer set below.

```
manyImages :- numOfImages(N), N > 1.
:~ element(carousel), manyImages. [-1]
:~ element(image), manyImages. [1]
```

Noted that in this example, the rationale behind the detection result as carousel or image are the same. Thus, the evaluation can be treated as a multiclass regression model. Concretely, given a set of classes $Y = \{y_1, y_2, ...\}$ (e.g. `element(text)`, `element(image)`, ...), a set of features derived from the HTML fragment as a feature vector $x = \{x_1, x_2, ...\}$ (e.g. `manyImages`, `hasText`, ...), the resulting class i is derived as follows:

$$\operatorname*{argmin}_i x^T w_i$$

where w_i are the weights associated with the soft constraints related to the class (e.g. `-1` and `1` in the above example). The setting of these weights can be done by an email editor and expert through trial and error among different trade-offs. In Section 4, we will also demonstrate how we can learn w from data.

4 USING MACHINE LEARNING TO IMPROVE DETECTION RESULTS IN ASP

Although designers can tune the weights to improve the detection performance on their own, the ASP model can also leverage Machine Learning to learn the weights of soft constraints from available emails in the inventory. Email fragments usually contain labels that indicate their component assignments, such as having a class called `btn-` or `image-` in their tags. Better still, we can acquire strictly defined components from emails that are created from our designer software. With such a resource, we can develop a Machine Learning pipeline to acquire the weights (Figure 2). We first

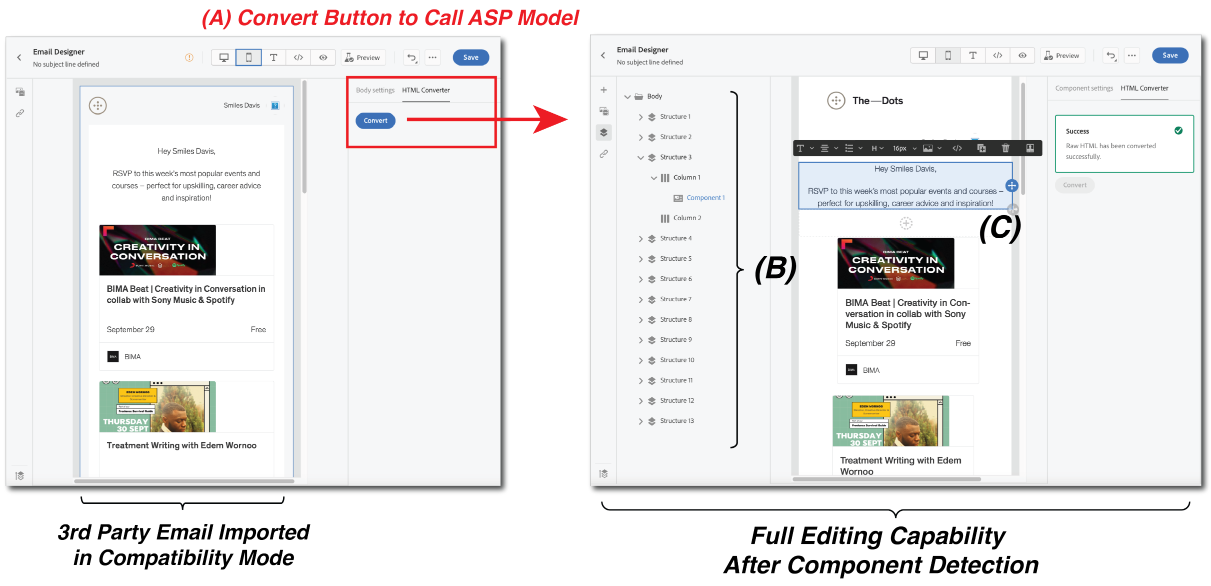


Figure 3: Email designer with detection model. (A) A converter button enables the conversion of third-party email. (B) After the conversion, the rows and columns are detected. (C) The text element is identified and allows the user to edit the content.

map the HTML fragments from the training samples individually to atoms in ASP and then run the program to count how many features the atoms satisfy. The counts from all the training samples can be transformed into a $i \times j$ matrix X where i is the number of samples, and j is the number of features defined in our ASP. The labels can be transformed to a vector Y_i. Using our soft constraint rules defined in Section 3.4.3, the weights W can form a $j \times k$ matrix where k is the number of email elements available in the designer software. With the three matrices X, W, and Y, we can use a multiclass logistics regression model to compute the values of weights W^* through gradient descent algorithms [21].

Since the pipeline in Figure 2 demonstrates learned weights using a backpropagation approach, we can foresee that the soft constraints can also be extended to non-linear models like neural networks. The only addition rules that define the activation functions and additional weights multiplications in the hidden layer. However, we focus on developing an expressive model for email component detection in our work. We will leave the usage of non-linear models in future discussions. To conclude, by integrating the learned weights, the ASP model becomes a knowledge base for email component definitions that combines both the designer knowledge and empirical data.

5 DEMONSTRATION AND EVALUATION

We present the usage of our ASP model by demonstrating our deployment in a real email editor platform. By showing the input and output of the system, we demonstrate how our model is useful for inspection and debugging. Then, we introduce a use case where we are able to use learning from data to improve the performance of distinguishing between a button and a text in the emails, which are similar in terms of their HTML specifications. We use publicly available emails from https://reallygoodemails.com/ for demonstrations and experiments. We have crawled 980 emails from the website and extracted around 3,000 fragments for Section 5.3.

5.1 Implementing ASP Model in an Email Designer Platform

We demonstrate the new user experience enabled by our email component detection model (Figure 3). When we import an email from a third-party source, the interface will display in a "compatibility mode", meaning the email would be only available for browsing only. However, with our component detection technologies, we could now provide a button on the interface to allow users to convert the email into our editor's format (Figure 3(A)). Therefore, the conversion results in two benefits in the system. First, the rows and columns are identified so that the user can easily drag and drop the components in the editor (Figure 3(B)). Second, since the elements are also identified, the user can also edit the content with native styles provided by the editor as well (Figure 3(C)).

≈

For the backend, the input and output of ASP provides an easy interpretation for understanding and debugging. For example, a button HTML fragment is compiled into a set of facts and constraints when the software sends the third-party email to the model:

```
task(detectelement). textColor(255, 255, 255). tag(div).
tag(a). tag(span). width(148). height(28). fontSize(18).
borderWidth(4). urlLength(1). numberOfTexts(2).
```

Our ASP model then returns the following answer set:

```
   fact(smallComponent). fact(hasBorder). fact(hasText).
fact(smallText). fact(lightColorText). element(button).
```

For the purpose of inspection and debugging, not only can the output be obtained through `element(button)`, but the constraints

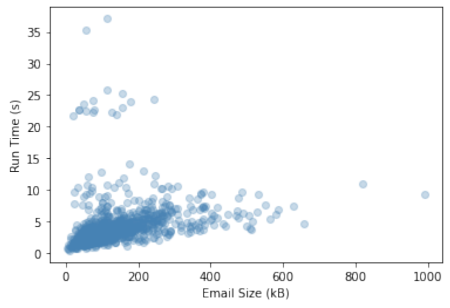


Figure 4: Run time performance on 980 emails.

that are compiled and used by the ASP are also shown through the fact-related atoms. Thus, the developer can easily understand the logic behind a detection with short and concise statements.

The whole conversion process is also very efficient. Figure 4 shows the computation time for the 980 emails, which involves the feature extraction, ASP model compilation, and transformation of the detected email to our designer's format. The whole process takes below 30 seconds for most emails, which is important to prevent time out in our software's architecture.

5.2 Evaluating Layout Components on Real Emails

While the component detection on the 980 emails is efficient, we are also interested in understanding whether the layout component detection works or not. We hired three freelancer with HTML design experience from UpWork [2]. We asked them to provide feedback by visually inspecting the email appearances before and after the detection/conversion. We also hoped to summarize the limitations of the current detection model. Overall, each of them agreed that more than 800 emails had their layout well segmented with little distortion. While the model could be continuously improved throughout the development, we outlined several challenging issues in the corner cases:

I. *The whole email is an image.* There are emails that are designed mainly by image editors so that the whole email only contains an image tag. In this case, our detection model will only return one row and column and an image component.
II. *Column first emails.* Our framework assumes the HTML first segment the email by rows then columns. For emails that do the opposite, the vertical segments will be aggregated into one row only.
III. *Highly responsive emails.* Some emails are designed with lots of responsive components for different device resolutions. For example, some images will appear only when shown on mobile phones but not on desktops. The DOM tree thus contains many "invisible" tags that could potentially break the partition after the detection.

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

Table 2: Detection Accuracy from the Soft Constraints.

	Hand-crafted ASP Model	Learned ASP Model
Text Fragment	0.249	0.875
Button Fragment	0.975	0.852

5.3 Applying Machine Learning to Detect Similar Elements

We now demonstrate how to incorporate Machine Learning to improve detection performance. To make the discussion simple, we demonstrate how to distinguish between a button and a text fragment in an HTML fragment. We acquire a sample for ML by extracting the HTML fragments that contain text- or button- as the respectively ground truths. There are 1,982 text fragments and 1,145 button fragments as a result. We split them into 80/20 for training/testing. We compile the facts from the ASP program and construct a matrix similar to the X in Figure 2 to learn a logistic regression model. The detection accuracy (i.e., number of correct predictions/number of total samples) for the learned model and hand-crafted ASP model are shown in Table 2. We can see a significant improvement in detection quality. Without knowing the importance of each preference, the hand-crafted model seems to have favor in identifying most fragments as buttons, which leads to a huge sacrifice on the text components (i.e., the model just turns everything into buttons). After adjusting the weights on the soft constraints, we can rectify the judgment to improve the accuracy. While the machine learning models participate in the improvement, our ASP model remains expressive and easy to understand with changes to the weights only (highlighted in red):

```
:~ soft(hasText), element(text). [-1] →[-4].
:~ soft(longParagraph) ,element(text). [-1] →[-21]
:~ soft(shortParagraph) ,element(text). [1] →[14]
:~ soft(hasHref), element(button). [-1] →[-26].
:~ soft(lightColorText), element(button). [-1] →[-10].
:~ soft(smallText), element(button). [-2] →[-5].
:~ soft(largeComponent), element(button). [2] →[2].
```

The learned weights also reflect the importance of each feature to the detection result. Thus, the weights can act as a profile for developers to understand the context of email design.

6 LIMITATIONS AND FUTURE WORK

6.1 Beyond Email Component Detection

Our detection framework only works on detecting components in business emails. However, we believe that the whole framework can be applied to different classification tasks on HTML files. For example, fake news detection [9] and visual designs [23] can both benefit from our approach. The human-centered approach with ASP allows users to incorporate their knowledge to eliminate or quickly identify obvious cases while leveraging Machine Learning to address more challenging scenarios. With both deterministic rules and probabilistic models expressed with the same syntax, humans can easily maintain a large knowledge base and incorporate with models to achieve more interpretable and robust models in real-life applications.

6.2 Detection on Creative Content

Our email design component detection framework can be easily adapted into other creative content and use cases, such as transforming PNG designs into Photoshop formats. The only difference is the mapping between pixels to logical facts. With today's evolving technologies from Computer Vision, we can generate features that could be transformed into facts in ASP to achieve similar objectives. We are interested in applying our detection approach to a broader set of creative contents.

6.3 Constraints from a Software Engineering Perspective

Our use of constraint programming provides easier development and maintenance of automated email component detection tools. We propose concise expressions to address two challenges: hard-to-maintain codes to express conditions in a complex scenario and hard-to-collaborate black box models. Our work provides an example to incorporate human knowledge in an extensible way. For example, if the email editor software decides to introduce an additional component, the developers can add rules to accommodate the unseen tasks. These rules lie on separate lines independently in the program so that everyone can track the changes easily through version control. In the future, we are interested in developing a collaborative platform from ASP so that developers and designers can synchronously address a task with ASP together.

7 CONCLUSION

This paper presents an email component detection model in Answer Set Programming (ASP). The model consists of hard and soft constraints to detect the layout and features in the HTML email. We illustrate its expressiveness by presenting the implementation as human interpretable statements. Finally, we demonstrate the model as a production ready feature in a commercial email editor.

REFERENCES

[1] 2022. *Adobe Email Editor.* https://experienceleague.adobe.com/docs/campaign-standard/using/designing-content/building-email-content/designing-from-scratch.html Accessed: 2022-09-15.

[2] 2022. *Bee Email Editor.* https://beefree.io/ Accessed: 2022-09-15.

[3] 2022. *Chamaileon Email Editor.* https://chamaileon.io/email-design-systems/ Accessed: 2022-09-15.

[4] 2022. *Mailchimp Email Editor.* https://mailchimp.com/help/design-an-email-new-builder/ Accessed: 2022-09-15.

[5] 2022. *Salesforce Email Editor.* https://help.salesforce.com/s/articleView?id=sf.email_template_builder_create.htm Accessed: 2022-09-15.

[6] 2022. *Stripo Email Editor.* https://stripo.email/en/demo/ Accessed: 2022-09-15.

[7] Massimiliano Albanese, Matthias Broecheler, John Grant, Maria Vanina Martinez, and VS Subrahmanian. 2011. PLINI: a probabilistic logic program framework for inconsistent news information. In *Logic programming, knowledge representation, and nonmonotonic reasoning*. Springer, 347–376.

[8] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. 2011. Answer set programming at a glance. *Commun. ACM* 54, 12 (2011), 92–103.

[9] Sonia Castelo, Thais Almeida, Anas Elghafari, Aécio Santos, Kien Pham, Eduardo Nakamura, and Juliana Freire. 2019. A topic-agnostic approach for identifying fake news pages. In *Companion proceedings of the 2019 World Wide Web conference*. 975–980.

[10] Esra Erdem, Erdi Aker, and Volkan Patoglu. 2012. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics* 5, 4 (2012), 275–291.

[11] Esra Erdem, Michael Gelfond, and Nicola Leone. 2016. Applications of answer set programming. *AI Magazine* 37, 3 (2016), 53–68.

[12] Esra Erdem and Umut Oztok. 2015. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming* 15, 1 (2015), 35–78.

[13] Esra Erdem and Ferhan Türe. 2008. Efficient Haplotype Inference with Answer Set Programming.. In *AAAI*, Vol. 8. 436–441.

[14] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC layout: Adaptive GUI layout with OR-constraints. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[15] Yue Jiang, Wolfgang Stuerzlinger, Matthias Zwicker, and Christof Lutteroth. 2020. ORCSolver: An Efficient Solver for Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–14.

[16] Mohd Nizam Kassim, Mohd Aizaini Maarof, and Majid Bakhtiari. 2020. Fraudulent e-Commerce Website Detection Model Using HTML, Text and Image Features. In *Proceedings of the 11th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2019)*, Vol. 1182. Springer Nature, 177.

[17] Yukun Li, Zhenguo Yang, Xu Chen, Huaping Yuan, and Wenyin Liu. 2019. A stacking model using URL and HTML features for phishing webpage detection. *Future Generation Computer Systems* 94 (2019), 27–39.

[18] Seung-Jin Lim and Yiu-Kai Ng. 2001. An automated change-detection algorithm for HTML documents based on semantic hierarchies. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 303–312.

[19] Sonal Mahajan and William GJ Halfond. 2015. Detection and localization of html presentation failures using computer vision-based techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[20] Marco Manna, Francesco Ricca, and Giorgio Terracina. 2013. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory and Practice of Logic Programming* 13, 2 (2013), 227–252.

[21] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[22] Torsten Schaub and Stefan Woltran. 2018. Special issue on answer set programming. *KI-Künstliche Intelligenz* 32, 2 (2018), 101–103.

[23] Yudong Yang, Yu Chen, and HongJiang Zhang. 2003. HTML page analysis based on visual cues. In *Web Document Analysis: Challenges and Opportunities*. World Scientific, 113–131.