

Fault Localization in Web Applications via Model Finding

Sylvain Hallé, Oussama Beroual

Laboratoire d’informatique formelle
Université du Québec à Chicoutimi, Canada

We describe a generic technique for fault localization independent from the nature of the object or the specification language used to declare its expected properties. This technique is based on the concept of “repair”, a minimal set of transformations which, when applied to the original object, restores its satisfiability with respect to the specification. We show how this technique can be applied with various specification languages, including propositional and finite first-order logic. In particular, we focus on its use in the detection of layout faults in web applications.

1 Introduction

The use of the web has seen significant changes since the inception of HTML in the early 1990s. The web has now become a large and mature software ecosystem on par with the complexity found in traditional desktop applications. However, due to the somewhat complex relationship between HTML, CSS and JavaScript, the layout of web applications tends to be harder to properly specify in contrast with traditional desktop applications. The same document can be shown in a variety of sizes, resolutions, browsers and even devices, making the presence of so-called layout “bugs” all the more prevalent. Such problems can range from relatively mundane quirks like overlapping or incorrectly aligned elements, to more serious issues compromising the functionality of the user interface.

In past work, we developed an automated tool for the detection of such bugs, evaluating expressions in a high-level declarative language based on first-order and linear temporal logic [3]. However, it was recognized early on that the basic evaluation of such properties, returning a simple true/false verdict, would not prove very helpful to a designer: web pages are composed of hundreds of elements with dozens of properties each, and over which many assertions are required to hold. What is more, sometimes the layout faults are too subtle to be visible to the naked eye (such as elements off by a single pixel). To provide real value to practitioners, a layout analysis tool should hence be able to pinpoint specific elements of the page that are responsible for some bug. Our work on layout bug detection has therefore turned into a form of *fault localization*.

In Section 2, we first show examples of layout bugs in real-world web applications, and describe an early attempt at building an explanation for the violation of a specification in that context. This process builds what is called a “witness”; its construction is based on a function applied recursively on the *formula* that is falsified. A witness highlights a set of elements in the page which, in some way, are related to the violation of a property.

This has proved insufficient in practice; therefore, in Section 3, we describe ongoing work on a new formal grounding, this time based on the concept of “repair”. Intuitively, a repair is a minimal set of transformations which, when applied to the original object, restores its satisfiability with respect to the specification. The advantage of this concept is that it is, at its highest level, independent from the nature of the object and the specification language used to declare its expected properties. It could hence be applied to a variety of other scenarios, besides web applications. Section 4 further discusses the concept of repairs, and provides a basic algorithm for computing them. Based on a few examples, we show how it

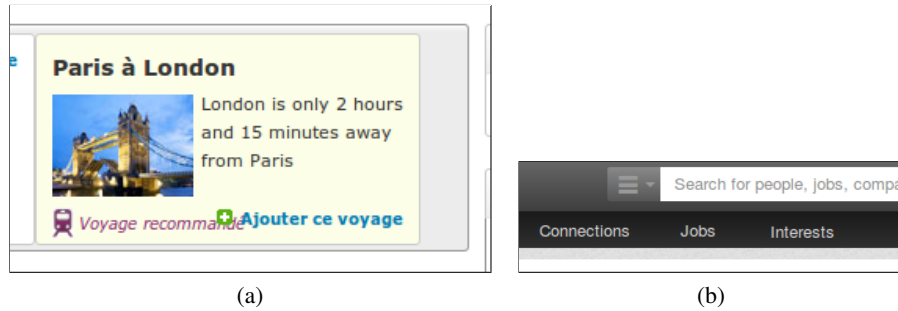


Figure 1: Two examples of real-world layout bugs in web applications: (a) overlapping elements; (b) incorrectly aligned elements.

presents the potential of improving the task of fault detection and correction, by automatically providing hints that correspond to intuition.

2 Fault Localization in Web Applications

A *layout-based bug* is a defect in a web system that has visible effects on the content of the pages served to the user. This content can be anything observable by the client, including the structure of the page’s elements and the dimensions and style attributes of these elements. In past work, we studied bugs in web applications that can be detected by analyzing the contents and layout of page elements inside a browser’s window. Based on an empirical analysis of 35 real-world web sites and applications (such as Facebook, Dropbox, and Moodle), we provided a survey and classification of more than 90 instances of layout-based bugs.

2.1 Examples of Layout Bugs

Figure 1 shows two examples of frequent layout bugs we encountered. In the first screenshot, two elements of the page that should be disjoint are actually overlapping. In that particular instance, the problem is caused by the fact that elements are absolutely positioned in a page with respect to their dimensions when the text they contain is in English. When displaying the web site in another language (such as French), it may occur that the corresponding text is longer than the English version, causing two elements that were disjoint to suddenly overlap.

Another mundane but frequent layout problem is the presence of elements which visibly should be aligned but are not. Figure 1b shows an instance of this bug for the web platform LinkedIn. Sometimes, the misalignment is subtle, as an element may be off by a single pixel, as is the case here.

We then developed Cornipickle, an automated testing tool that provides a declarative language to express desirable properties of a web application as a set of human-readable assertions on the page’s HTML and CSS data [3]. Such properties can be verified on-the-fly as a user interacts with an application.

The core of the Cornipickle language is a high-level, English-like grammar that translates in the background into first-order linear temporal logic. For example, one can express that all elements of some list with ID “menu” should be vertically aligned by writing the following expression:

- | | |
|--|--|
| <ul style="list-style-type: none"> • A list item • Another list item • A third list item • The last list item <p style="text-align: center;">(a)</p> | <ul style="list-style-type: none"> • A list item • Another list item • A third list item • The last list item <p style="text-align: center;">(b)</p> |
|--|--|

Figure 2: Example of a simple web layout fault: (a) one of the list items is incorrectly aligned with the others; (b) a witness produced by the Cornipickle tool.

```

For each $x in $(#menu li) (
  For each $y in $(#menu li) (
    $x's left equals $y's left
  ))

```

Here $\$(\#menu li)$ is called CSS *selector*. A selector is a filter expression designating a set of elements in a web page, as defined in the CSS specification [1]. For example, the expression $p > li.foo$ designates all elements with name `li` and class attribute `foo` immediately nested within an element of name `p`. The syntax $\$x's left$ is used to refer to the CSS `left` property of an element, which designates the absolute x coordinate of its top-left corner within the page. For this particular example, Figure 2a shows a simple page for which the property would be violated.

2.2 Witnesses

Cornipickle can easily evaluate properties of this kind on the contents of a page and report a true/false verdict. However, while in some cases the reason for the violation of a property is obvious, in many others a mere Boolean outcome bears little meaning for a web designer. After all, some errors can be subtle, or refer to a single element in a page that contains hundreds. Being simply told that “something is wrong” provides little added value when someone is required to hunt for the problem in such a complex page.

To this end, Cornipickle was fitted with a mechanism for attempting to circumscribe parts of a page that account for the discovered fault, in the form of what is called a *witness*. A witness is tree of DOM elements; let W be the set of all witnesses. The set of *verdicts* is defined as $V : \{\top, \perp, ?\} \times W \times W$; a verdict is composed of a truth value and two witnesses: one corresponding to truth value \top , the other to truth value \perp .

The verdict conjunction is a function $\otimes : V \times N \times V \rightarrow V$ defined as follows:

$$\otimes(\langle b, w_{\top}, w_{\perp} \rangle, \nu, \langle b', w'_{\top}, w'_{\perp} \rangle) = \begin{cases} \langle \perp, w_{\top}, w_{\perp} \cup \{(\nu, w'_{\perp})\} \rangle & \text{if } b' = \perp \\ \langle ?, w_{\top} \cup \{(\nu, w'_{\top})\}, w_{\perp} \rangle & \text{if } b \neq \perp \text{ and } b' = ? \\ \langle b, w_{\top} \cup \{(\nu, w'_{\top})\}, w_{\perp} \rangle & \text{if } b \neq \perp \text{ and } b' = \top \\ \langle b, w_{\top}, w_{\perp} \rangle & \text{otherwise} \end{cases}$$

The notation (ν, w) designates the creation of a new witness whose root is the DOM node ν , with witness w as its child. The notation $w \cup w'$ designates the addition of w' to the children of witness w . We will abuse notation and accept that the second argument of \otimes be some “empty” element of N we will designate as ν_{\emptyset} .

Verdict conjunction updates the contents of an existing verdict v , given another verdict v' and some DOM element ν . If v' is false, it carries a witness of that falsehood, namely w'_{\perp} ; this witness is attached as

$$\begin{aligned}
\omega(t, v \text{'s } a \text{ equals } v \text{'s } a') &= \begin{cases} \langle \top, \{v, v'\}, \emptyset \rangle & \text{if } v(a) = v'(a') \\ \langle \top, \emptyset, \{v, v'\} \rangle & \text{otherwise} \end{cases} \\
\omega(t, v \text{'s } a \text{ equals } v) &= \begin{cases} \langle \top, \{v\}, \emptyset \rangle & \text{if } v(a) = v \\ \langle \perp, \emptyset, \{v\} \rangle & \text{otherwise} \end{cases} \\
\omega(t, \text{Not } \varphi) &= \ominus(\omega(t, \varphi), v_\emptyset) \\
\omega(t, \varphi \text{ And } \psi) &= \otimes(\otimes(\langle \top, \emptyset, \emptyset \rangle, v_\emptyset, \omega(t, \varphi)), v_\emptyset, \omega(t, \psi)) \\
\omega(t, \varphi \text{ Or } \psi) &= \oplus(\oplus(\langle \perp, \emptyset, \emptyset \rangle, v_\emptyset, \omega(t, \varphi)), v_\emptyset, \omega(t, \psi)) \\
\omega(t, \text{If } \varphi \text{ Then } \psi) &= \oplus(\oplus(\langle \perp, \emptyset, \emptyset \rangle, v_\emptyset, \ominus(\omega(t, \varphi), v_\emptyset)), v_\emptyset, \omega(t, \psi)) \\
\omega(t, \text{There exists } \xi \text{ in } \\ \quad \$(c) \text{ such that } \varphi) &= \bigoplus_{v \in \mathcal{X}(t_0, c)}^{\langle \perp, \emptyset, \emptyset \rangle} \omega(t, \varphi[\xi/v]) \\
\omega(t, \text{For each } \xi \text{ in } \$(c) \varphi) &= \bigotimes_{v \in \mathcal{X}(t_0, c)}^{\langle \top, \emptyset, \emptyset \rangle} \omega(t, \varphi[\xi/v])
\end{aligned}$$

Table 1: The recursive definition of the verdict computation function ω

a child of a new tree whose root is v , and that tree is added to v 's witness of falsehood, w_\perp . Moreover, v 's truth value is set to \perp . In other words, v 's explanation for being false is added to v 's explanation for being false. Otherwise, if neither v nor v' is false, then v 's witness associated to \top is added to v 's \top witness, and its truth value is updated accordingly. In all other cases, v is left unchanged. A dual definition can be built for verdict disjunction.

Using these operators, the formal semantics of the language can be lifted to a function $\omega : T^* \times \Phi \rightarrow V$, which, out of an expression $\varphi \in \Phi$ and a DOM tree $t \in T^*$, computes a verdict. The recursive semantics of that function is shown in Table 1. The details of that function are out of the scope of the present paper. However, we can see the result of applying ω on the DOM tree of Figure 2a. The function returns a tree containing pointers to two of the page's elements, highlighted in red in Figure 2b. (Actually, the function returns multiple sets, each of which contains the second list item and one of the remaining items.)

Intuitively, such a result makes sense for a web designer; indeed, these two elements should be aligned, while they are not. However, this information can only be deduced through knowledge of the violated property; the witness simply points to these two elements, without providing information about “what is wrong” about them.

3 A Generic Definition of Repairs

While the recursive counter-example generation present in the current version of Cornipickle provides more information than a simple true/false verdict, in many cases it may still prove too vague to be useful.

In this section, we introduce the notion of *repair*, which can be defined intuitively as a set of modifications required to some object to make it satisfy a property. The notion of repair can be seen as fault localization, expressed in reverse: stating how an object needs to be repaired indirectly points to aspects of its structure that are responsible for the fact that the property is not currently fulfilled. We shall see that, contrarily to the concept of witness, which is heavily coupled with the specification language and domain objects used, repairs are defined at a level of abstraction that does not rely on properties of either.

3.1 Definition

Let Σ be a set of *structures*, and T_Σ a set of endomorphisms on Σ ; that is, each $\tau \in T_\Sigma$ is a function $\tau : \Sigma \rightarrow \Sigma$. Let 2^{T_Σ} designate the set of all subsets of T_Σ . A set of endomorphisms $T = \{\tau_1, \dots, \tau_n\} \in 2^{T_\Sigma}$

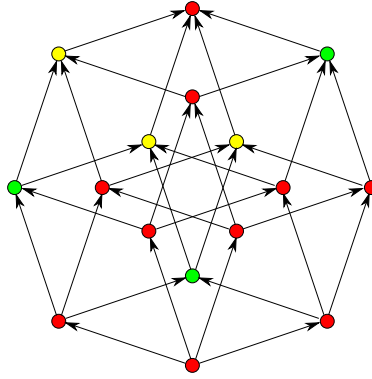


Figure 3: Illustration of the concept of prime repair.

is said to be *well defined* if any two elements τ_i, τ_j are such that $\tau_i \circ \tau_j \equiv \tau_j \circ \tau_i$. Such a well defined set will be called a *transformation*. When the context is clear, we shall abuse notation and consider T as the (uniquely defined) endomorphism $\tau_1 \circ \dots \circ \tau_n$. Set inclusion induces a partial ordering over transformations.

Let Φ be a set of *language expressions* equipped with a satisfaction relation $\models : \Sigma \times \Phi \rightarrow \{\top, \perp\}$. For an expression $\varphi \in \Phi$ and a structure $\sigma \in \Sigma$, we will write $\sigma \models \varphi$ if and only if $\models(\sigma, \varphi) = \top$. In such a case, we shall say that σ “satisfies” φ , or alternately that σ is a *model* of φ .

Let $\sigma \in \Sigma$ be a structure such that $\sigma \not\models \varphi$ for some expression $\varphi \in \Phi$. A *repair* is defined as a transformation $T \in 2^{T_\Sigma}$ such that $T(\sigma) \models \varphi$. A repair is said to be *prime* if no subset $T' \subseteq T$ is such that T' is also a repair. Intuitively, a prime repair is a set of “changes” to a structure σ that make it satisfy φ , such that no “smaller” change also restores satisfiability. Since \subseteq is a partial order, there may be multiple, mutually incomparable prime repairs.

Figure 3 illustrates this concept. The picture represents all transformations that can be applied to a structure, in the simple case where only four morphisms exist. The empty transformation is at the bottom, and each arrow in the graph represents the addition of one more morphism to an existing transformation. Red nodes indicate transformations that are not repairs, while yellow and green nodes indicate repairs. Of these, prime repairs are coloured in green; one can see that all antecedents of green nodes are red. The converse, however, is not true: not all descendants of a repair are repairs themselves.

3.2 Examples

This simple definition can then be applied to a variety of specification languages, as we shall illustrate through the examples that follow.

3.2.1 Propositional Logic

As a first example, let Φ be the set of propositional logic formulæ with variables $X = \{x_1, \dots, x_n\}$ for some $n \geq 1$. Let Σ be the set of functions $X \rightarrow \{\top, \perp\}$, which we shall call *valuations*. The satisfaction relation \models is defined as $\sigma \models \varphi = \top$ if and only if φ evaluates to true when its variables are replaced by the corresponding truth value specified by σ , and \perp otherwise.

Let $b \in \{\top, \perp\}$ and $i \in [1, n]$. We will note $\tau_{x_i \mapsto b}$ the endomorphism defined as:

$$(\tau_{x_i \mapsto b}(\sigma))(x) = \begin{cases} b & \text{if } x = x_i \\ \sigma(x) & \text{otherwise} \end{cases}$$

This morphism sets x_i to b and leaves the rest of the original valuation unchanged. The set of endomorphisms T_Σ is then defined as:

$$T_\Sigma = \bigcup_{i \in [1, n]} \bigcup_{b \in \{\top, \perp\}} \tau_{x_i \mapsto b}$$

Two transformations $\tau_{x \mapsto b}$, $\tau_{y \mapsto b'}$ commute if $x \neq y$. Hence a set of transformations $T \in 2^{T_\Sigma}$ is well defined if and only if every endomorphism it contains changes the value of a different variable.

Let $X = \{a, b, c\}$, σ be the valuation $\{a \mapsto \top, b \mapsto \perp, c \mapsto \perp\}$ and φ the propositional formula $a \wedge b$. One can easily observe that $\sigma \not\models \varphi$. A repair is the transformation $T = \{\tau_{b \mapsto \top}\}$; that is, $T(\sigma) \models \varphi$. This corresponds to the intuition that the explanation for the falsehood of φ is that b is false while it should be true. Note that although $T' = \{\tau_{b \mapsto \top}, \tau_{c \mapsto \top}\}$ would also make φ true, it is not a *prime* repair, since $T \subseteq T'$. This corresponds to the intuition that the truth value of c is not relevant to the falsehood of φ .

Let σ be the valuation $\{a \mapsto \top, b \mapsto \perp, c \mapsto \perp\}$ and φ the propositional formula $a \rightarrow b$. This time, two prime repairs exist: $T = \{\tau_{b \mapsto \top}\}$ and $T' = \{\tau_{a \mapsto \perp}\}$. It is possible to check that both fix the truth value of the original valuation. Informally, the first transformation accounts the falsehood of φ on the fact that a is true, while the other one rather explains it by the fact that b is false—which indeed corresponds to the intuition. Since both repairs are incomparable, none of these explanations is “preferred”. We shall revisit this concept later.

3.2.2 First-Order Logic

The concept of repair can easily be lifted to the set Φ of first-order logic formulæ on finite domains. Let A be a set of elements; an n -ary predicate is defined as a function $p : A^n \rightarrow \{\top, \perp\}$; let P^i be the set of predicates of arity i . A signature is a set of predicates $P = \{p_1, \dots, p_m\}$, respectively of arity a_1, \dots, a_m . For a given signature, the set of domain elements is defined as:

$$\Sigma = P^{a_1} \times \dots \times P^{a_m}$$

The satisfaction relation \models is defined as $\models (d, \varphi) = \top$ if φ evaluates to true when evaluating predicates as defined in σ , and \perp otherwise.

In this context, an endomorphism will represent the change in the truth value for one input of one predicate. Let p_k be a predicate of arity i , $(a_1, \dots, a_k) \in A^n$ be a k -tuple of elements of A , and $b \in \{\top, \perp\}$. The transformation $\tau_{p_k(a_1, \dots, a_k) \mapsto b}$ is defined as the predicate p'_k such that:

$$p'_k(x_1, \dots, x_k) = \begin{cases} b & \text{if } x_1 = a_1, \dots, x_k = a_k \\ p_k(x_1, \dots, x_k) & \text{otherwise} \end{cases}$$

The set of transformations for p_k , noted T_{p_k} , is defined as:

$$T_{p_k} \triangleq \bigcup_{(a_1, \dots, a_k) \in A^n} \left(\bigcup_{b \in \{\top, \perp\}} \{\tau_{p_k(a_1, \dots, a_k), b}\} \right)$$

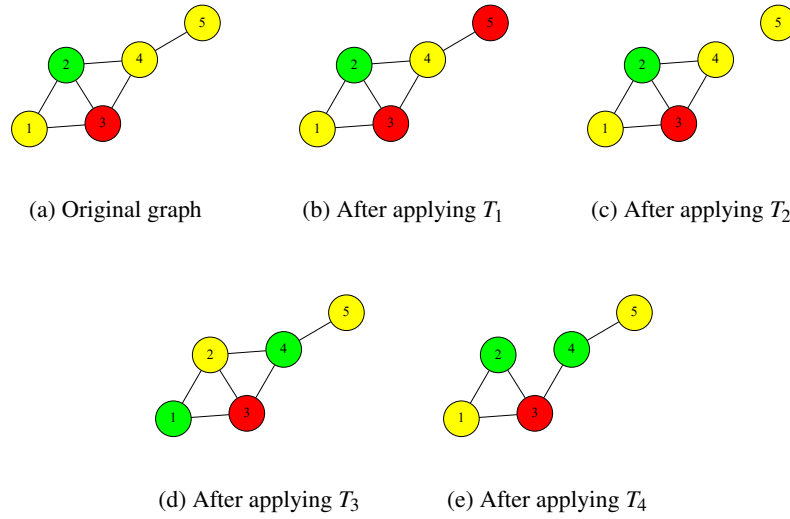


Figure 4: A few possible repairs for a faulty graph colouring

The global set of transformations is then:

$$T_{\Sigma} \triangleq \bigcup_{p \in P} T_p$$

Similarly to first-order logic, one can check that two endomorphisms commute if they operate on a different predicate, or change the value of a different input on the same predicate.

Let $A = \{0, 1, 2\}$, φ be the first-order formula $\forall x : \exists y : x \neq y \wedge p(x, y)$, and the binary predicate p defined as $\{(0, 0), (0, 1), (1, 1)\}$. There are two prime repairs for restoring the truth of φ : $T_1 = \{\tau_{p(2,0) \mapsto \top}\}$, $T_2 = \{\tau_{p(2,1) \mapsto \top}\}$. This corresponds to the intuition that value 2 is missing at least one “partner” in p , and that either 0 or 1 could fit that purpose.

Let $A = [1, 5]$ be a set of graph vertices, p a binary predicate encoding the adjacency relationship of graph edges, and q_1, q_2, q_3 a set of unary predicates such that $q_i(x)$ holds if and only if vertex x has colour i . Suppose predicates p and q are defined according to the graphical representation shown in Figure 4a.

A solution to the graph colouring problem can be represented by three first-order expressions:

$$\varphi_1 \triangleq \forall x : (q_1(x) \wedge \neg q_2(x) \wedge \neg q_3(x)) \vee (\neg q_1(x) \wedge q_2(x) \wedge \neg q_3(x)) \vee (\neg q_1(x) \wedge \neg q_2(x) \wedge q_3(x))$$

$$\varphi_2 \triangleq \forall x : \forall y : p(x, y) \rightarrow p(y, x)$$

$$\varphi_3 \triangleq \forall x : \forall y : p(x, y) \rightarrow ((q_1(x) \rightarrow \neg q_1(y)) \wedge (q_2(x) \rightarrow \neg q_2(y)) \wedge (q_3(x) \rightarrow \neg q_3(y)))$$

The first stipulates that every vertex has exactly one colour; the second indicates that the adjacency relation is symmetric, and the final expression stipulates that no adjacent vertices can have the same colour. One can see that the original graph does not satisfy $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. There exist multiple prime repairs, a few

of which are shown here:

$$\begin{aligned}
T_1 &= \{ \tau_{q_1(5) \mapsto \perp}, \tau_{q_2(5) \mapsto \top} \} \\
T_2 &= \{ \tau_{p(4,5) \mapsto \perp}, \tau_{p(5,4) \mapsto \perp} \} \\
T_3 &= \{ \tau_{q_1(1) \mapsto \perp}, \tau_{q_3(1) \mapsto \top}, \tau_{q_1(4) \mapsto \perp}, \tau_{q_3(4) \mapsto \top} \} \\
T_4 &= \{ \tau_{p(2,4) \mapsto \perp}, \tau_{p(4,2) \mapsto \perp}, \tau_{q_1(4) \mapsto \perp}, \tau_{q_3(4) \mapsto \top} \}
\end{aligned}$$

Repair T_1 fixes the graph by changing the colour of vertex 5 to red. Note that this necessitates not only setting $q_2(5)$ to \top , but also $q_1(5)$ to \perp ; otherwise the resulting structure would violate φ_1 . Another repair (not shown), changes vertex 5 to green. Repair T_3 rather alters the adjacency relation and cuts vertex 5 from the rest of the graph, so that the colour conflict is resolved.

These correspond to the “intuitive” ways of fixing the graph colouring. However, there exist multiple other prime repairs that fulfill the definition. For example, transformation T_4 exchanges the colours of vertices 1, 2 and 4. Note that this is indeed a prime repair, in that no subset of these endomorphisms restore satisfiability of the original formula. In the same way, T_5 cuts the edge between vertices 2 and 4, and turns 4 to green. In total, there are 17 distinct prime repairs in this particular example.

Again, it should be noted that without additional context, none of these repairs is a more likely explanation of the falsehood of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ on the original graph.

3.2.3 Extended First-Order Logic

The previous example shows the need to extend the semantics of first-order logic to arbitrary functions instead of strictly Boolean predicates. This can easily be done as follows. Let A_1, \dots, A_n and B be finite sets. We will denote by $F^{A_1 \dots A_n \rightarrow B}$ the set of all functions $(\prod_i A_i) \rightarrow B$. A signature is a tuple of the form:

$$\langle (A_{1,1}, \dots, A_{1,n_1}) \rightarrow B_1, \dots, (A_{m,1}, \dots, A_{m,n_m}) \rightarrow B_m \rangle$$

such that f_i is a function of arity n_i with domain $A_{1,1}, \dots, A_{1,n_i}$ and image B_i . Predicate logic is the special case where $B_1 = \dots = B_{n_m} = \{\top, \perp\}$, in which case the image can be omitted, and where the $A_{i,j}$ are all the same, so that only the arity needs to be known. If f is a function $A \rightarrow B$ and x designates an element of A , we shall write $x.f$ to denote $f(x)$, thus allowing some form of “object” notation for functions.

In this setting, first-order quantifiers need to precise over which of the $A_{i,j}$ they apply, so that expressions become of the form $\forall x \in A_{i,j} : \varphi$ and $\exists x \in A_{i,j} : \varphi$. Ground terms can now compare values of two function terms, using any appropriate binary operator. Endomorphisms are still defined in the same way as for classical first-order logic, with the provision that they refer to appropriate values in the domain and image of the function subject to the change.

It should be noted that this extended formalism does not add any expressiveness to first-order logic if all sets are kept finite. It shall, however, simplify the expression of many properties.

Equipped with this modified formalism, we are ready to consider repairs in web layout properties. Let E be a set of page elements, P be a set of pixel values, and C be a set of CSS colours. Over these three sets, let us define the functions $E \rightarrow P$ called *left*, *right*, *top*, and *bottom*, corresponding to the x and y coordinates of the top-left and bottom-right corner of an element, respectively. Additionally, we define a set S of CSS *selectors*; the evaluation of a CSS selector over a document can be formalized as a function $\$: S \rightarrow 2^E$ which, for a given filter expression, returns the subset of E matching the selector.

Endomorphisms can be defined for each of these functions, and shall be written using the notation introduced earlier. For example, $\tau_{\text{width}(e) \mapsto k}$ corresponds to the endomorphism setting the value of function *width* for element $e \in E$ to k , and leaving everything else as is.

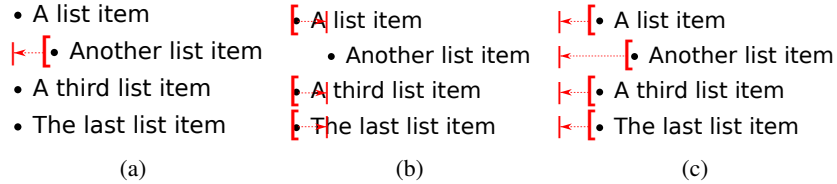


Figure 5: Three repairs for the web example

One can then express the property that all items within a list with ID “menu” should be left-aligned as the following first-order expression:

$$\forall x \in \$(\#menu li) : \forall y \in \$(\#menu li) : x.left = y.left$$

Note that this expression corresponds directly to the first-order translation of the Cornipickle expression shown in Section 2.

Finding the prime repairs for that expression and the page fragment shown in Figure 2a produces a number of solutions, three of which are shown in Figure 5. The first two are fairly intuitive. Figure 5a fixes the page by moving the lone misaligned list item in line with the others, while Figure 5b does the opposite, and aligns the three leftmost list items to the second one. Figure 5c gives an example of one of the many remaining solutions; in this case, all list items are moved to a new, common x position, which turns out to be a coordinate that no element had in the original page.

This last example provides a graphical illustration of the difference between the original concept of witness, and that of repair. While a witness in this case highlights a randomly chosen pair of misaligned elements (as was shown in Figure 2b), a repair picks specific elements and, in addition, describes what should be done with them to fix the violation of the property. This is arguably more telling to a user, and constitutes in our view one of the key advantages of this technique.

4 Computing Repairs

The basic concept of repair introduced in the previous section lends itself to a few discussion points. In particular, the number of possible prime repairs is potentially high, and the task of generating these repairs can therefore prove computationally intensive.

4.1 Basic Algorithm and Complexity

Algorithm 1 shows an algorithm for iterating over all possible repairs of a structure. The algorithm simply enumerates all possible transformations $T \in 2^{T_2}$. It first checks whether T is well defined (i.e. that any pair of endomorphisms commutes), and whether any previously generated repair (stored in set T_S) is a subset of the current one. It finally checks whether applying that transformation fixes the original structure. It skips to the next candidate transformation, should any of these three situations occur. Otherwise, the algorithm adds this transformation to its set, and returns it as its next element.

Theorem 1 *Algorithm 1 is sound and complete.*

Let T be a transformation output by the algorithm. By construction, T is a repair, since is well defined and fixes the truth value of σ on φ . Moreover, at the moment T is output, it is such that none of the

Algorithm 1 Generic algorithm for iterating over prime repairs

```

procedure COMPUTEREPAIRS( $\varphi, \sigma, 2^{T_\Sigma}$ )
   $T_S = \emptyset$ 
  for all  $T \in 2^{T_\Sigma}$  do ▷ Enumerated by increasing cardinality
    if  $\neg \text{WELLDEFINED}(T)$  then
      skip
    end if
    if  $\text{SUBSUMED}(T, T_S)$  then
      skip
    end if
    if  $T(\sigma') \neq \varphi$  then
      skip
    end if
     $T_S \leftarrow T_S \cup \{T\}$ 
    yield  $T$ 
  end for
end procedure

```

elements of T_S are a subset of T . Since T_S contains all repairs of cardinality smaller than T , and that, by construction, all transformations of similar cardinality cannot be subsets of each other, it follows that T is not subsumed by any existing repair, and is hence prime. This proves the soundness of the algorithm.

The fact that all such prime repairs are eventually enumerated is guaranteed by the fact that all subsets of T_Σ are generated at some point, thereby proving completeness.

This algorithm has been implemented in Java and is publicly available¹. Owing to its simplicity and its genericity, the implementation of expressions, structures and repair iteration amounts to a mere 325 lines of code. The enumeration of repairs is exposed to the user in the form of a classical Java Iterator class, which can be used through the traditional `hasNext()` and `next()` methods to pass through the entire set of prime repairs, in increasing order of cardinality. Domain-specific classes defining propositional and first-order logic constructs are made of roughly 500 additional lines of code.

It is easy to see that the running time of this algorithm is exponential in the size of T_Σ , which itself may be exponential in some other factor. In first-order logic, if a_1, \dots, a_n is the respective arity of each predicate in the signature, the number of endomorphisms is $\sum_i 2|A|^{a_i}$ for a given domain A .

Despite this, it is possible to show that this algorithm is limited by a theoretical lower bound. A set of endomorphisms T_Σ is said to be *complete* if for every $\sigma, \sigma' \in \Sigma$, there exists a well-defined transformation $T \subseteq T_\Sigma$ such that $T(\sigma) = \sigma'$.

Theorem 2 *Given a set of structures Σ , a set of language expressions Φ and a complete set of transformations T_Σ , the problem of computing prime repairs is at least as hard as the satisfiability problem for Φ .*

Let $\varphi \in \Phi$ be some language expression. If φ is satisfiable, then there exists some structure $\sigma \in \Sigma$ such that $\sigma \models \varphi$. Take an arbitrary structure $\sigma' \in \Sigma$. Since T_Σ is complete, there exists at least one transformation $T \subseteq T_\Sigma$ such that $T(\sigma') = \sigma$. Take the smallest such set; by definition, it is a prime repair and will eventually be enumerated by Algorithm 1. Since the algorithm is sound and complete, on the contrary, no repair will be found if φ is not satisfiable.

¹<https://bitbucket.org/sylvainhalle/fault-finder>

4.2 Reducing Number of Candidate Solutions

These basic complexity results warrant a discussion about the reduction in the number of potential repairs that need to be explored.

4.2.1 Removing Endomorphisms

The number of potential transformations can first be reduced by removing endomorphisms that are known to be impossible, based on the context. For example, suppose that the propositional symbols a and b in Example 3.2.1 correspond to the assertions “the client pays for an item” and “the client is shipped the item”, respectively. One could assume that a valuation where a is true cannot be modified by making it false; this would correspond to the fact that an action done by some actor cannot be undone. In such a context, only endomorphisms setting false variables to true would be considered.

In the case of graphs, as in Example 3.2.2, one could impose restrictions on what changes are allowed to it; for example, one could say that existing edges must remain unchanged, or that only specific vertices may be coloured differently. This, again, has for effect of preferring some transformations over others, and globally reduces the number of available repairs.

4.2.2 Transformations in Bulk

The granularity of available endomorphisms can also be changed. In the case of the graph colouring example, it is obvious that no repair will ever consist of a single endomorphism $\{\tau_{q_i(x) \mapsto \top}\}$. The reason is that expression φ_1 requires that each vertex be of exactly one colour; assigning some q_i to \top for a vertex entails that the remaining q_j for $j \neq i$ be set to \perp . One can hence define a new set of transformations appropriate for the context, representing *colour changes*:

$$T_C = \bigcup_{x \in A} \bigcup_{\substack{i \in [1,3] \\ j \neq i \\ k \neq j \neq i}} \{\{\tau_{q_i(x) \mapsto \top}, \tau_{q_j(x) \mapsto \perp}, \tau_{q_k(x) \mapsto \perp}\}\}$$

Similarly, as the adjacency relation is symmetric, setting $p(x, y)$ to \top (resp. \perp) cannot be done without also setting $p(y, x)$ to \top (resp. \perp). Instead of considering individual changes to single inputs of p , one can define a set of *edge changes*:

$$T_E = \bigcup_{x \in A} \bigcup_{y \in A} \bigcup_{b \in \{\top, \perp\}} \{\{\tau_{p(x,y) \mapsto b}, \tau_{p(y,x) \mapsto b}\}\}$$

One could then use $T_C \cup T_E$ as the set of transformations, instead of T_Σ . While this makes no change in theory on the actual solutions, the fact that $T_C \cup T_E$ is smaller in size than T_Σ has a positive effect on the performance of an enumeration algorithm in practice.

The same can be said of the endomorphisms of Example 3.2.3. Rather than consider all individual changes of (x, y) coordinates of all four corners of every element, one could define subsets corresponding to more intuitive modifications; for example, the set of *horizontal displacements* could be defined as:

$$T_H = \bigcup_{e \in E} \bigcup_{p \in P} \{\{\tau_{\text{left}(e) \mapsto p}, \tau_{\text{right}(e) \mapsto (\text{right}(e) - p)}\}\}$$

One can then restrict the search for repairs to those that are made only of (horizontal or vertical) displacements, or (horizontal or vertical) *resizings* of elements, etc.

5 Related Work

The concept of repair calculation is under construction, and its ties to related work still need to be established. As we have seen in the previous section, finding repairs relates to the concept of satisfiability solving (SAT), and more precisely the problem of *incremental SAT* [5]. Traditional SAT solvers are required to find a single model of an expression if such model exists. In incremental SAT, a solver finds a first model of an expression, but can also be repeatedly asked to provide additional models. When a set of transformations is complete, iterating over models amounts to iterating over repairs.

The use of model finding has also been studied in the field of network configuration management. An early version of the concept of witness was suggested by one of the authors [4] in the context of self-configuration. Couch *et al.* proposed the notion of *convergent* actions for repairing the configuration of a network [2] (similar to our notion of well-definedness), while Narain [6] suggested six uses for a procedure $P(\varphi, x)$ that finds values for x that satisfy some configuration constraint φ . Two of them deserve special attention:

- Configuration Error Fixing: if a configuration does not fulfill some specification φ , compute $P(\varphi, x)$ and take the solution x closest to the current configuration.
- Requirement Strengthening: to reconfigure a network with a set of additional constraints ψ , compute $P(\varphi \wedge \psi, x)$ (that is, the constraints of φ and ψ together) and take the result x .

In both cases, Narain suggested the use of a satisfiability solver for generating the target configuration; however, a SAT solver in general does not give control over what model is returned. The solution “closest” to the current configuration is better captured by the concept of prime repair, which corresponds exactly to that concept.

6 Conclusion

The proof-of-concept prototype of Cornipickle has shown promising results in its ability to easily express conditions for layout-based bugs in web applications, and efficiently detecting them in sample pages from more than 35 real-world applications. However, its ability to return a useful explanation for the violation of a property on a given web document is limited. This paper has introduced a definition of the concept of *repair*, whose calculation provides more precise information about the changes required to a structure in order to satisfy a given specification.

The study of repairs and their computation is part of ongoing work, and many problems are still open. For example, an efficient computation of repairs relies on the deletion of as many candidate transformations as possible; therefore, techniques to *easily* identify endomorphisms that can never be part of a solution could be sought after. Similarly, we are planning to study techniques that could generate the set of repairs directly from the specification and the faulty structure, rather than using the crude generate-and-test algorithm presented in this paper. In spite of this, early results on a number of examples show that using repairs as a form of fault localization is promising, and in particular provides results that correspond to intuition in many cases.

References

- [1] Bert Bos, Tantek Çelik, Ian Hickson & Høakon Wium Lie (2011): *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. Technical Report, World Wide Web Consortium. <https://www.w3.org/TR/CSS2/>.

- [2] Alva L. Couch & Yizhan Sun (2003): *On the Algebraic Structure of Convergence*. In Marcus Brunner & Alexander Keller, editors: *DSOM, Lecture Notes in Computer Science* 2867, Springer, pp. 28–40, doi:10.1007/978-3-540-39671-0_4.
- [3] Sylvain Hallé, Nicolas Bergeron, Francis Guerin & Gabriel Le Breton (2015): *Testing Web Applications Through Layout Constraints*. In: *ICST, IEEE*, pp. 1–8, doi:10.1109/ICST.2015.7102635.
- [4] Sylvain Hallé, Éric Wenaas, Roger Villemaire & Omar Cherkaoui (2006): *Self-configuration of Network Devices with Configuration Logic*. In Dominique Gäiti, Guy Pujolle, Ehab S. Al-Shaer, Kenneth L. Calvert, Simon A. Dobson, Guy Leduc & Olli Martikainen, editors: *AN, Lecture Notes in Computer Science* 4195, Springer, pp. 36–49, doi:10.1007/11880905_4.
- [5] Alexander Nadel & Vadim Ryvchin (2012): *Efficient SAT Solving under Assumptions*. In Alessandro Cimatti & Roberto Sebastiani, editors: *SAT, Lecture Notes in Computer Science* 7317, Springer, pp. 242–255, doi:10.1007/978-3-642-31612-8_19.
- [6] Sanjai Narain, Gary Levin, Sharad Malik & Vikram Kaul (2008): *Declarative Infrastructure Configuration Synthesis and Debugging*. *J. Network Syst. Manage.* 16(3), pp. 235–258, doi:10.1007/s10922-008-9108-y.