Semantic Constraint Inference for Web Form Test Generation

Parsa Alian
University of British Columbia
Vancouver, Canada
palian@ece.ubc.ca

Noor Nashid
University of British Columbia
Vancouver, Canada
nashid@ece.ubc.ca

Mobina Shahbandeh
University of British Columbia
Vancouver, Canada
mobinashb@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

Abstract

Automated test generation for web forms has been a longstanding challenge, exacerbated by the intrinsic human-centric design of forms and their complex, device-agnostic structures. We introduce an innovative approach, called FormNexus, for automated web form test generation, which emphasizes deriving semantic insights from individual form elements and relations among them, utilizing textual content, DOM tree structures, and visual proximity. The insights gathered are transformed into a new conceptual graph, the Form Entity Relation Graph (FERG), which offers machine-friendly semantic information extraction. Leveraging LLMs, FormNexus adopts a feedback-driven mechanism for generating and refining input constraints based on real-time form submission responses. The culmination of this approach is a robust set of test cases, each produced by methodically invalidating constraints, ensuring comprehensive testing scenarios for web forms. This work bridges the existing gap in automated web form testing by intertwining the capabilities of LLMs with advanced semantic inference methods. Our evaluation demonstrates that FormNexus combined with GPT-4 achieves 89% coverage in form submission states. This outcome significantly outstrips the performance of the best baseline model by a margin of 25%.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**.

Keywords

Web Forms, Test Input Generation, Large Language Models

ACM Reference Format:
Parsa Alian, Noor Nashid, Mobina Shahbandeh, and Ali Mesbah. 2024. Semantic Constraint Inference for Web Form Test Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680332

1 Introduction

Forms are crucial for gathering dynamic user data and facilitating effective communication between users and software applications. Given their importance, it is essential to thoroughly test the functionality of these web forms to ensure their accuracy and reliability.

While there have been advancements in web testing methodologies [11, 14, 16, 52], the realm of form test generation remains sparsely explored [41]. Generating test cases for forms introduces a distinct set of challenges. Since forms are tailored for human interaction, generating suitable values necessitates a grasp of the *context* of each input field: understanding the semantics of fields, as well as how they relate to one another. In the context of the black box test generation for web forms in particular [21], understanding the Document Object Model (DOM) introduces another layer of intricacy, which can at times overshadow the form's inherent semantics. The flexibility in coding that allows for visually identical displays adds a layer of complexity to the web form's architecture. Furthermore, a push for web applications to be device-agnostic impacts the design of HTML structures, making it even more elusive. These complexities can pose significant hurdles for the automated testing of web forms.

Given the imperative to comprehend form semantics for automated test generation, leveraging Natural Language Processing (NLP) techniques for form input generation emerges as a promising avenue. The spectrum of potential methodologies has broadened notably with the recent introduction of Large Language Models (LLMs) such as GPT-4 [6] and Llama-2 [48]. The adeptness of LLMs in emulating human-like language processing and generation paves the way for a new approach to this endeavor. Recently, studies have leveraged LLMs for a wide array of tasks, such as unit test generation [28, 31, 37, 42, 44, 51] and mobile app form-filling [35]. The advent of these techniques presents an exciting frontier in addressing the complexities of automated web form test generation.

In this paper, we introduce FormNexus, a novel LLM-agnostic technique designed explicitly for the automated generation of web form tests. At its core, this method grapples with the intricacies inherent in understanding the context of input fields. We do this by transforming the form's DOM layout into a more organized structure, called **F**orm **E**ntity **R**elation **G**raph (**FERG**), where the semantics and relationships of form elements become more clear and better suited for machine interpretation. To make this transformation possible, we analyze each node's characteristics, including its textual content and position in the DOM hierarchy. Based on these factors, we determine similarities between various HTML nodes and identify potential relationships between different nodes, which in turn provides insights into the semantics of individual inputs and the connections that might exist between them.

After establishing the semantic linkages within the form, we adopt a feedback-driven methodology that leverages these connections in conjunction with LLMs to formulate test cases for the form. The procedure begins by deducing preliminary constraints rooted in semantic associations, followed by generating input values following these constraints. FormNexus then verifies and modifies the inferred constraints, generates new input values, and submits the form under these modified constraints. Our goal is to corroborate the constraints using the feedback obtained after submission. Once the constraints are validated, FormNexus converts them into a comprehensive set of test cases. These cases serve as a valuable vehicle for checking and deciphering the form's runtime functionality.

To evaluate FormNexus, we employ a diverse selection of real-world and open-source applications. We utilize Llama 2 and GPT-4 as the LLM underpinning FormNexus. Our results show that FormNexus instantiated with GPT-4 delivers the best results, achieving a state coverage of **89%** marking a significant **25%** improvement over the next best performer baseline, GPT-4 alone. Additionally, successful form submission test cases are equivalent to the form-filling task, where FormNexus with GPT-4 demonstrates **83%** success rate in successfully submitting and passing the forms, outperforming all other baselines by at least **27%**. Our evaluation also scrutinizes the individual modules of FormNexus, revealing the contributions of the different components, the inference of semantic relations via FERG, and our feedback loop approach toward the overall effectiveness of our technique.

This work makes the following contributions:

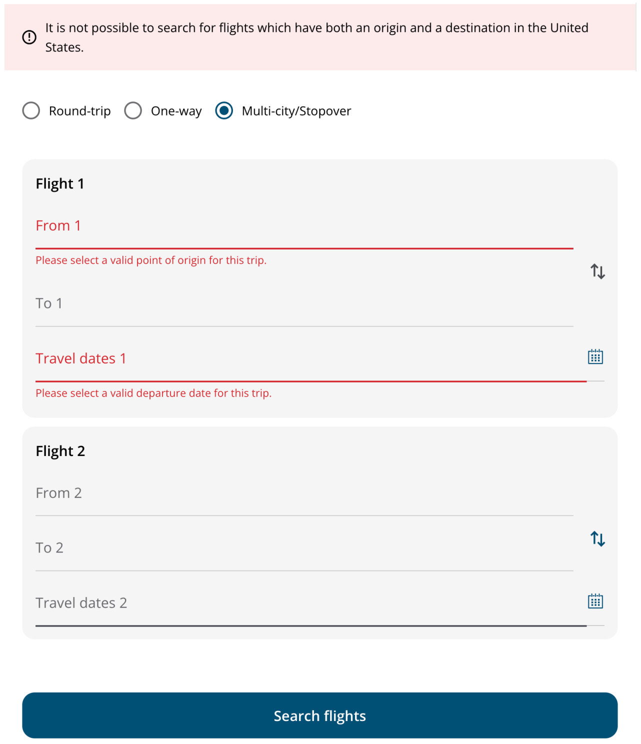
- A novel technique (FERG) that embeds contextual information of each form entity and quantifies the relevance between them.
- A feedback-driven, constraint-based method utilizing LLMs to generate test cases for comprehensive form validation, covering both success and failure scenarios in form submissions.
- A new dataset containing web forms, their input values, and corresponding submission states, enabling rigorous evaluation of form testing techniques.

2 Motivation

We use Air Canada's [1] multi-city flight reservation web form as a motivating example, illustrated in Figure 1. Before deploying such forms, developers need to conduct thorough testing to ensure the form's appropriate response to both valid and invalid input combinations. This is essential to assure the functionality and reliability of the form under various conditions. To methodically assess the effectiveness of their test cases, developers evaluate the coverage across diverse scenarios within the form.

Covering Form Submission States. To quantify the extent of coverage and measure it, we introduce the concept of *Form Submission States*:

Definition 1 (Form Submission State (FSS)). A **Form Submission State** (FSS) is a unique tuple $S = (I, F)$, where:


It is not possible to search for flights which have both an origin and a destination in the United States.

( ) Round-trip ( ) One-way (●) Multi-city/Stopover

Flight 1
From 1
Please select a valid point of origin for this trip.
To 1
Travel dates 1
Please select a valid departure date for this trip.

Flight 2
From 2
To 2
Travel dates 2

Search flights

Figure 1: Air Canada's multi-city flight reservation form

- I represents the minimal subset of input fields that, when submitted, directly influences the generation of specific feedback F by the form processing logic. The subset is minimal in that it contains only those fields necessary to trigger the feedback F, excluding any inputs that do not alter the outcome.
- Given I, F is the unique and single feedback provided by the form, which is logically and causally related to the specific input fields in I.
- Value mutations to an input subset will not create a new FSS if the feedback remains unchanged.
- The set of all FSSs is disjointed, meaning combining multiple feedback messages would not constitute a new FSS.

According to this definition, Air Canada's form [1] has a total of 20 distinct FSSs. Table 1 presents six of the FSSs identified. Within the table, the input subset I is highlighted, and the feedback cell's background color denotes the form's submission outcome (green signifying success, red signifying failure). FSS 1 in Table 1 exemplifies a scenario where all input fields contribute to a successful form submission feedback, resulting in a redirection to the flight list. Subsequent FSSs showcase instances where specific input fields influence the feedback, while the remaining input fields do not exert an influence.

It is imperative to recognize that each *unique* feedback response for an input subset corresponds to a distinct FSS. For instance, encountering invalid dates (FSS 3) in the form triggers a uniform feedback message, `Select a valid departure date`, regardless of the nature of the date error. In contrast, some forms are designed to generate diverse feedback for value mutations in a

Table 1: Sample FSSs for Air Canada's flight reservation form

label	From 1	To 1	Travel dates 1	From 2	To 2	Travel dates 2	Feedback
FSS 1	Toronto	Vancouver	08/04	Vancouver	Montreal	12/04	Redirect to flight list.
FSS 2	- abcdefg	Vancouver	08/04	Vancouver	Montreal	12/04	Please select a valid point of origin for this trip.
FSS 3	Toronto	Vancouver	- not-a-date 08	Vancouver	Montreal	12/04	Please select a valid departure date for this trip.
FSS 4	Toronto	Toronto	08/04	Vancouver	Montreal	12/04	Departure and arrival cities are the same.
FSS 5	Calgary Toronto	Vancouver	08/04	Calgary Toronto	Montreal	12/04	You've entered the same point of origin and/or The same destination twice.
FSS 6	Toronto	Vancouver	15/06 12/04	Vancouver	Montreal	12/05 08/04	Return date cannot be before departure date.
...	...	...	...	...	...	...	...

single input field, such as distinguishing between an empty date field and an invalid date format. This differentiation in feedback implies distinct underlying logic and, consequently, different execution scenarios. Hence, the uniqueness of the feedback given an input subset serves as an indicator of separate handling logic for each scenario, while identical feedback suggests possibly a lack of such distinction.

Form Testing Challenges. While the interface of the form appears straightforward in this example, it encompasses a variety of scenarios, each necessitating thorough testing to achieve adequate coverage. Generating values that guarantee successful form submission (FSS 1 in Table 1) presents a significant challenge in itself. Further complicating the testing process are the form's validation requirements. For instance, specific input fields like location or date, such as `From 1` and `Travel dates 1` in Figure 1, must receive inputs that conform to particular formats (FSS 2 and 3 in Table 1). This necessity extends to other fields which may only accept data matching specific patterns, such as email addresses or telephone numbers. Beyond these standard validations, the form may also incorporate more complex scenarios that require additional, detailed testing, as can be observed in Figure 1 and Table 1:

- Geographic constraints play a crucial role in travel planning. In any general travelling scenario, the points of departure and arrival (`From` and `To`) must differ, as it is logically inconsistent to embark on a journey that begins and ends at the same location. For example, while a trip from `Toronto` to `Vancouver` is feasible, a journey from `Toronto` to `Toronto` is not (FSS 5 in Table 1). Specifically in multi-city travel scenarios, such as the case presented in Figure 1, additional constraints govern location choices. Consecutive trips cannot share the same origin (or destination), as the completion of a flight fundamentally alters the user's starting point for subsequent trips (FSS 4). Therefore, a user cannot book two consecutive trips departing from `Toronto` (FSS 5).
- The order of temporal events is equally vital. Travel dates must be arranged in chronological order, with the `Travel dates 2` date necessarily occurring after the `Travel dates 1` date (FSS 6). Moreover, the dates should not be set in the past or unreasonably far in the future.

Automating the process of test case generation for forms presents numerous challenges. An effective automation system necessitates a multifaceted understanding of the form under evaluation. First, it requires the ability to understand the context and intended purpose of the form. Second, the system must possess knowledge of the nature of each input field, enabling the identification of constraints such as data type and field length. Finally, automation necessitates the ability to comprehend the interrelationships between input fields and how these relationships influence the generation of diverse scenarios requiring test case coverage.

Current LLM-based Approaches. The rapid development of LLMs presents a compelling opportunity to address some of the outlined complexities. To explore these opportunities in the context of web form testing, we employed two baselines, GPT-4 and QTypist [35] an LLM-based mobile app form input generator. Our focus was to guide these baselines in generating input combinations that maximize FSS coverage for Air Canada's form depicted in Figure 1.

Our evaluation exposed significant limitations in these baseline approaches for FSS coverage (Section 4 presents our empirical evaluation). GPT-4 failed to process the form due to its sheer size, exhibiting a token limit error. QTypist managed to cover only three out of 20 FSSs with coverage of 15%; it failed to recognize the distinct input sets for Flight 1 and Flight 2, thus restricting its test case generation to single-field validations within Flight 1 (e.g., FSS 6 in Table 1). We also utilized a GPT-4 variant with a larger context window. While this version covered 6 out of 20 FSSs (30%), it could only handle validations within single input fields. After altering the prompt to include examples of identical departure and arrival cities (FSS 6 of Table 1) as a few-shot prompt, it generated a value set for that isolated case however it failed to generalize to broader geographical or date-related constraint scenarios.

The limitations posed by LLMs in processing web forms become evident when considering the extensive information these forms can contain, potentially surpassing the LLM's context size limit. A strategy to mitigate this challenge involves simplifying the information presented to LLMs, thereby enhancing their capacity for effective data processing. Moreover, while LLMs may possess the intrinsic capability to deduce constraints within web forms and generate corresponding test cases, the complexities of such an endeavor may lead to oversight of numerous scenarios. Implementing a structured approach, wherein LLMs are guided through a sequential analysis of each input field via tailored prompts, can significantly augment their proficiency in covering FSSs.

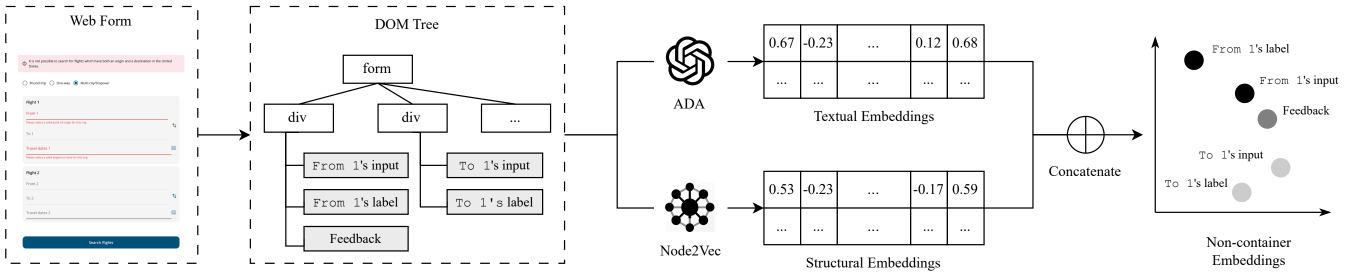


Figure 2: FERG embedding creation stage

3 Approach

We present our approach, FormNexus, for web form test generation. The main focus of our work is to understand the *context* of each form field instead of relying on the whole form's HTML code. The *Input Field Context* refers to any pertinent information that assists in clarifying the requirements and limitations of a given form input field. This context may manifest as textual annotations associated with the field, such as labels or hint texts, or it might be derived from other fields that influence and define the constraints of the focal field.

We adopt a step-by-step process that leverages Input Field Context to guide test case generation, aiming to increase FSS coverage. By harnessing the capabilities of LLMs and employing a feedback-driven approach, we decipher the constraints tied to input fields, which aids in value generation. These identified constraints then form the foundation upon which we craft test cases for the web form.

3.1 Input Context Construction

To tackle the complexities associated with comprehending the context of input fields, we introduce a novel approach that converts the form's DOM tree into a graph structure, termed the **F**orm **E**ntity **R**elation **G**raph (**FERG**). The goal of this graph is two-fold: first, it enhances the contextual information of each individual input field, second, it captures information pertaining to how relevant form entities are to each other and provides a quantitative score of the relevance.

Understanding the relationships among form elements can be complex, particularly when explicit connecting attributes (such as the `for` attribute that links a label and an input) are missing. Implicit relationships between input fields or hint/feedback text often remain unstated within the form's structure. While hierarchical and textual features are available, neither in isolation offers a reliable method for determining these relationships. HTML's flexibility, with its frequent use of `<div>` and `<span>` elements for styling and functionality, defies simple hierarchical analysis. Moreover, textual cues can be misleading; for instance, `From` might reference a geographical input, while `Departure` denotes a date input, despite their textual similarity.

We hypothesize that a combination of hierarchical and textual attributes should reveal a clearer sense of connectivity between elements. To achieve this, we employ embedding techniques to combine the individual features of the elements, thus overcoming their separate limitations. Subsequently, these combined embeddings are utilized to elucidate potential relationships among the elements. This is achieved by constructing a graph, FERG, which represents the interconnected structure of the elements.

3.1.1 *Creating Embedding Space.*

In the construction of the FERG, the initial step involves establishing an embedding space for the elements within the form, as delineated in Figure 2. To encapsulate the connections in the form, we start by generating textual and structural embeddings for non-container elements. In the realm of web forms, non-container elements are defined as those either directly encompassing textual content or functioning as input elements, and these are the elements that users directly interact with. For the generation of text embeddings, which involve the transformation of sentences into embedding vectors, we utilize ADA [38] embeddings. To elucidate the structural relationships among elements, we apply the node2vec methodology [22] on the DOM tree. This approach results in a distinct embedding vector for each node within the DOM tree.

We iterate over the non-container nodes from the DOM tree and calculate the textual and structural embeddings for these nodes. We concatenate the embeddings to form an embedding space, in which we expect to find the similarity of different nodes. For example, since the `From 1` *label* is structurally close to the `From 1` *input* field, and also the text in the label (`From 1`) is similar to the `name` attribute of the input `From 1`, we expect these two elements to fall close to each other in the embedding space. We can then use similarity metrics such as cosine similarity to measure the closeness of the nodes.

3.1.2 *Local Textual Context.*

The first type of context that we aim to clarify is the local textual context. This context commonly consists of any piece of text that might provide clarifications for the input field, such as labels, hint texts, or any relevant feedback for the input field. These relationships are key to deciphering the nature of an input field.

We can observe in Figure 1 that textual elements that are related to each input field are positioned close to their related input field. For instance, the `From 1` input field is closely flanked by two related elements: its corresponding label `From 1` and the feedback text, both sharing visual boundaries with the input field. This phenomenon is typically true in forms since proximity also aids human operators in associating the textual information with the input field. So in this phase of our methodology, we start connecting the elements that share visual boundaries.

We iterate over input fields and compute the cosine similarities with their adjacent textual elements. These calculations are integrated into a graph $G = (V, E, W)$, where V represents the graph's nodes, encompassing the non-container nodes in the form.

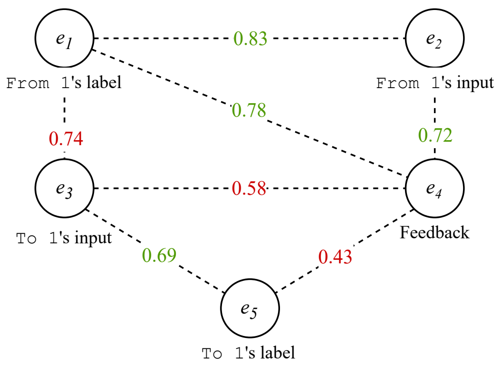


Figure 3: Relating local textual context in the relation graph

E denotes the edges connecting visually adjacent elements, and W is the weight of these edges. The weight is determined by the cosine similarity between the embedding vectors of the elements, which quantifies the extent of contextual dependency between two elements. A sample of the formed graph can be seen in Figure 3.

It is worth noting that being neighbors does not always translate to being related. Therefore, the formation of the initial graph is followed by a pruning process. In this context, we categorize entities within the form into two types: main and auxiliary. Input fields are treated as main entities or *first-class citizens* in form contexts; they are capable of existing without any other elements, such as labels, with their function potentially indicated via attributes such as `placeholder` or `value`. Conversely, auxiliary elements, such as labels or hint texts, inherently depend on the existence of an input field for their relevance. A form composed exclusively of labels or hint texts would lack functional meaning without connection to input fields.

This conceptual framework informs our edge pruning strategy within the graph G. We begin by examining the auxiliary (non-input) nodes in the graph. For each of these nodes, we inspect the edges connected to it. If an auxiliary node is linked to multiple input fields, we retain only the edge with the highest cosine similarity score. For instance, in Figure 3, `From 1`'s label and feedback are connected to both `From 1` and `To 1` input fields. However, since the similarity score of their connection is stronger with `From 1`'s input field, we only keep those edges.

As for text-to-text edges, we apply a statistical method for their retention. We compile the scores of all such edges and filter out those less than the threshold of $\mu + \lambda.\sigma$, where μ represents the mean score, σ the standard deviation, and λ is a predetermined factor set at $\frac{1}{2}$. Again, in Figure 3, the feedback is connected to both `From` and `To` labels, although the connection to label `To` is not statistically significant for us to keep in the graph. After applying the filtering process, edges removed from the graph are indicated in red in Figure 3, while the remaining edges are shown in green.

3.1.3 Relevant Input Context. Having established the textual context for each input field, the subsequent phase of our method identifies relevant inputs that are interrelated. Figure 1 demonstrates that input fields with relationships, such as `From 1` and `To 1` (or `From 2` and `To 2`) , not only share a similar textual context but also are often in close structural proximity. This design is intuitive for human interpretation. The relationship between elements is typically indicated by semantically related labels and their spatial closeness, as the significant distance between elements generally diminishes their perceived relevance.

In light of this observation, we utilize the embeddings previously computed to gauge the degree of connectedness between groups of input fields. Employing the same embedding is advantageous for this task, as it encompasses both textual similarity and structural proximity. We define the relationship score between two input fields (`InputFieldSim`) as the maximum of the input-to-input and label-to-label similarity scores as follows:

$$InputFieldSim = \max\{sim(label_1, label_2), sim(input_1, input_2)\}$$

In instances where input fields lack associated labels, we adapt the methodology by omitting the label terms from the calculation.

It is important to recognize that the relationships discerned between input groups are not inherently obligatory; lower score relations do not necessarily imply a meaningful connection. To effectively identify and exclude less relevant relationships, we employ the same statistical approach previously applied to text-to-text edges, as outlined in Section 3.1.2.

3.2 Constraint Generation and Validation

Our objective now is to use the information in FERG to infer a series of constraints that align with the attributes and relationships of the input fields within the form. In an iterative process, we query the previously constructed FERG to extract relevant elements for each input field. We construct prompts based on the retrieved information and subsequently prompt the LLM for constraint and value generation.

3.2.1 Initial Generation Phase. Following the connection of input fields to their corresponding elements within the form, we can make educated guesses regarding the specific constraints associated with each field. For instance, at this stage, we possess knowledge of the type requirement imposed upon the `From 1` input field, derived from its underlying HTML code, we have inferred the surrounding textual context including its associated label, and we have established a relationship between the `From 1`, `To 1`, and `From 2` input fields. Leveraging this information, humans are capable of inferring a significant number of constraints (e.g., the field should be alphabetical, not equal to the `To 1` or `From 2` fields, etc.). If the specific application context for which the form is intended is known, we can infer virtually all of the constraints for the `From 1` field.

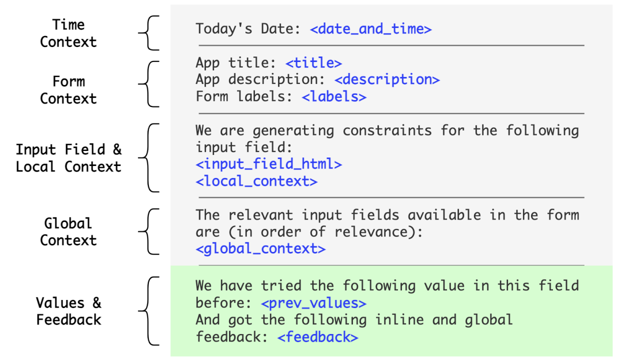
Leveraging the vast datasets on which LLMs are trained, we anticipate their ability to perform similar inferential tasks as humans. Therefore, following the construction of FERG, our subsequent step involves utilizing the relationships and information it encodes to infer an initial set of constraints for the form's input fields. To achieve this, we employ an LLM, tasked with selecting a series of constraints from a pre-defined list of constraint templates, as outlined in Table 2.

While the generated constraints derived from these templates are readily evaluable functions, it is crucial to acknowledge that certain inherent complexities within web applications cannot be effectively

Table 2: Constraint Templates

Signature	Definition
`toBeEqual(value)`	The input field value is exactly equal to the given value.
`toHaveLengthCondition(condition, value)`	The length of the input field value matches the given condition.
`toBeAlphabetical()`	The input field should be alphabetical.
`toContainWhiteSpace()`	The input field should contain whitespace.
...	...

captured using such functions. A pertinent illustration can be observed in the Air Canada form depicted in Figure 1, where the user encounters the following error message: `It is not possible to search for flights which have both an origin and a destination in the United States`. In recognition of this limitation and to accurately represent such intricate logical constraints prevalent in forms, we have introduced a novel constraint type, termed `freeTextConstraint`. This type specifically caters to the capture and articulation of scenarios that transcend the capabilities of conventional constraint types.

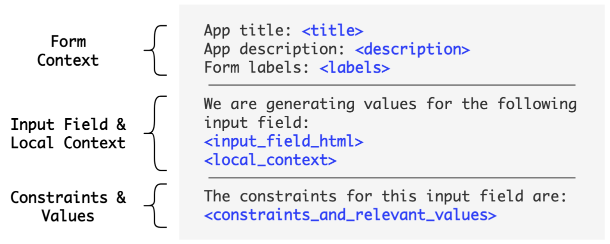


```
Today's Date: <date_and_time>

App title: <title>
App description: <description>
Form labels: <labels>

We are generating constraints for the following
input field:
<input_field_html>
<local_context>

The relevant input fields available in the form
are (in order of relevance):
<global_context>

We have tried the following value in this field
before: <prev_values>
And got the following inline and global
feedback: <feedback>
```

Figure 4: Constraint prompt structure

Our approach to prompting the LLM begins by focusing on each input field. We employ a structured prompt, the details of which are outlined in Figure 4. This prompt includes the following information:

(1) **Time Context:** This detail facilitates constraint generation for date and time fields by enabling the LLM to compare them with the current date.
(2) **Form Context:** To provide general context about the form, we incorporate application's metadata (title and description) and labels from the form. This information informs the LLM about the type and purpose of the application and form, thereby facilitating more accurate and relevant constraint generation.
(3) **Input Field and Local Context:** We provide the HTML of the target input field as context for the LLM. Additionally, we include local textual context extracted from FERG (subsubsection 3.1.2) to hint at the input field's intended purpose within the form structure.
(4) **Global Context:** We consider elements that have a global association with the input field (subsubsection 3.1.3) and incorporate them into the prompt. These inferred relationships assist the LLM in understanding the inter-variable constraints of the input field.



```
App title: <title>
App description: <description>
Form labels: <labels>

We are generating values for the following
input field:
<input_field_html>
<local_context>

The constraints for this input field are:
<constraints_and_relevant_values>
```

Figure 5: Value prompt structure

For each input field, the LLM is provided with these pieces of information, and it is asked to select a set of constraints from constraint templates in Table 2. Given the extensive training of the LLM on a diverse data corpus, we anticipate its ability to grasp the semantics embedded in each input field, and leverage form and FERG's contextual information as guidance. As a result, we expect the LLM to generate a set of constraints closely mirroring the real-world constraints associated with these fields. An example of the LLM's response is demonstrated in Listing 1, showcasing the constraints generated for the `From 1` field in Figure 1.

```
1 expect(field('From 1'))
2 .toBeTruthy()
3 .toBeAlphabetical()
4 .toHaveLengthCondition('>', 2)
5 .not.toBeEqual('To 1')
6 .not.toBeEqual('From 2')
```

Listing 1: `From 1` field's generated constraints

The presented example illustrates the generation of specific constraints (lines 2-4), which are literal, and extrapolated from localized data such as labels. This field is expected to satisfy several conditions, namely, it should (1) not be empty (line 2), (2) adhere to a specified minimum length (line 3), and (3) contain exclusively alphabetical characters (line 4). Additionally, certain constraints (line 5, 6) depend on values drawn from different fields; for instance, (4, 5) the `From 1` field is expected to be distinct from `To 1` and `From 2` fields. These five constraints concisely mirror the anticipated requirements for the values of the `To` field.

The constraints generated by the LLM form a crucial basis for approximating the underlying logic inherent to the input fields, thereby facilitating the generation of appropriate values for the form. This process of value generation is executed by directing the LLM to produce values that comply with the deduced constraints. The specific structure of the prompt used for guiding the LLM in value generation is detailed in Figure 5. This prompt includes:

(1) **Form Context, Input Field, and Local Context:** These sections are identical to the constraint prompt, and are included for the LLM to grasp the overall context of the form. The Time Context is omitted, as we expect the date and time requirements to reflect in the generated constraints.
(2) **Constraints and Values**: These are the generated constraints resulting from prompting the LLM in the previous step. Since we generate values for input fields one by one, we would include the generated values for the relevant input fields in the constraints

if we have already generated a value for that relevant field. For example, when generating a value for the `From 1` field, if the `To 1` field has already been assigned the value *Toronto*, we would include the constraint `input field should not be equal to 'Toronto'` (as per line 5 of Listing 1). However, if the `To 1` field remains unassigned, this constraint would not be included, as it would not yet influence the value generated for `From 1`.

With the contextual information in this prompt, the LLM can effectively generate a variety of values that may meet the requirements of the form.

3.2.2 Feedback Loop and Constraint Updating. Upon completion of the previous step, we obtain a series of constraints and values based on the context of the form. However, at this point, we have not interacted with the form yet, and the adequacy of these values for the form is still undetermined. Therefore, the next step is to populate the form with these generated values and subsequently submit it, thereby triggering a response or *feedback* from the form.

After the submission of the form, several scenarios can unfold. The user might either remain on the initial page or be redirected to a new page. In each of these scenarios, textual indicators may appear, signaling either the success or failure of the form submission. In general, we can define the success or failure of the submission as follows:

Definition 2. A *Failure* in submission is identified by the reception of error feedback from the web application. In contrast, a *Successful* submission is denoted by the absence of such failure feedback and the transition to the intended outcome of the form.

For instance, in the case of Air Canada's flight reservation form, a successful submission would navigate to a page displaying available flight options, while a failed submission would typically generate feedback with error messages, such as the ones shown in Figure 1 .

Given the complexity inherent in identifying the state of the page post-submission, we employ a heuristic-based approach to discern the form's status. This involves calculating the differential (*diff*) of the DOM tree before and after the form's submission. Subsequently, we refine these differences by filtering them through specific keywords commonly associated with feedback messages, such as `not valid`, `required`, `denied`, and similar terms. The elements that emerge after this filtration process are then regarded as the feedback resulting from the form submission. It is important to underscore that this method can be effective in identifying feedback irrespective of whether the submission leads to a page redirection or remains on the same page since it searches for the failure keywords on the page.

After submission, the FERG creation algorithm can be redeployed to update the FERG. Using this algorithm, we can connect the inline feedback that is in the form to their respective input field, using the local textual context connection described in subsubsection 3.1.2. However, there might be some pieces of feedback text that the algorithm is not able to connect to input fields, because it is not in the proximity of an input field, or because of the page redirect, which results in the form not being available. In both of these cases, we are dealing with *global* feedback, which is feedback that is applied to multiple fields or all of the input fields in the form. For instance, the error present at the top of Figure 1 (flight being in the United States), is not attached to any specific input.

To refine the constraints and ensure their accurate representation of the form's requirements, we initiate another prompting process with the LLM, as delineated in Figure 4. This process involves generating a new set of constraints, considering the feedback received from the previously submitted values. The feedback part can be viewed at the end of the constraint prompt in Figure 4. In this iteration, the previously submitted values and post-submission feedback are incorporated into the prompt, allowing the model to align its responses more closely with the received feedback. This includes integrating inline feedback for each element, and in instances of global feedback, incorporating it into the prompts for all elements.

This refined approach enables the LLM to adjust and fine-tune the constraints in response to the provided feedback, thereby facilitating the generation of new values that comply with these updated constraints. The form is subsequently resubmitted with these new values, and this iterative cycle is repeated until a successful form submission is achieved. At this juncture, the algorithm concludes its operation. Successfully reaching this stage allows us to assert with considerable confidence that the derived constraints mirror the actual requirements stipulated by the form.

3.2.3 Constraint Validation. After finishing the previous step, we are left with a set of constraints that can pass the form. While these constraints may have facilitated successful form submissions, they could be unnecessarily restrictive. Take, for instance, the constraints in Listing 2, particularly `toHaveLengthCondition('>', 2)`. This appears to be a reasonable constraint for the field, however, the developers might not have applied any length restrictions to this particular field. Even though any values with lengths exceeding 2 will pass the form validation, this constraint might not accurately reflect the intended logic for the form.

To validate these constraints, we leverage an iterative process where we analyze the deduced constraints for each field. During each iteration, we negate one constraint at a time while preserving all other constraints and generating corresponding values for evaluation to make sure that the constraint is a valid one for the input field. For instance, considering the `To` field and the length constraint, the corresponding validation constraint would be `.not.toHaveLengthCondition('>', 2)`.

```
expect(field('bkmgFlights_origin_trip_1'))
.not.toHaveLengthCondition('>', 2)
```

Listing 2: Air Canada's `To` Field Constraint Negation

We employ these revised constraints to generate a value for the field and proceed to submit the form. The new form submission can yield two potential outcomes:

- **Success**: This indicates that the initial constraint was ineffective and not considered by the developers. Nonetheless, as this constraint was derived from the semantic interpretation of the input field and was expected to hold, we recorded this discrepancy in a database. This acts as a notification to the developers about the discrepancy between our expectations based on semantic interpretation and the actual logic of the input field. Therefore, we keep this attempt as a test in the test generation phase.

- **Failure**: A feedback indicating a failure validates that the initial constraint was correctly inferred since its negation causes failure. We preserve the values used in the form along with the feedback (both inline and global) for generating assertions in the subsequent test generation phase.

After iterating through these steps for every input field and each associated constraint, we accumulate a database comprising discrepancies, submission success, and submission errors.

3.3 Test Generation

The overarching objective of our test generation is to cover a comprehensive range of form submission states, inclusive of both successful and failed form submissions. Each test case examines a submission state of the form-under-test (See Definition 1).

Throughout the prior stages, we have generated and validated anticipated constraints for each input field. We expect these constraints to correspond to a potential execution scenario within the form's functionality. By submitting values that either adhere to or violate each constraint, we are effectively verifying the presence of the associated execution scenarios within the form's logic. Simultaneously, we systematically record the input values used and the resulting outcomes of each form submission in a database for future reference and analysis.

According to this scheme, each set of values and submission outcome that we encountered in the previous phases can be transformed into a test case. These tests essentially function as end-to-end tests for the form under examination, ensuring its correct operation under varying input conditions. From the local relation edges, we obtain single-variable test cases, and we generate test cases for the combination of different input fields using the relations that we inferred during global relation creation. Each generated test case performs the following actions: (1) navigate to the page containing the form, (2) populate the form fields with the inferred values, (3) submit the form, and (4) assert that the submission state expected is present on the page.

3.4 Implementation

FormNexus is developed in Python and supports the integration of either the GPT-4 [6] or the Llama 2 [48] model. We opted for GPT-4 due to its established performance as one of the most advanced LLMs available, while Llama 2 was chosen for its demonstrated capabilities as a top-performing open-source LLM across various benchmarks. For textual embedding generation, we utilized the ADA architecture [38], and a standard implementation of node2vec [22] to capture the underlying graph structure. The definition of our constraint templates drew inspiration from the Jest library [4], a testing framework equipped with a comprehensive set of built-in assertions for evaluating variables under diverse conditions. By adapting these assertions to our specific requirements, we arrived at a final set of 14 constraint types. Notably, the test cases generated by FormNexus leverage the Selenium framework [5] for robust execution. The implementation of our technique, FormNexus, has been made publicly accessible [3].

4 Evaluation

We have framed the following research questions to measure the effectiveness of FormNexus:

- **RQ1**: How effective is FormNexus in generating tests for forms?
- **RQ2**: How does FormNexus compare to other techniques?
- **RQ3**: What is the contribution of FormNexus's components towards the end results?

For running our experiments, we set the temperature parameter of the LLMs to 0 to produce the same response every time.

4.1 Ground Truth

Establishing a ground truth for comparing the coverage efficacy of different test generation methods requires a reliable measure of the total number of FSSs. As automatic FSS evaluation would presume the ability to achieve perfect test coverage (which is not feasible), manual determination is necessary. This task, while potentially complex, can be approached systematically. The following procedure outlines the process for capturing FSSs within the application:

(1) **Baseline Establishment**: Each author independently identifies a set of valid input values that result in successful form submissions (an initial FSS). This serves as the initial reference point for subsequent manipulations.
(2) **Isolated Input Variation**: Each author individually mutates the value of a single input at a time, holding all others constant to their baseline values. A predetermined set of mutation rules (e.g., empty values, exceeding length limits, incorrect data types) guides this process, ensuring a consistent and focused exploration of potential new FSSs, such as FSS 2 in Table 1.
(3) **Combinatorial Input Variation**: After single-input mutations, authors explore possible scenarios triggered by specific input combinations. This stage incorporates domain knowledge and logical inference to hypothesize constraints (e.g., dependencies between `From` and `To` fields). Deliberate constraint violations are introduced to expose FSSs associated with input interplay, such as FSS 3-5 in Table 1.
(4) **Collaborative Consolidation**: The authors aggregate their independently discovered FSSs, ensuring completeness and minimizing omissions. Discrepancies in findings are discussed and resolved through collaborative analysis.

Following this systematic approach to FSS discovery, we cover a wide range of input combinations and form responses.

4.2 Dataset

Given that there exists no dataset that contains information about form values and the associated submission states, we curated and annotated a list of web forms. Drawing from the Mind2Web dataset [19], which comprises a wide array of popular websites in the US across various domains, we aimed to construct a diverse dataset. Additionally, to address tasks that are infeasible in real-world applications using automated tools, such as user creation, we integrated open-source applications into our dataset. Our selection criteria for forms included: (1) representation across a range of web application domains; (2) diversity in form types and categories; (3) the presence of input value validation, crucial for evaluating baselines and our technique's efficacy in exploring these validation scenarios; and (4)

Table 3: Dataset Categories

Category	Form Count	Input Count
Travel	8	31
Query	13	17
Registration	4	24
Data Entry	5	30
Total	30	102

forms that do not require user authentication, making them more accessible for real-world application analysis.

Our emphasis was primarily on free-form input fields such as text, number, or date inputs, necessitating value generation, rather than selection-based inputs, i.e., checkboxes or dropdowns.

Table 3 presents the range of subjects covered in our study, along with the respective counts of forms and input fields in each category. Our methodology was assessed on a total of **30** web forms, spanning **4** distinct categories of functionality. These forms incorporate a cumulative total of **102** input fields, with individual forms containing between 1 to 14 inputs, averaging at 3.4 inputs per form. Each form implements some level of validation, varying in complexity, thereby contributing to the diversity of our test dataset. Out of 30, 6 of the forms were from open-source applications containing 37 of the input fields, while the rest were from real-world applications.

4.3 Baselines

As previously stated, the field of generating test cases for web forms is sparsely explored. We considered employing Santiago et al. [41] as a baseline for our comparison. However, it was excluded from our comparison due to the unavailability of their replication package and the absence of a detailed description of their approach.

To assess our method's efficacy, we compared it with various alternative strategies. Our first approach involves a *static* module that tests pre-defined values, chosen to identify potential errors based on the input field's type attribute. For example, in numeric fields, this module inputs extremes like very large or small numbers, including zero. Additionally, we utilize Crawljax [36], a web crawler equipped with a random value generation for form inputs, to generate 20 values for each input field in our subjects. We use these values to test the forms.

An alternative approach for test case generation is directly employing the LLM. In this approach, we designed modules to prompt GPT-4 and Llama 2 with the forms' HTML, directing these models to generate both successful and erroneous test inputs for each form. This approach bypasses the additional techniques incorporated in FormNexus. We also adopt a method akin to QTypist [35] for generating a variety of test values. A direct application of QTypist was not feasible as it was not intended for testing, and the model used, the Curie version of GPT-3 [15], is no longer available for fine-tuning. Additionally, the specific dataset used for fine-tuning was not disclosed in their repository. Therefore, we utilized GPT-4 [6], applying linguistic patterns similar to those described in QTypist, and instructed the model to generate both passing and failing values for the form. This can give QTypist an advantage since GPT-4 is likely more powerful than their fine-tuned model.[1]

[1]We also considered Auto-GPT [2], but it was unable to generate form input values.

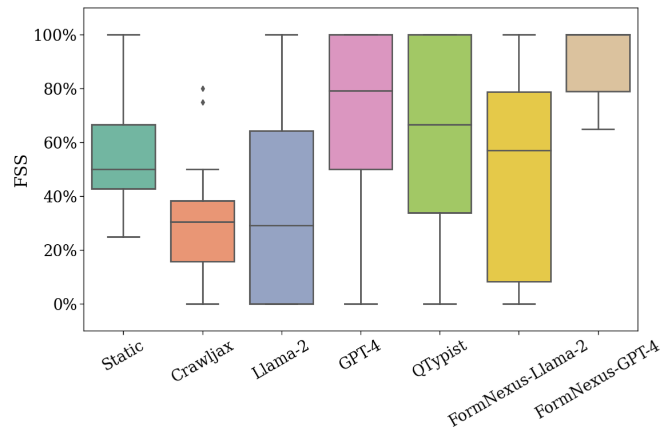


Figure 6: Box plots of FSS Coverage

4.4 RQ1: Effectiveness

Our primary objective is to cover as many states behind forms as possible. Thus, we measure effectiveness as a percentage of covered FSSs. Figure 6 illustrates the distribution of FSS coverage for different methods. Tests generated by FormNexus-GPT-4 and FormNexus-Llama 2 successfully cover **89%** and **51%** of the known FSSs respectively.

FormNexus-GPT-4 was, in most cases, effective in inferring an accurate model of the constraints on the first attempt. On average, the method reached stable constraints within **1.13** iterations. In **4** out of **30** cases was a second iteration necessary to derive a more accurate list of constraints. No instances required more than two iterations to achieve a stable constraint list.

On average, FormNexus-GPT-4 produced **3.77** constraints per input field. In the constraint validation phase, approximately **26%** of these constraints were invalidated on average, showing that these invalidated constraints were not factored into the application's design by the developers. In total, FormNexus-GPT-4 generated **389** constraints, which averages around **13.0** constraints for each form. FormNexus-Llama 2 produced **13.11** constraints per input field, **45.3** per each form, with a total of **1359** constraints, where on average around **76%** of the constraints were invalidated.

4.5 RQ2: Comparison

The data presented in Figure 6 demonstrate that FormNexus-GPT-4, achieves an average coverage rate of **89%**, which significantly surpasses the results achieved by the baselines; The static method attained a 57% coverage rate, while Crawljax only attained a 30% coverage rate. The standalone Llama 2 and GPT-4 models managed a coverage rate of 35% and 71%, respectively. FormNexus-Llama 2 can improve its accuracy to 51%. Therefore, our technique represents an 25% improvement in FSS coverage over the next best-performing baseline, namely standalone GPT-4.

It is worth noting that there were 3 out of 30 (10%) instances where GPT-4 could not generate any viable values for form inputs. Two of these cases were due to the context size of the form being exceeded, rendering the GPT-4 model unable to produce meaningful outputs. In another case, the response generated by GPT-4 was nonsensical and could not be interpreted in a useful way. Similarly,

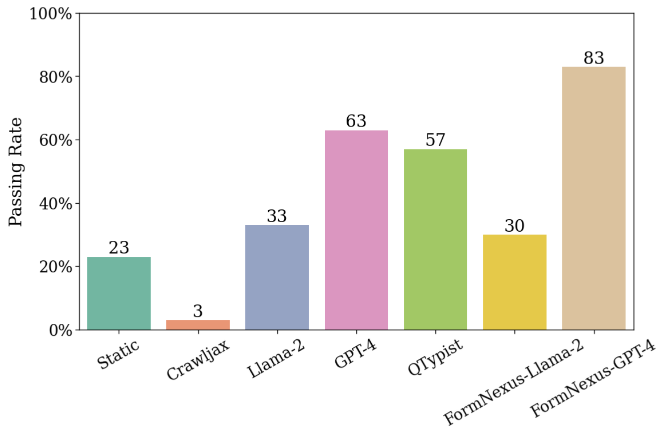


Figure 7: Passing Rates

Llama 2 was unable to produce viable responses in 14 out of 30 (46%) instances due to context size limitations. These limitations underscore the benefits of our approach. By constraining each LLM prompt to focus solely on one input and supplying contextualized information, FormNexus decreases the LLM's context size. Additionally, by structuring the constraint and value generation process into distinct steps, we delegate fewer internal processing tasks to the LLM, thereby reducing the likelihood of nonsensical or unwanted outputs.

We measure the rate of successful FSSs as the *form passing rate*, presented in Figure 7. FormNexus-GPT-4 was able to generate successful entries for **83%** of the forms, marking a notable **27%** improvement over GPT-4 model with a 63% passing rate. The static method, Crawljax, and QTypist yielded 23%, 3%, and 57% passing rates, and Llama 2 and FormNexus-Llama 2 yielded 33% and 30% respectively. Their generated values often deviated from the forms' specific requirements, limiting their success to only forms with simple validation rules.

4.6 RQ3: Ablation Study

Our approach is comprised of multiple sub-modules, each of which contributes to the final results. In addressing RQ3, our objective is to elucidate the individual contributions of these components. Specifically, for each part of the ablation, we remove one specific module while keeping everything else the same. We utilize the FormNexus-GPT-4 given its superior performance shown in the previous sections. The ablation study is conducted across all applications listed in our dataset, and the averages are presented in Table 4.

Effectiveness of FERG in Test Generation. By excluding the local textual context and relevant input context from the constraint and value generation prompts, we can quantify the extent to which FERG's information enhances the test generation results. Table 4 indicates that employing FERG and appending pertinent information to the prompt can bolster the state coverage from 82% to 89%, equating to a 9% improvement. Furthermore, the passing rate of the method increased from 70% to 83% by adding the FERG to the prompt. Even without employing FERG, FormNexus-GPT-4 still significantly outperforms (82%) the standalone GPT-4 model (71%).

Table 4: Ablation Results of FormNexus-GPT-4

Variation	Average FSS Coverage	Total Passing Rate
No FERG	82%	70%
No Date	83%	70%
No Feedback	87%	73%
No Form Context	88%	83%
FormNexus-GPT-4	89%	83%

This superior performance can be primarily attributed to FormNexus's structured workflow. By supplying the LLM with a set of constraints, we guide it toward identifying a broader range of potential validations on the input fields. Consequently, the LLM becomes capable of generating values that it would not have been able to produce without this additional guidance.

Inclusion of Date. A significant number of forms feature date-related fields, many of which contain validations to ensure the provided date falls within an appropriate time. Therefore, including the current date in the prompt, as detailed in subsection 3.2, can assist with generating test cases for these forms. As demonstrated in Table 4, incorporating the date into the prompt can increase the method's coverage from 83% to 89%. However, this increase is constrained by the fact that some web forms either lack date-related validation or date-related input fields. The average improvement for forms with date fields is around 20%, compared with the 7% overall improvement. Moreover, the inclusion of date increased the passing rate from 70% to 83%.

Effects of Feedback. As indicated in Table 4, the improvement with a feedback loop is less significant than the two previous variations since the LLM infers the correct constraints of the input fields mostly during the first iteration. Nevertheless, in the cases that required more than one iteration, we noted an average of 2% improvement in the covered FSSs. Overall, the inclusion of a feedback loop is justified, as there may be numerous real-world scenarios where the LLM is unable to accurately infer the constraints on the first attempt. By incorporating feedback into the prompt, we were able to cover successful submissions more and it contributed to increasing the passing rate from 73%.

Effects of Form Context. According to data in Table 4, form context (see Figure 5) provides a slight advantage in FFS coverage but no effect on the passing rate. This is mainly because the semantic constraints of most of the input fields are reflected in the FERG and can be inferred independently from the form context.

5 Discussion

Variations in FormNexus Effectiveness. The improvement in FSS coverage achieved by FormNexus compared to baseline results varies significantly across different categories of web forms. For example, query web forms, which typically lack the complex validations found in travel forms, usually consist of a single free-form text input and seldom give failure feedback. Consequently, we observed notable improvements in FSS coverage with FormNexus in various categories. For instance, in the context of travel forms, FormNexus-GPT-4 achieved an FSS rate of **82%**, a considerable enhancement over the 42% with GPT-4 and 36% using QTypist. In contrast, for query forms, the FSS coverage rates are 92% for

FormNexus-GPT-4, 91% for GPT-4, and 90% for QTypist. These findings indicate that FormNexus is more effective in enhancing coverage rates in complex scenarios.

Limitations. Despite its strengths, our method has certain limitations that warrant consideration. These limitations influence its discovery effectiveness in specific web form scenarios:

- When feedback from the web application lacks sufficient detail, it may hinder our approach's ability to precisely infer the constraints necessary for successful value generation.
- Our method, as currently designed, does not account for certain dynamic changes in the web application's underlying state. For example, scenarios involving repeated user creation (where most applications would generate errors) would not be detected.
- Some constraints inherently involve complex regular expressions or intricate mathematical relationships between input fields. These may prove challenging for LLMs to process directly. Integrating our method with specialized solvers could improve the handling of such constraints.
- Reliance on LLMs introduces the potential for hallucinations. While our methodology mitigates certain types (e.g., generating values for nonexistent fields), hallucinations in constraint naming remain a possibility. In practice, a similarity metric can be used to map generated constraints to a predefined list, addressing this issue.

Importantly, these limitations are not fundamental flaws in our approach; they highlight potential areas for refinement in future work. Despite these limitations, FormNexus demonstrates its effectiveness by generating tests that discover a significant number of successful and failing submission states.

Threats to Validity. One external threat to the validity of our work is the representativeness of our experimental subject selection. To mitigate this threat, we chose subjects from diverse categories of web applications and included diverse web forms.

The validity of our work may also be threatened by the populated ground truth dataset. Given the inherent difficulties in fully understanding the underlying logic of real-world web applications, the numbers we have measured might not perfectly represent the actual logic of the form. To address this issue, multiple authors independently tested the web forms and consolidated their results, aiming to minimize the possibility of missing any form submission states. However, it should be noted that regardless of the actual total number of submission states, our method has consistently demonstrated a significant improvement in discovering submission states over the baselines.

6 Related Work

Automated Form Filling. Research in automated form filling has progressed from heuristic-based methods [9, 12, 45, 46, 50] to machine learning techniques [13, 47], addressing challenges in web crawling [40] and automatic completion of web forms [26, 27, 29, 30, 39, 54]. Specialized strategies focus on mobile form-filling [10, 23, 33, 49], with LLMs like GPT-3 aiding in input generation [15, 35]. However, these do not encompass form-testing such as FormNexus, which also integrates FERG and GPT-4 for semantic comprehension of web forms.

Sparse literature on test generation for web forms includes Santiago et al.'s machine learning approach for extracting web form semantics [41], but it lacks the flexibility and real-world application of FormNexus. Form understanding is further explored in projects like OPAL [21] and studies like Zhang et al. [53], focusing on classifying input fields using various features. However, they fall short in identifying interrelationships between form elements, which is vital for capturing form complexities.

Shahbaz et al. [43] proposed generating valid and invalid string test data based on web searches and predefined regular expressions. Clerissi et al. [18] presented DBInputs, a technique for generating test inputs for web applications by exploiting the application's own database. By exploiting syntactic and semantic similarities between web page input fields and database identifiers, DBInputs automatically identifies and extracts domain-specific and application-specific inputs, overcoming the limitations of manual curation of data sources. However, our approach does not depend on direct database access or specific knowledge of the application's codebase or database schema. ARTE [8] generates realistic test inputs for web APIs by automatically extracting data from knowledge bases such as DBpedia. It uses natural language processing and knowledge extraction techniques to automate the generation of meaningful and valid test inputs for web APIs. In contrast, in this work, we focus on test generation for web forms.

LLMs. LLMs have been pivotal in various web-related tasks such as HTML understanding [25], information extraction [34], and web page summarization [17]. Language models specific to HTML like HTLM [7], Webformer [24], DOM-LM [20], and MarkupLM [32] have been developed. Their integration into software testing has been innovative [28, 31, 37, 42, 44, 51]. Mind2Web introduces a dataset with real-world web applications using LLMs, but its understanding of form semantics is limited [19]. In contrast, FormNexus leverages FERG to enrich web form semantics, enhancing form filling effectiveness.

7 Conclusion

Web form testing has been an under-explored area of research, despite its considerable potential utility for developers. In this paper, we introduced FormNexus, a novel technique for automatically generating test cases for web forms. Our approach leverages a unique technique to discern the context of input fields within forms by creating a graph called FERG. We leverage these contexts within a workflow to generate constraints for input fields, and subsequently, to generate test cases based on these constraints using LLMs. We demonstrate that FormNexus achieves an impressive **89%** submission state coverage and an **83%** form passing rate, outperforming other techniques by a minimum of 25% in coverage and 27% in passing rate.

For future work, we plan to expand our dataset and improve our work to accommodate multi-step web forms. Additionally, we plan to investigate different Graph Neural Network-based architectures for generating embeddings, with a goal to further enhance the effectiveness of FERG in identifying semantic relationships.

References

[1] 2023. Air Canada. https://www.aircanada.com/ca/en/aco/home.html. Accessed: 2023-07-01.

[2] 2023. Auto-GPT. https://github.com/Significant-Gravitas/Auto-GPT/.

[3] 2023. Form Nexus. https://github.com/parsaalian/webform-testing.

[4] 2023. Jest. https://jestjs.io/docs/expect.

[5] 2023. Selenium. https://www.selenium.dev. Accessed: 2023-07-01.

[6] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. (2023). arXiv:2303.08774

[7] Armen Aghajanyan, Dmytro Okhonko, Mike Lewis, Mandar Joshi, Hu Xu, Gargi Ghosh, and Luke Zettlemoyer. 2021. HTLM: Hyper-Text Pre-Training and Prompting of Language Models. arXiv:2107.06955

[8] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated generation of realistic test inputs for web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2022), 348–363. https://doi.org/10.1109/TSE.2022.3150618

[9] Nadia Alshahwan and Mark Harman. 2011. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 3–12. https://doi.org/10.1109/ASE.2011.6100082

[10] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[11] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 1494–1505. https://doi.org/10.1145/3510003.3510106

[12] Luciano Barbosa and Juliana Freire. 2010. Siphoning hidden-web data through keyword-based interfaces. *Journal of Information and Data Management* 1, 1 (2010), 133–133.

[13] Hichem Belgacem, Xiaochen Li, Domenico Bianculli, and Lionel Briand. 2023. A machine learning approach for automated filling of categorical fields in data entry forms. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–40. https://doi.org/10.1145/3533021

[14] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2020. Dependency-Aware Web Test Generation. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 175–185. https://doi.org/10.1109/ICST46399.2020.00027

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[16] Xiaoning Chang, Zheheng Liang, Yifei Zhang, Lei Cui, Zhenyue Long, Guoquan Wu, Yu Gao, Wei Chen, Jun Wei, and Tao Huang. 2023. A Reinforcement Learning Approach to Generating Test Cases for Web Applications. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. 13–23. https://doi.org/10.1109/AST58925.2023.00006

[17] Huan-Yuan Chen and Hong Yu. 2023. Intent-Based Web Page Summarization with Structure-Aware Chunking and Generative Language Models. In *Companion Proceedings of the ACM Web Conference 2023*. Association for Computing Machinery, 310–313. https://doi.org/10.1145/3543873.3587372

[18] Diego Clerissi, Giovanni Denaro, Marco Mobilio, and Leonardo Mariani. 2021. Plug the database & play with automatic testing: improving system testing by exploiting persistent data. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, 66–77.

[19] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. arXiv:2306.06070

[20] Xiang Deng, Prashant Shiralkar, Colin Lockard, Binxuan Huang, and Huan Sun. 2022. DOM-LM: Learning Generalizable Representations for HTML Documents. (2022). arXiv:2201.10608

[21] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, and Christian Schallhart. 2013. The ontological key: automatically understanding and integrating forms to access the deep Web. *The VLDB Journal* 22 (2013), 615–640. https://doi.org/10.1007/s00778-013-0323-0

[22] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, 855–864. https://doi.org/10.1145/2939672.2939754

[23] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280. https://doi.org/10.1109/ICSE.2019.00042

[24] Yu Guo, Zhengyi Ma, Jiaxin Mao, Hongjin Qian, Xinyu Zhang, Hao Jiang, Zhao Cao, and Zhicheng Dou. 2022. Webformer: Pre-Training with Web Pages for Information Retrieval. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, 1502–1512. https://doi.org/10.1145/3477495.3532086

[25] Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. 2022. Understanding html with large language models. (2022). arXiv:2210.03945

[26] Inma Hernández, Carlos R Rivero, and David Ruiz. 2019. Deep Web crawling: a survey. *World Wide Web* 22 (2019), 1577–1610. https://doi.org/10.1007/s11280-018-0602-1

[27] Lu Jiang, Zhaohui Wu, Qinghua Zheng, and Jun Liu. 2009. Learning deep web crawling with diverse features. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 1. IEEE, 572–575. https://doi.org/10.1109/WI-IAT.2009.96

[28] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models Are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering*. IEEE Press, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[29] Gustavo Zanini Kantorski and Carlos Alberto Heuser. 2012. Automatic Filling of Web Forms. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012 (CEUR Workshop Proceedings, Vol. 866)*. CEUR-WS.org, 215–219.

[30] Juliano Palmieri Lage, Altigran S da Silva, Paulo B Golgher, and Alberto HF Laender. 2004. Automatic generation of agents for collecting hidden web pages for data extraction. *Data & Knowledge Engineering* 49, 2 (2004), 177–196. https://doi.org/10.1016/j.datak.2003.10.003

[31] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[32] Junlong Li, Yiheng Xu, Lei Cui, and Furu Wei. 2022. MarkupLM: Pre-training of Text and Markup Language for Visually Rich Document Understanding. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 6078–6087. https://doi.org/10.18653/v1/2022.acl-long.420

[33] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[34] Zimeng Li, Bo Shao, Linjun Shou, Ming Gong, Gen Li, and Daxin Jiang. 2023. WIERT: Web Information Extraction via Render Tree. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13166–13173. https://doi.org/10.1609/aaai.v37i11.26546

[35] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367. https://doi.org/10.1109/ICSE48619.2023.00119

[36] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web* 6, 1, Article 3 (mar 2012), 30 pages. https://doi.org/10.1145/2109205.2109208

[37] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[38] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. (2022). arXiv:2201.10005

[39] Alexandros Ntoulas, Petros Zerfos, and Junghoo Cho. 2005. Downloading Textual Hidden Web Content through Keyword Queries. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '05)*. Association for Computing Machinery, 100–109. https://doi.org/10.1145/1065385.1065407

[40] Sriram Raghavan and Hector Garcia-Molina. 2001. Crawling the Hidden Web. In *Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 129–138. https://doi.org/10.5555/645927.672025

[41] Dionny Santiago, Justin Phillips, Patrick Alt, Brian Muras, Tariq M King, and Peter J Clarke. 2019. Machine learning and constraint solving for automated form testing. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 217–227. https://doi.org/10.1109/ISSRE.2019.00030

[42] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. (2023). arXiv:2302.06527

[43] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. 2015. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Sci. Comput. Program.* 97, P4 (2015), 405–425. https://doi.org/10.1016/j.scico.2014.04.008

[44] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. (2023). arXiv:2305.00418

[45] Moumie Soulemane, Mohammad Rafiuzzaman, and Hasan Mahmud. 2012. Crawling the hidden web: An approach to dynamic web indexing. *International Journal of Computer Applications* 55, 1 (2012). https://doi.org/10.5120/8717-7290

[46] Ben Spencer, Michael Benedikt, and Pierre Senellart. 2018. Form filling based on constraint solving. In *Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18*. Springer, 95–113. https://doi.org/10.1007/978-3-319-91662-0_7

[47] Guilherme A Toda, Eli Cortez, Altigran S da Silva, and Edleno de Moura. 2010. A probabilistic approach for automatically filling form-based web interfaces. *Proceedings of the VLDB Endowment* 4, 3 (2010), 151–160. https://doi.org/10.14778/1929861.1929862

[48] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. (2023). arXiv:2307.09288

[49] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1232–1243. https://doi.org/10.1145/2660267.2660372

[50] Tanapuch Wanwarang, Nataniel P Borges Jr, Leon Bettscheider, and Andreas Zeller. 2020. Testing apps with real-world inputs. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 1–10. https://doi.org/10.1145/3387903.3389310

[51] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. (2023). arXiv:2305.04764

[52] Rahul Krishna Yandrapally and Ali Mesbah. 2023. Fragment-Based Test Generation for Web Apps. *IEEE Transactions on Software Engineering* 49, 3 (2023), 1086–1101. https://doi.org/10.1109/TSE.2022.3171295

[53] Shaokun Zhang, Yuanchun Li, Weixiang Yan, Yao Guo, and Xiangqun Chen. 2021. Dependency-aware Form Understanding. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 139–149. https://doi.org/10.1109/ISSRE52982.2021.00026

[54] Qinghua Zheng, Zhaohui Wu, Xiaocheng Cheng, Lu Jiang, and Jun Liu. 2013. Learning to crawl deep web. *Information Systems* 38, 6 (2013), 801–819. https://doi.org/10.1016/j.is.2013.02.001

Received 2024-04-12; accepted 2024-07-03