Etna: Harvesting Action Graphs from Websites

Oriana Riva
Microsoft Research

Jason Kace
Microsoft Research

ABSTRACT

Knowledge bases, such as Google knowledge graph, contain millions of entities (people, places, etc.) and billions of facts about them. While much is known about *entities*, little is known about the *actions* these entities relate to. On the other hand, the Web has lots of information about human tasks. A website for restaurant reservations, for example, implicitly knows about various restaurant-related actions (making reservations, delivering food, etc.), the inputs these actions require and their expected output; it can also be automated to execute those actions. To harvest *action knowledge* from websites, we propose *Etna*. Users demonstrate how to accomplish various tasks in a website, and Etna constructs an action-state model of the website visualized as an action graph. An action graph includes definitions of tasks and actions, knowledge about their start/end states, and execution scripts for their automation. We report on our experience in building action-state models of many commercial websites and use cases that leveraged them.

CCS CONCEPTS

• **Information systems** → **Web mining**; • **Human-centered computing**;

KEYWORDS

Action graphs; graphical user interfaces; web; programming by demonstration; UI automation.

ACM Reference Format:
Oriana Riva and Jason Kace. 2021. Etna: Harvesting Action Graphs from Websites. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3472749.3474752

1 INTRODUCTION

Extracting machine-understandable knowledge from the web has been a long-standing research goal [22, 117]. Techniques for automatically extracting *entities* from websites have been investigated extensively [9, 12, 23, 28, 89, 109, 116]. This enabled the creation of knowledge bases such as Bing and Google's knowledge graphs that today include many millions of entities (people, places, organizations, etc.) with billions of facts about them (attribute values and relationships with other entities). In contrast, understanding and representing a website in terms of the functionality or *actions* it supports has received little attention. The only action-related information found in even the largest knowledge graphs [2, 16, 30, 43] consists of general action verbs that may be associated with an entity, e.g., the verb "play" is related to the entity "video". Information on the input parameters an action supports or how an action relates to other actions is completely missing. More importantly, while the value of an entity fact lies directly in the information it stores (e.g., Robert De Niro is an *actor* born in *1943*), the value of an action lies *also* in the output it generates upon execution. Hence, besides its name, scheme, and relationships, we must also capture information on how to execute an action (by calling a service or interacting with an application) and on the expected output.

Action-related knowledge can benefit many user experiences. Users use search engines to acquire information but also to complete tasks. Generating actionable outputs can increase user satisfaction [14]. For example, if a user searches for "price of a ride to Boston airport", today's search engines will likely return links to the Uber price estimator and various transportation websites. Now imagine if the engine knew about a "get-ride-estimate" action that could be executed on the Uber website by supplying an "origin" and "destination" as inputs. It could then map the user's query to a parametrized action, execute it by automatically driving the website UI, and either return the price sheet or re-direct the user to the result page. Knowing the actions a website supports can also benefit AI assistants and chatbots by adding flexibility and expanding their capabilities beyond API-based functionality, as demonstrated by Google's Duplex on Web [112], a voice assistant that makes car reservations by controlling the UI of a car rental website.

To extract action-related knowledge from a website, one may leverage automatic methods ranging from unsupervised to supervised learning models. A reinforcement learning agent can, in principle, learn correct sequences of UI interactions to complete a task in a website by trial and error. However, such an agent may take a long time to converge unless it executes in simplified environments and is pre-trained using expert demonstrations [34, 71]. Further, rewarding an agent based on task completion is challenging to automate [65], and accuracy is likely to unacceptably degrade with complex tasks or an increasingly diverse set of websites. Graph neural networks [122] and action learning techniques [36, 66] could also be leveraged. However, these approaches typically require large amounts of labeled interaction traces, which developers are often unwilling [52] or unable [87] to provide. Thus, their precision may be insufficient for reliable action execution. Crowdsourcing data collection is an option in some cases [24, 25], but it requires infrastructure and has cost implications.

Challenges and overview. In this paper, we focus on the problem of how to enable developers to extract action-related knowledge from websites in diverse domains by providing a handful of task demonstrations. To this end we leverage *programming by demonstration* techniques which have been relatively successful at supporting web UI automation and data scraping [4, 18, 40, 46, 57, 58, 98, 99].

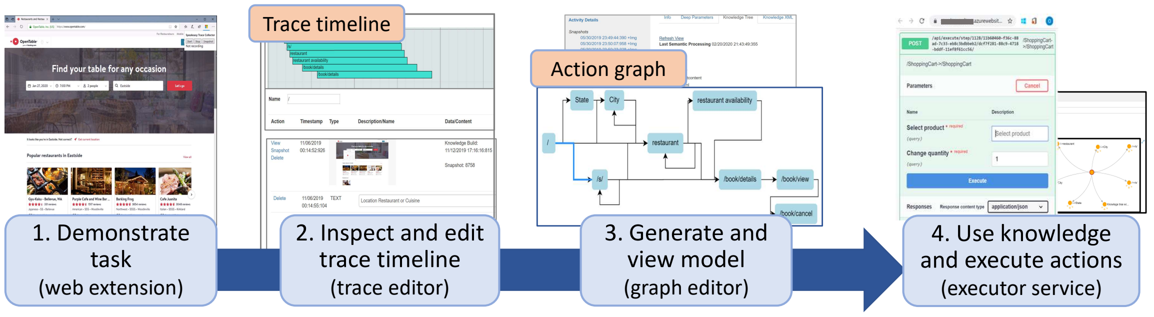

Trace timeline
Action graph
1. Demonstrate task (web extension)
2. Inspect and edit trace timeline (trace editor)
3. Generate and view model (graph editor)
4. Use knowledge and execute actions (executor service)

Figure 1: Etna workflow: *(i)* Users demonstrate tasks using the website UI; *(ii)* From a task demonstration Etna builds a timeline of states and transitioning actions, which users can modify; *(iii)* From trace timelines Etna constructs an action-state model, represented as an action graph; *(iv)* Users use the extracted action knowledge and execute actions through the executor service.

These techniques allow a user to record a task interaction in a website as a sequence of UI events, and to obtain a UI script to replay it. This is a natural form of programming, since users perform the actions in their familiar environment [59]. However, these techniques come with limitations. First, they lack *semantic representation* and *generalization*: actions are usually represented at the level of raw UI events (clicks, text inputs, selections, etc.), and do not generalize into parametrizable functions. Some of these tools can capture action semantics [20], but only by using application instrumentation. Second, they lack the concepts of *start* and *end states*, and assume that replaying the same sequence of recorded UI events will always produce the same outcome, which is often not the case (e.g., making a shuttle reservation at different times may lead to a different list of results depending on schedule and availability). Overall, content variability together with periodic updates to websites leads to poor *robustness* in action replay.

To address these issues, we propose *Etna*, a programming by demonstration approach for extracting action-related data from websites, coupled with an executor service to support execution of the extracted actions. From task demonstrations in a website of interest, Etna produces an *action-state space model* of the application, visualized as an *action graph*. Every vertex in the graph represents a unique state in the website (e.g., a page or UI change) and every edge corresponds to the actions that trigger a state transition. Etna represents actions as parametrizable functions: they have a set of required input parameters and each input has a schema (a type and, if applicable, a set of possible values, such as the sizes of a product).

System design. We designed Etna to operate as summarized in Fig. 1, with the following principles in mind:

- *Partial automation of model building*: Ideally, Etna should infer action graphs from task demonstrations automatically. Full automation, however, inevitably requires many task demonstrations to resolve possible ambiguities [52]. A one-size-fits-all approach is also hard when dealing with websites from diverse domains and with many different UI structures. In automating the model building process we observed that automation often caused errors that were hard to debug, while we could have easily handled them early in the processing pipeline. We therefore designed Etna to operate in two stages. First, it transforms individual interaction traces into timelines of visited states with transitioning actions. Then, it automatically aggregates timelines to build a model and visualizes it as an action graph. While action graphs can be edited, we expect users to do most of their edits via the timeline interface, which is more natural than the abstracted and aggregated view of action graphs, as it resembles the user's mental model at recording time.
- *Customization and iterative modeling*: There is often more than one way to model a task, and a model may need to be updated over time. Depending on the intended use of the final model, one may decide to break down a state into multiple sub-states to capture an application's functionality at a finer granularity, or vice-versa to merge less relevant states. For example, in a recorded interaction in a shuttle reservation website, a user may first pick a shuttle route and then select pickup and dropoff locations. The interaction may be represented as a single action with three inputs (route, pickup and dropoff) or as three consecutive actions (set_route, set_pickup and set_dropoff) to enforce the order in which inputs must be specified and to allow for validation. We designed the timeline view such that users can customize their task flow by creating or merging states of the automatically-generated flow. Moreover, users can generate a model from a few traces and later add new ones (or replace existing ones), thus being able to control the quality and completeness of the final model and to evolve it over time.
- *Robust action execution*: The novelty of Etna lies in the *combination* of action modeling (extracting comprehensive action knowledge) and execution (executing extracted actions reliably). To achieve robustness, Etna does the following: *(i)* it ensures all input parameters associated with an action are extracted from demonstration traces by introducing *multi-input events* specifically designed to handle nested list selections; *(ii)* for every action, in addition to UI scripts, it extracts *parameterizable deeplinks*, which can be faster and more reliable; and *(iii)* it introduces *semantic ranking*, a novel approach to locate UI elements by progressively limiting the search context using element attributes, and by searching for *text strings* (e.g., a restaurant name) rather than fixed attributes (e.g., a class name).

Contributions. Over the past four years, we have used the data captured by Etna to support various use cases that we illustrate in §4. We have used Etna to *(i)* create APIs for web data extraction, *(ii)* build a runtime to generate chatbots without any coding, *(iii)* implement an agent for resilient robotic process automation (RPA), and *(iv)* train natural language interfaces for web navigation.

Our goal in this paper is to demonstrate the richness and utility of action-related web data and to propose a relatively simple workflow for extracting it. We make the following contributions:

(1) the action graph concept, including action and state labels, action dependencies, semantic knowledge for every state in the graph, and parametrizable action functions which can handle multi-input events (§3.2 and §3.3);
(2) an automated approach to generate parameterizable deeplinks and UI scripts along with a semantic ranking technique to reliably execute actions in real-world websites (§3.4);
(3) an implementation that, unlike prior approaches [17, 20], does not require website instrumentation, and that can work across a variety of websites; and
(4) an informal evaluation (§5) of how Etna has served our needs so far and four use cases (§4) demonstrating the utility of Etna's action knowledge.

2 BACKGROUND AND RELATED WORK

In this section, we review both programming by demonstration and ML techniques that studied how to model application tasks and automate their execution.

2.1 Programming by demonstration tools for UI automation

Programming by demonstration (PBD) tools for UI automation [4, 40, 46, 50, 57, 58, 70, 98, 99, 107, 113] allow users to synthesize UI scripts from recordings of task interactions (demonstrations), without writing any code. With a few exceptions [54, 63, 64], these tools have been used primarily for testing and web data scraping [18, 55, 67]. Recently, they have found new application in the area of RPA (robotic process automation) [6, 11, 111, 114] where they are used to automate repetitive tasks such as invoice processing, payroll operations, and other back-office operations [51]. UI automation is also relevant to AI assistants to execute user tasks directly through web interfaces, as an alternative to connecting to APIs [112].

Etna is a PBD tool, and RPA and AI assistants for web navigation are two areas we explored using it. To execute actions, Etna generates UI scripts, but, as UI scripts can fail due to layout changes and content updates, for more robustness it takes an approach similar to Ringer [7] and extends it with semantic ranking (§3.4). A fundamental difference between Etna and prior PBD tools is Etna's concept of *application state* as the basic building block of action-state models. Traditional PBD tools represent a task as a sequence of UI events and implicitly assume that replaying them will result in the same state as in the recorded process. This is often not true. For example, specifying different values for an input parameter may "branch" the application into different states supporting different actions. Different input values may also lead to error states. The concept of application state and the data that Etna collects to characterize states allowed us to design more robust RPA (§4.3).

Another difference between Etna and prior PBD automation tools is that Etna processes recorded UI events not only to produce replayable UI scripts. It semantically represents events into high-level actions and inputs, and generalizes them into parametrizable functions. For example, a traditional PBD tool may record that a user clicked on *Hard Rock Cafe* in a list; instead, Etna aims to infer that the user selected a *restaurant name* and learn the sequence of UI interactions necessary to select *any other restaurant name* in the same list. Some PBD tools have tackled a similar challenge. SUGILITE [63], Peridot[86] and CoScripter [57] use simple heuristics to achieve generalization, but they have limited applicability. Others use AI-based program synthesis techniques [33, 53, 81], but end up requiring many task demonstrations and producing results that may be hard to interpret. APPINITE [59] and PUMICE [60] semantically represent data inputs by asking users to express their intentions in natural language. Inspired by lessons learned for usable AI [52], Etna compromises between the demonstration effort and the level of automation achievable; it generalizes task flows through a structural and semantic analysis of the DOM tree and allows users to control the final model through a timeline interface.

Computer-vision-based UI automators have also been proposed [42, 101, 119]. Vasta [101], for example, records bounding boxes of interacted UI elements during user demonstration and uses the RetinaNet [68] object detector to lookup UI elements at replay time. Bernal-Cárdenas et al. [8] replay video recordings. Etna currently relies only on OCR for visual features, but ongoing work is exploring how to integrate computer vision techniques [92, 93].

2.2 Inferring task models from UI traces

Prior work has studied how to infer general task models from interaction traces collected through crowdsourcing or expert demonstrations, for various purposes including *(i)* task completion [17, 62], *(ii)* automated testing [20, 21, 69, 76], and *(ii)* UI design [24–26].

Task completion. Kite [62] mines Android interaction traces to extract task templates, and represents them as graphs of actions. Task templates are used to bootstrap the logical flow of a dialogue system. Unlike Etna's action graphs, Kite's graphs are not executable. Moreover, Kite's states correspond to Android Activities, while Etna defines states at a finer granularity (e.g., a shopping cart page may correspond to two states with different actions, depending on whether the cart is empty or filled). W-graphs [17] encode multiple demonstrations of a fixed 3D design task in a workflow graph. At the high-level W-graphs are similar to Etna's graphs but serve a different purpose. W-graphs can be used to provide suggestions on how to perform portions of a complex task for which alternative methods exist or to identify the most efficient way to complete the task. W-graphs require application instrumentation. In contrast, Etna is a general-purpose tool (no instrumentation required) aimed to support task modeling but also task execution.

Testing. Techniques have been proposed to model the behavior of a GUI as event-flow graphs [15, 20, 27, 80], finite state machines [88, 102], and Petri nets [95]. These models are used to automatically generate GUI test cases. As their goal is testing, unlike Etna's action graphs, they tend to be low-level, and do not include state/action semantics or action/input schemes. To name a few, SwiftHand [21] learns "approximate" models of Android apps where state transitions consist of low-level UI actions (scroll, type, click, etc.) and two UI screens are considered equivalent (the same state) if they present the same set of enabled user inputs. State identification in Etna considers many more features, thus being more accurate (see §5.2 for a comparison), and state/actions are semantically represented. Ermuth and Pradel [27] define macro events that summarize sequences of low-level UI events corresponding to a single logical step (e.g., selecting an item from a menu), but these are still low-level actions and not parameterizable functions as in Etna. Chen et al. [20] provide application testers with an interactive event-flow graph that tracks and aggregates every tester's interactions with the goal of summarizing the navigation paths that have already been explored. In this case, state transitions are represented as high-level intents similar to Etna's actions. However, building the event-flow graph requires manually instrumenting a target web application to capture events for certain DOM nodes, thus hindering scalability (only 3 web applications were instrumented). In contrast, Etna is distributed as an app-agnostic web extension, which does not require instrumentation.

UI design. ERICA [24], Rico [25], and ZIPT [26] mine user interaction traces of Android apps to infer task flows and UI design patterns. Similar to Kite, they assume application states correspond to Android Activities. Moreover, they model only atomic tasks. For example, while Etna models the task of booking a hotel, they model the tasks of searching a term or adding a new user to a contact list.

2.3 Action learning and GNNs

ML techniques have been proposed to automatically learn actions from a variety of data inputs including text [5, 13, 29, 45, 74, 79, 104], videos [96, 121], and images [47, 75]. Human action recognition enables many applications, including action-centric video retrieval [31], intelligent visual surveillance [32], code extraction [121], and cooking recipe captioning [108]. In much of this work the focus is on recognizing and modeling actions, but not on *executing* them. For example, one approach uses text appearing in recipes to construct an action graph of the recipe describing what actions should be performed on which ingredients in sequence [48, 84]. The concept of action graph in Etna is substantially different; Etna's goal is not only to obtain action descriptions but also parametrizable and executable action procedures.

Related to Etna is work on learning actions defined as executable UI workflows. Li et al. [66] and He et al. [36] learn action procedures from large collections of crowdsourced (Android) interaction traces. Branavan et al. [13] use reinforcement learning to extract action procedures for a Windows application and a puzzle game, but rely on a custom reward function. More recently, reinforcement learning agents that learn to navigate the web [34, 71, 103] have been proposed. Overall, the data required for training these models (10–200 demonstrations per each task to learn [71, 103]) is not feasible in many cases, when developers are unwilling to provide enough demonstrations [52] or unable to provide samples that are sufficiently different from each other [56, 87]. Another issue with some of this work is that it assumes restricted execution environments, such as single applications [13] or simplified settings (MiniWob consisting of static HTML pages with only 10–50 UI elements [103]). On the other hand, we borrow from this work the idea of modeling websites as action-state spaces.

Graph neural networks (GNNs) [122] could be used to build action graphs. However, GNNs require millions of labeled items to train a dedicated model for each graph dataset (a website in our case) which is not usually transferable to out-of-domain data [91]. Learning GNNs with self-supervision [49, 118] would still require large amounts of data to capture the dynamism of web content and all combinations of inputs a user may submit during interaction. Instead, we target scenarios in which developers may be able to provide only small amounts of demonstrations. Work on GNN pre-training [39, 91] could enable pre-training of domain-specific GNNs (e.g., the restaurant domain) and fine-tuning on specific websites in that domain. However, learning domain-specific GNNs can be challenging without manual data alignment because actions supported by different websites in the same domain manifest with different representations and parameter sets (e.g., in OpenTable the "search-restaurant" action supports four parameters while in Yelp it requires two parameters of different types) [77].

2.4 Action knowledge graphs

An action knowledge graph could benefit many services including search engines that need to detect user intent from search queries [10]. At present, such action knowledge graph does not exist. Even in enterprise knowledge graphs [16, 30], actions appear only in the form of entity-related verbs. Schema.org [2], which is a vocabulary of entity types used to annotate webpages with structured data, contains definitions of some generic action verbs (e.g., buy, read, order, etc.), which are not as fine-grained and as comprehensive as Etna's action definitions. Commonsense knowledge graphs, such as ConceptNet [106] and Atomic2020 [41] as well as E-Commerce knowledge graphs [3, 73], contain textual descriptions of entities and events, and do not focus on actions either.

The first step in constructing a knowledge graph is to gather data. Prior work on automated navigation of websites for (deep) web crawling [35, 37, 44, 78, 94] cannot be directly leveraged to capture actions. First, these crawlers are programmed to maximize coverage of content, while a crawler of actions must maximize completion of tasks. However, automatically verifying task completion without app-specific assumptions is challenging [65]. Second, web crawlers navigate by using hyperlinks and in some cases form filling, but they ignore most of the interactable DOM elements in a webpage, which are required to support a large range of tasks. Glider [65], an approach based on reinforcement learning to extract UI scripts (called tasklets) for web tasks specified in natural language, is a first attempt towards building a web action crawler. It currently works only on simple 2-4 step tasks such as performing unit conversions or filling in a form for flight search, but could otherwise complement Etna to reduce the demonstration effort and to automatically update extracted UI scripts.

3 SYSTEM DESIGN AND IMPLEMENTATION

Fig. 1 summarizes the 4-step process a user follows when using Etna: *(i)* Using the Etna recorder, the user records a task interaction. One long recording or many short recordings can be provided, depending on the task type and complexity. *(ii)* From the sequence of UI views and events logged during demonstration, Etna generates a trace timeline. At this stage, the user can modify the task flow,

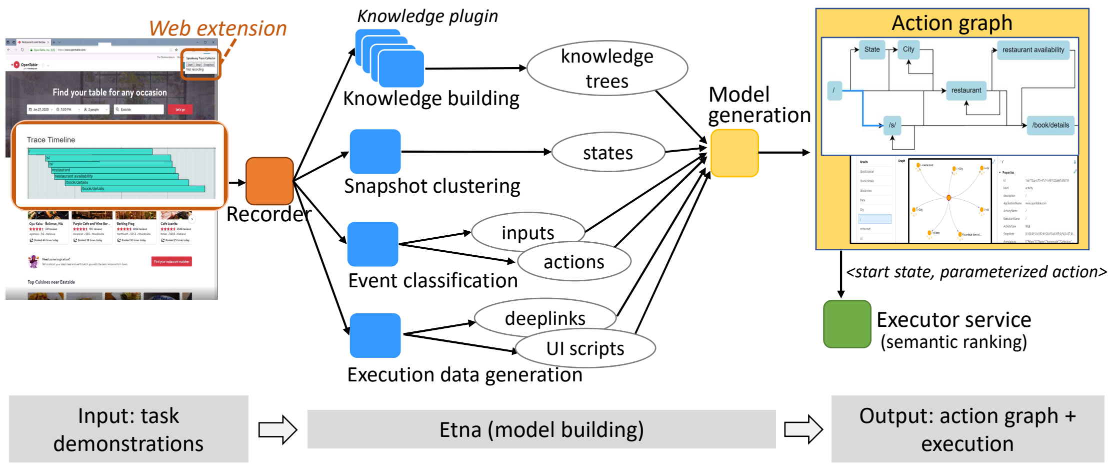


Figure 2: The Etna workflow. Using the Etna recorder, a user demonstrates a task and generates interaction traces. A trace consists of a sequence of UI snapshots and UI events. Etna processes each UI snapshot to compute various UI and semantic features (stored in knowledge trees), and uses them to cluster UI snapshots in distinct states. Etna classifies UI events into input- or action-related events, and generates corresponding deeplinks and UI scripts for execution. After processing all traces, ETNA combines them to produce an action-state model of the website visualized as an action graph (accessible via the Etna app or through a Gremlin API). An executor service is provided to parametrize and execute the extracted actions.

rename states and actions, correct UI event assignments, etc. *(iii)* From the trace timeline(s), Etna constructs an action-state model of the application visualized as an action graph. An action graph consists of a set of unique *states* and *actions*. A state may correspond to a webpage or a significant UI change in a webpage. An action causes a state transition and corresponds to one or more UI-level interactions (click, type, select, etc.). A task consists of a sequence of actions and corresponds to a *path* from a start state to a goal state. *(iv)* Finally, the user uses the generated model. Through the graph explorer, a user can browse the action knowledge or select a path in the graph, parametrize it, and launch its execution in a browser. Alternatively, the user can access the model from a graph database or export its knowledge.

Overall, Etna consists of three main components: *(i)* a recorder, implemented as a web browser extension in Microsoft Edge, *(ii)* a model builder, implemented as a web app hosted in Microsoft Azure, to process traces and support visualization/editing of timelines and action graphs, and *(iii)* an executor service, using the Selenium WebDriver [99], to execute tasks. In the following, we illustrate every step in the Etna workflow, including all components and sub-components involved in the process, as summarized in Fig. 2.

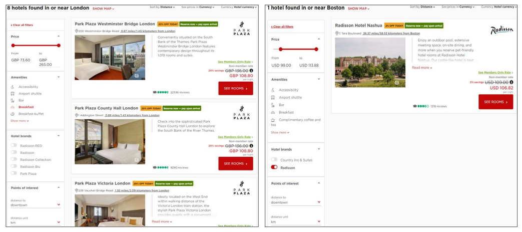
3.1 Step 1: Acquiring task execution traces

In this step, the user (e.g., a developer or a crowdworker) opens the web extension and starts a demonstration, for which a unique trace identifier is returned. Etna's recorder is comparable to those of prior UI automation tools [18, 99]. In the browser window, the user demonstrates a task such as searching for a product in a shopping website by providing a search term and then filtering the results, or ordering food for pick up by entering data fields in a form. As the user interacts with the website, the extension collects and reports *UI snapshots* and *UI events* to the model builder service.

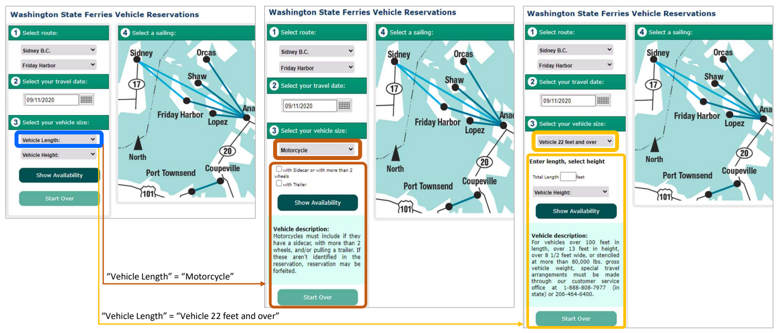
UI snapshots. A UI snapshot represents the state of the interaction at a single point in time. Soon after a UI event is reported, a UI snapshot is captured. It consists of a screenshot of the current webpage and a hierarchical model of all UI elements in the page, derived from the DOM tree. Each UI element is defined by a set of properties including: *(1)* custom identifiers from DOM attributes such as identifier, classname, etc., *(2)* text description inferred using a combination of attributes (text, placeholder, value, etc.), *(3)* standardized labels such as HTML data attributes [115], *(4)* element type, whether the element is "clickable" or "editable" inferred from various HTML tags (p, input, button, etc.), *(5)* x/y/z layout locations, *(6)* height and width, *(7)* visibility/opacity, and *(8)* style information such as font color and size.

UI events. Etna logs UI events of the following types: *(i)* click: clicking on a UI element such as a button or a hyperlink; *(ii)* select: selecting a sub-item of an element, such as an item in a menu; *(iii)* type: typing text into an editable UI element such as a text field; and *(iv)* enter: submitting the content in an editable element by pressing the "Enter" keyboard key. Each UI event is timestamped and tagged with a unique identifier. An event is named after the attributes of the corresponding UI element (name, placeholder text or label) or of elements that are close in the UI tree. Later, this name is used to label the inferred action/input associated with it.

Long vs. short demonstrations. A user can decide to record one long task interaction or break it down in multiple short traces. We recommend recording multiple short traces (typically 5–10), where every trace captures a specific step or aspect of the task. This approach allows for iterative modeling and makes it easier to maintain and scale the trace collection. For example, when capturing traces for a restaurant reservation task, a user may proceed by first demonstrating how to *search* for a restaurant of interest by specifying a location or a cuisine. Then, the user may show


8 hotels found in or near London
1 hotel found in or near Boston
Park Plaza Westminster Bridge London
Park Plaza County Hall London
Park Plaza Victoria London
Radisson Hotel Nashua

(a) Screenshots of 2 UI snapshots from a hotel website classified as the same *Search Results* state.


Washington State Ferries Vehicle Reservations
Select route:
Select a sailing:
Select your travel date:
Select your vehicle size:
Vehicle Length:
Vehicle Height:
Show Availability
Start Over
Motorcycle
Vehicle 22 feet and over
"Vehicle Length" = "Motorcycle"
"Vehicle Length" = "Vehicle 22 feet and over"

(b) Screenshots of 3 UI snapshots from a ferry reservation website classified as 3 distinct states. Selecting different values for the input *Vehicle Length* (blue box) leads to states with different actions (orange and yellow boxes).

Figure 3: Examples of UI snapshot classification.

how to filter *search results* (e.g., by neighborhood, by price, etc.) or sort them (e.g., by distance). Third, the user may pick a specific restaurant and view the menu or query it for availability in the *restaurant profile* page. One final trace may capture how to complete the *restaurant reservation*. Breaking the recording in multiple traces allows a user to record different parts of the task at different granularities (e.g., more traces for a page supporting many actions such as the search results page) and later extend the model with new states (e.g., canceling a reservation).

To allow the user to have the most natural interaction with the website, the recorder is optimized to collect data and upload it asynchronously to the remote service. UI snapshots are captured seamlessly, any time a UI interaction is detected or every time a page with a new deeplink is loaded. Finally, back transitions are allowed and automatically removed at processing time.

3.2 Step 2. Inferring states and actions

During demonstration or once the recorder is stopped, the user can use the provided trace identifier to look up the generated timeline in the Etna web app (Fig. 5). The model building service generates this timeline by processing the captured UI snapshots and UI events with two goals: *(i)* clustering the UI snapshots into a set of distinct states that were traversed during the interaction, and *(ii)* identifying from raw UI events the logical actions that connect the visited states.

Snapshot clustering and knowledge trees. A state may represent a page/view in a website or a significant UI change. This is an important difference from previous work [62] in which states correspond to uniquely-named application pages (e.g., Android Activity). Fig. 3 illustrates the challenge. In case *(a)*, the 2 snapshots represent the same state, i.e., a search results page with a different number of results. In case *(b)*, the 3 snapshots are visually and structurally similar, but represent 3 distinct states: by selecting a different value for the input parameter *Vehicle Length* (in the first screen) the application "branches" into two different states supporting different actions (for a vehicle, but not a motorcycle, the submit action requires both length and height of the vehicle). Prior work relies on page identifiers, such as deeplinks and class names [62], or the set of interactable UI elements [21] to distinguish between application states. However, deeplinks may not be supported by all websites and webpages with the same deeplink may correspond to multiple states (the snapshots in Fig. 3b have indeed the same deeplink). Comparing the set of UI elements in each webpage may also be misleading because UI controls may vary depending on the content (e.g., buttons associated with search results, as in Fig. 3a) and because developers often hide dead or alternate user views in the DOM tree.

To provide a robust but also general-purpose solution, for each recorded snapshot, Etna extracts a combination of textual, visual, and structural features, including the following:

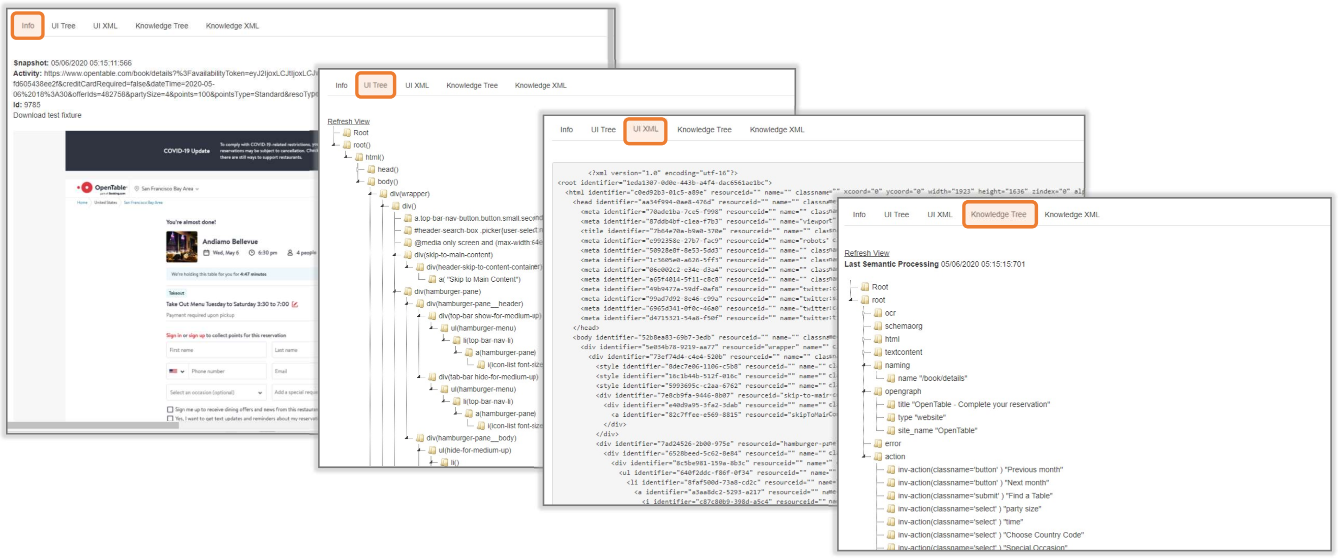
(1) *application-provided labels*, such as URLs and deeplink parameters and page identifiers such as SEO meta-tags [85];
(2) *semantic features*, extracted from the DOM tree such as HTML, Open Graph [1] and Schema.org [2] annotations;
(3) *set of interactable UI elements*, such as buttons and input fields present in the DOM tree; and
(4) *text content*, obtained by invoking an optical character recognition (OCR) API to detect content visible to the user.

Each type of feature is computed, in parallel, by a separate **knowledge plugin**, and each plugin is responsible for maintaining a distinct subtree of a hierarchical **knowledge tree** associated with every snapshot. Fig. 4 shows the information associated with a UI snapshot including a knowledge tree snippet.

Given N features $\{f_1, ..., f_N\}$ the **snapshot similarity** between snapshot s_j and s_k is then computed as follows

$$snapSim(s_j, s_k) = \frac{1}{N_a} \sum_{i=1}^{N} w_i \times simScore_{f_i}(s_j, s_k) \qquad (1)$$

where $simScore_{f_i}$ is the similarity score computed based on feature f_i and $w_i \in [0, 1]$ is the associated weight. Each $simScore_i$ is computed differently for each type of feature. For application-specific labels and SEO tags, similarity is based on exact match. For HTML, Open Graph and Schema.org annotations as well as textual content the similarity score measures the percentage of overlapping annotations or texts, respectively. For the sets of interactable UI elements, the similarity score measures the percentage of overlapping UI elements but each percentage is weighted differently



Snapshot: 05/06/2020 05:15:11:566
Activity: https://www.opentable.com/book/details?%3FavailabilityToken=eyJ2IjoxLCJtIjoxLCJ
fd605438ee2f&creditCardRequired=false&dateTime=2020-05-
06%2018%3A30&offerIds=482758&partySize=4&points=100&pointsType=Standard&resoType
Id: 9785
Download test fixture

```
Root
root()
  html()
    head()
    body()
      div(wrapper)
        div()
          a.top-bar-nav-button.button.small.second
          #header-search-box .picker{user-select:n
          @media only screen and (max-width:64e
          div(skip-to-main-content)
            div(header-skip-to-content-container)
              a( "Skip to Main Content")
          div(hamburger-pane)
            div(hamburger-pane__header)
              div(top-bar show-for-medium-up)
                ul(hamburger-menu)
                  li(top-bar-nav-li)
                    a(hamburger-pane)
                      i(icon-list font-size
              div(tab-bar hide-for-medium-up)
                ul(hamburger-menu)
                  li(top-bar-nav-li)
                    a(hamburger-pane)
                      i(icon-list font-size
            div(hamburger-pane__body)
              ul(hide-for-medium-up)
                li()
```

```
    <?xml version="1.0" encoding="utf-16"?>
<root identifier="1eda1307-0d0e-443b-a4f4-dac6561ae1bc">
  <html identifier="c0ed92b3-01c5-a89e" resourceid="" name="" classname="" xcoord="0" ycoord="0" width="1923" height="1636" zindex="0" alp
    <head identifier="aa34f994-0ae8-476d" resourceid="" name="" classname
      <meta identifier="70ade1ba-7ce5-f998" resourceid="" name="" classna
      <meta identifier="87ddb4bf-c1ea-f7b3" resourceid="" name="viewport"
      <title identifier="7b64e70a-b9a0-370e" resourceid="" name="" classn
      <meta identifier="e992358e-27b7-fac9" resourceid="" name="robots" c
      <meta identifier="50928e8f-8e53-5dd3" resourceid="" name="" classna
      <meta identifier="1c3605e0-a626-5ff3" resourceid="" name="" classna
      <meta identifier="06e002c2-e34e-d3a4" resourceid="" name="" classna
      <meta identifier="a65f4014-5f11-c8c8" resourceid="" name="" classna
      <meta identifier="49b9477a-59df-0af8" resourceid="" name="twitter:c
      <meta identifier="99ad7d92-8e46-c99a" resourceid="" name="twitter:s
      <meta identifier="6965d341-0f0c-46a0" resourceid="" name="twitter:c
      <meta identifier="d4715321-54a8-f50f" resourceid="" name="twitter:t
    </head>
    <body identifier="52b8ea83-69b7-3edb" resourceid="" name="" classname
      <div identifier="5e034b78-9219-aa77" resourceid="wrapper" name="" c
        <div identifier="73ef74d4-c4e4-520b" resourceid="" name="" classn
          <style identifier="8dec7e06-1106-c5b8" resourceid="" name="" cl
          <style identifier="16c1b44b-512f-016c" resourceid="" name="" cl
          <style identifier="5993695c-c2aa-6762" resourceid="" name="" cl
          <div identifier="7e8cb9fa-9446-8b07" resourceid="skip-to-main-c
            <div identifier="e40d9a95-3fa2-3dab" resourceid="" name="" cl
              <a identifier="82c7ffee-e569-8815" resourceid="skipToMainCo
            </div>
          </div>
          <div identifier="7ad24526-2b00-975e" resourceid="hamburger-pane
            <div identifier="6528beed-5c62-8e84" resourceid="" name="" cl
              <div identifier="8c5be981-159a-8b3c" resourceid="" name=""
                <ul identifier="640f2ddc-f86f-0f34" resourceid="" name=""
                  <li identifier="8faf500d-73a8-cd2c" resourceid="" name=
                    <a identifier="a3aa8dc2-5293-a217" resourceid="" name
                      <i identifier="c87c80b9-398d-a5c4" resourceid="" na
```

Last Semantic Processing 05/06/2020 05:15:15:701

```
Root
root
  ocr
  schemaorg
  html
  textcontent
  naming
    name "/book/details"
  opengraph
    title "OpenTable - Complete your reservation"
    type "website"
    site_name "OpenTable"
  error
  action
    inv-action(classname='button' ) "Previous month"
    inv-action(classname='button' ) "Next month"
    inv-action(classname='submit' ) "Find a Table"
    inv-action(classname='select' ) "party size"
    inv-action(classname='select' ) "time"
    inv-action(classname='select' ) "Choose Country Code"
    inv-action(classname='select' ) "Special Occasion"
```

Figure 4: Information associated with a UI snapshot, including screenshot, UI tree, and knowledge tree.

depending on the UI element type such that the presence/absence of uncommon UI elements (e.g., radio buttons, sliders, etc.) has a higher weight compared to that of common UI elements (e.g., buttons, text fields, etc.). If some features are not applicable to some snapshots (e.g., Schema.org annotations are often not present) the similarity score is 0. To account for that, *snapSim* is normalized over the number of applicable features $Na \leq N$.

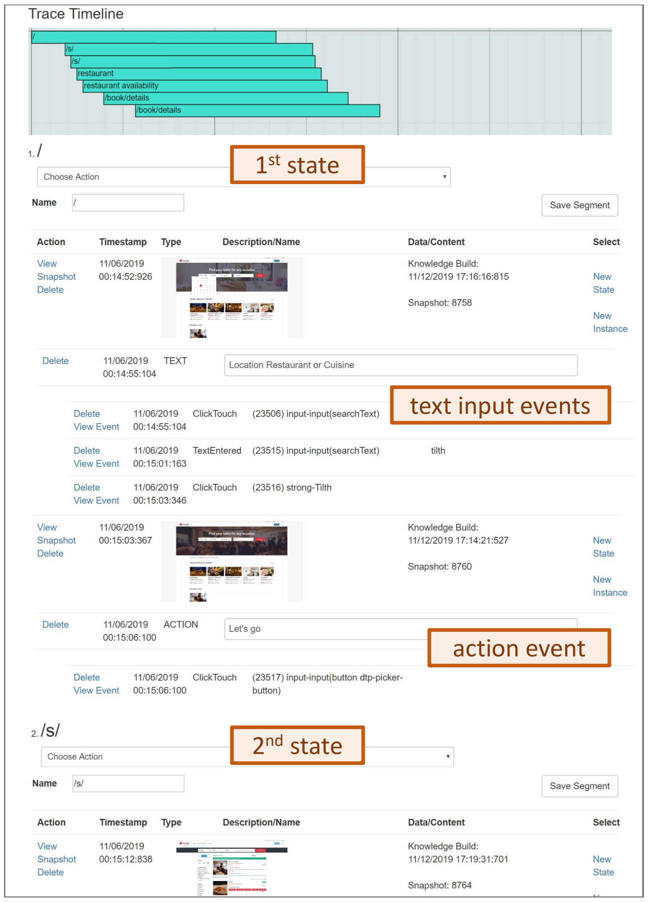
Empirically, we found how to weigh the different features (w_i weights). Deeplink parameters as well as Open Graph and Schema.org annotations are usually highly reliable similarity clues, but they are not always present. Matching texts is generally noisy because of dynamic content. Relying on the presence/absence of interactive UI elements can work well, especially if uncommon element types are present. These simple heuristics allowed us to outperform prior approaches (see §5.2) without the need to collect a labeled dataset for training. If developers are willing and able to provide more data, a screen embedding model [61] could be trained.

In summary, Etna receives snapshots from the recorder, computes their knowledge trees, and then clusters them based on their timestamps and pairwise similarity. If a subsequent snapshot has similarity higher than a set threshold (e.g., 0.6) with the previous one, it is considered as part of the same state, otherwise, a new state is created. The trace timeline is then segmented into ordered states. A portion of a timeline with two states is shown in Fig. 5.

UI event classification. After state classification, Etna processes UI events to identify the logical actions that connect the states. A logical action may correspond to multiple subsequent UI events. For example, the 9 UI events shown in Fig. 6 correspond to the logical action of "finding a table at a restaurant". Executing this action involves multiple sub-steps: the first two UI events correspond to selecting a reservation date (one event corresponds to opening the time menu and one to selecting an item), the third and forth UI events to selecting a time, and so forth. Etna's goal is to distinguish between *input-related events* that are responsible for setting input parameters associated with a logical action, and *action-related events* that are responsible for submitting or concluding the actual action. For the example in Fig. 6, Etna aims to infer that the ninth event is action related and the other 8 events are related to 4 different input parameters (date, time, party size and search term). Further, Etna needs to name action and inputs in a meaningful way (e.g., the input corresponding to the 7-8th events should *not* be named "Tilth" but something more general).

This classification process is challenging especially for two reasons. First, UI traces are inherently noisy, containing duplicate or unrelated UI events. To be a practical and general-purpose tool, Etna cannot rely on many task demonstrations or on application instrumentation to clear possible ambiguity. To deal with this challenge Etna leverages and extends prior work [38, 62] by discarding UI events that are associated with empty or invisible content or that are irrelevant to the current flow based on spatial locality. Like Ringer [7], Etna ignores high-frequency mouseover, mousemove and mouseout events, which can generate delays and are usually unnecessary to action replay. The second problem is specific to Etna. Etna's goal is to extract not only action definitions, but *executable* action functions, which means it cannot afford to miss any parameters. This is problematic because not only multiple UI events may correspond to a single input parameter but also a single UI event may correspond to multiple parameters. To cope with this Etna introduces the concept of multi-input events.

For each recorded UI event, first, Etna searches the timeline forward from the time of the event and associates it with the first state which contains the UI element corresponding to the event. Second, it uses the DOM tree structure and element attributes to group all UI events into candidate *input clusters* (e.g., event 1 and 2 in Fig. 6 belong to the "date" input cluster). Input clusters can be of one of three types: *(a) simple input interactions*, such as entering text in a form field or selecting an element from a menu, which can be easily recognized using DOM attributes; *(b) simple list selections*,



Trace Timeline

/
/s/
/s/
restaurant
restaurant availability
/book/details
/book/details

1. /

1st state

Choose Action

Name /

Save Segment

Action	Timestamp	Type	Description/Name	Data/Content	Select
View Snapshot Delete	11/06/2019 00:14:52:926			Knowledge Build: 11/12/2019 17:16:16:815 Snapshot: 8758	New State New Instance
Delete	11/06/2019 00:14:55:104	TEXT	Location Restaurant or Cuisine		
Delete View Event	11/06/2019 00:14:55:104	ClickTouch	(23506) input-input(searchText)		
Delete View Event	11/06/2019 00:15:01:163	TextEntered	(23515) input-input(searchText)	tilth	
Delete View Event	11/06/2019 00:15:03:346	ClickTouch	(23516) strong-Tilth		
View Snapshot Delete	11/06/2019 00:15:03:367			Knowledge Build: 11/12/2019 17:14:21:527 Snapshot: 8760	New State New Instance
Delete	11/06/2019 00:15:06:100	ACTION	Let's go		
Delete View Event	11/06/2019 00:15:06:100	ClickTouch	(23517) input-input(button dtp-picker-button)		

text input events

action event

2. /s/

2nd state

Choose Action

Name /s/

Save Segment

Action	Timestamp	Type	Description/Name	Data/Content	Select
View Snapshot Delete	11/06/2019 00:15:12:838			Knowledge Build: 11/12/2019 17:19:31:701 Snapshot: 8764	New State

Figure 5: Trace timeline for opentable.com showing a transition from the "/" state (home page) to the "/s/" state (search results page); in the demonstration the user typed "tilth" in the "Location Restaurant or Cuisine" search text field and clicked the "Let's go" button.

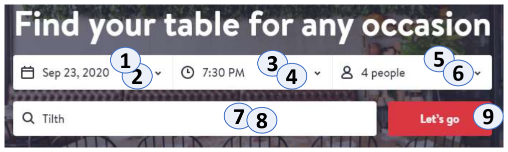


Figure 6: An interaction with the OpenTable search interface generates 9 UI events. Etna infers the search action "Let's go" with date, time, party size, and search term as inputs.

which require analyzing the DOM tree to identify similar subtrees to ascertain whether the click is on a list member or on a unique item (e.g., a submit button at the bottom of the list); and *(c) nested list selections*, which require analyzing each level of similar subtrees.

Dealing with list selections can be challenging. In modern webpages, it is common to find collection-like UI elements, such as grids or lists of items, where each item may recursively contain other collections. Fig. 7 shows an example where in the list of restaurant

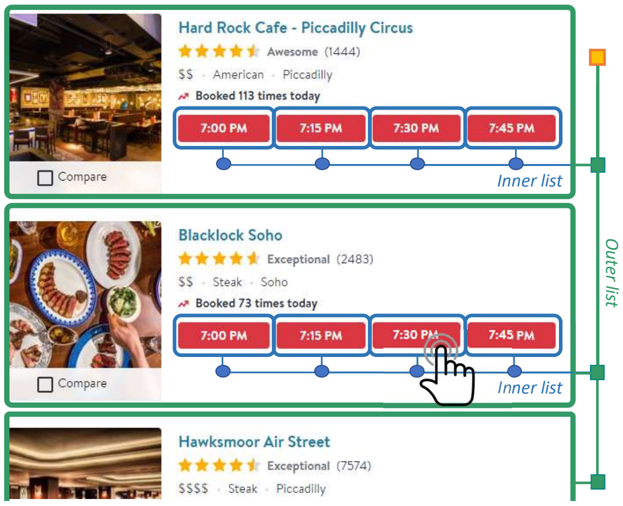


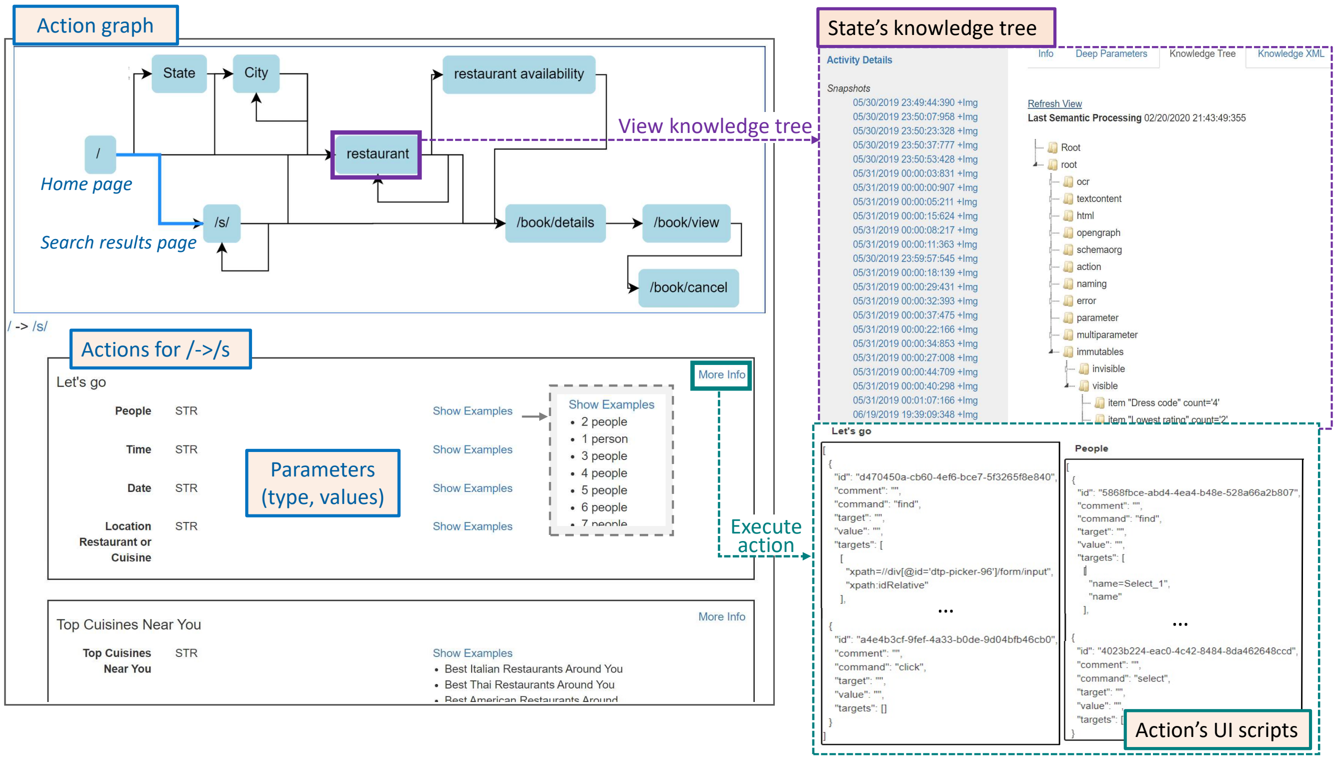
Figure 7: Example of multi-input event. The "7:30 PM" click event sets 2 inputs: booking time and restaurant name.

items (the outer list) each item includes the restaurant name, various other details, and the list of available reservation times (the inner list). When interacting with the restaurant list, a user may click on the name of a restaurant which triggers a transition to the restaurant's details page or they may click on the reservation time of a specific restaurant (as in Fig. 7), which causes the same page transition. However, the first case corresponds to a simple list selection where the user has specified one input (the restaurant name); the latter corresponds to a nested list selection where the user has specified *two* inputs: the restaurant name ("Blacklock Soho") and the reservation time ("7:30 PM"). When a *single* recorded UI event is responsible for multiple input selections we call it **multi-input event**.[1] Prior automation systems did not consider multi-input events. Even web data scraping tools [18, 90], which are programmed to iteratively scrape many content entries from a website, assume all entity attributes to scrape have been clicked in the user demonstration, hence this problem does not arise.

To handle multi-input events, given a collection-like UI element (detected using DOM attributes), Etna compares multiple levels of UI subtrees rooted at the collection node for similar content, location, and structure to infer the "maximal repeating UI pattern", defined as the longest sequence of UI elements that repeats identically, e.g. the pattern {restaurant-name, ratings, price-cuisine-location, booked-text, time-list} in Fig. 7. Based on the depth at which the UI event occurred, Etna generates one (the event occurred at the outer level) or more (the event occurred at an inner level such as the time-list) input slots. Likewise, at replay time, *all* input slot values are used to configure a single UI event to replay.

At the end of this process, some UI events will be grouped into input clusters and some others will remain isolated. Isolated events that are classified as inconsequential to the overall task (e.g., a touch event on the opposite side of the screen) are deleted. To recognize

[1]Note that even if instructed, in this scenario, users cannot produce an event for every input because they cannot click on the restaurant name or the reservation time *without* causing a page transition.



Let's go

People	STR	Show Examples
Time	STR	Show Examples
Date	STR	Show Examples
Location Restaurant or Cuisine	STR	Show Examples

Show Examples
- 2 people
- 1 person
- 3 people
- 4 people
- 5 people
- 6 people
- 7 people

Top Cuisines Near You

Top Cuisines Near You	STR	Show Examples

- Best Italian Restaurants Around You
- Best Thai Restaurants Around You
- Best American Restaurants Around

Activity Details

Snapshots

05/30/2019 23:49:44:390 +Img
05/30/2019 23:50:07:958 +Img
05/30/2019 23:50:23:328 +Img
05/30/2019 23:50:37:777 +Img
05/30/2019 23:50:53:428 +Img
05/31/2019 00:00:03:831 +Img
05/31/2019 00:00:00:907 +Img
05/31/2019 00:00:05:211 +Img
05/31/2019 00:00:15:624 +Img
05/31/2019 00:00:08:217 +Img
05/31/2019 00:00:11:363 +Img
05/30/2019 23:59:57:545 +Img
05/31/2019 00:00:18:139 +Img
05/31/2019 00:00:29:431 +Img
05/31/2019 00:00:32:393 +Img
05/31/2019 00:00:37:475 +Img
05/31/2019 00:00:22:166 +Img
05/31/2019 00:00:34:853 +Img
05/31/2019 00:00:27:008 +Img
05/31/2019 00:00:44:709 +Img
05/31/2019 00:00:40:298 +Img
05/31/2019 00:01:07:166 +Img
06/19/2019 19:39:09:348 +Img

Refresh View
Last Semantic Processing 02/20/2020 21:43:49:355

```
Root
root
  ocr
  textcontent
  html
  opengraph
  schemaorg
  action
  naming
  error
  parameter
  multiparameter
  immutables
    invisible
    visible
      item "Dress code" count='4'
      item "Lowest rating" count='2'
```

Let's go

```
[
 {
  "id": "d470450a-cb60-4ef6-bce7-5f3265f8e840",
  "comment": "",
  "command": "find",
  "target": "",
  "value": "",
  "targets": [
   [
    "xpath=//div[@id='dtp-picker-96']/form/input",
    "xpath:idRelative"
   ],
               ...
 {
  "id": "a4e4b3cf-9fef-4a33-b0de-9d04bfb46cb0",
  "comment": "",
  "command": "click",
  "target": "",
  "value": "",
  "targets": []
 }
]
```

People

```
[
 {
  "id": "5868fbce-abd4-4ea4-b48e-528a66a2b807",
  "comment": "",
  "command": "find",
  "target": "",
  "value": "",
  "targets": [
   [
    "name=Select_1",
    "name"
   ],
               ...
 {
  "id": "4023b224-eac0-4c42-8484-8da462648ccd",
  "comment": "",
  "command": "select",
  "target": "",
  "value": "",
  "targets": [
 }
```

Figure 8: The Etna graph editor showing the action graph for opentable.com. By selecting an edge, one can browse the associated actions. Each action has a name (e.g.,"Let's go") and input parameters. Each parameter has a name, a type and a set of valid values (in the "Show Examples" window). By selecting a vertex such as restaurant**, one can view the aggregated knowledge tree for this state (upper right) or the knowledge trees of individual snapshots (not shown). To execute an action (lower right), a user selects the action from the graph , obtains the associated UI script(s) (which can be edited), and executes them (upon starting the Etna executor service).**

action-related events, the data analysis works backward in time from the last event in every state. If the last event is an isolated event that has features that comply with an interactive UI element (e.g., button attribute), it is classified as action related. Otherwise, the last input cluster is labeled as action related. All remaining candidate input clusters are labeled as input related. Finally, actions and inputs are named after the names of the UI events associated with them (name, placeholder text, value, etc. of the corresponding UI elements). As our evaluation shows (§5.1), naming in Etna is not yet optimal, and could be improved by leveraging screen understanding techniques [19, 72, 97, 120], which use object detection models to locate and classify UI elements in a screen.

Trace editing. Ideally, at the end of this analysis, all recorded UI events should be either deleted or classified as action or input related. In practice, Etna may not always be able to classify all events with enough confidence because UI interactions may be noisy or lack important details (e.g., missing DOM attributes). In the Etna timeline interface (Fig. 5) we found that *detecting incorrectly-classified UI events* was generally harder than *manually classifying unassigned events*. Hence, in case of low confidence, Etna makes a conservative decision and leaves the UI event unassigned. Using the timeline editor, a user can drag & drop an unassigned UI event under the correct input/action or create a new cluster. In Appendix A (Fig. 13) we provide an example of this process. Other modifications that users can make using the timeline editor include: *(i)* creating a new state or cloning a state (using the "New state" and "New instance" buttons), *(ii)* removing a state, *(iii)* deleting an input or an action, and *(iv)* creating new inputs.

3.3 Step 3. Constructing the action-state model

Once traces have been processed to identify states, actions and inputs, generating the *action-state model* is straightforward. Etna represents the model as an *action graph* where vertices correspond to the application's states and edges to the transitioning actions. Fig. 8 shows the action graph generated for the restaurant reservation task in opentable.com. It consists of 9 states and 28 actions. The actions Let's go and Top Cuisines Near You go from the home state (/) to the search results state (/s/); this state has a self-loop action to filter a list of restaurants by cuisine, region, price, etc. From here, selecting a restaurant name takes to the restaurant state (restaurant), and then selecting a reservation time leads to filling a restaurant reservation form (/book/details vertex), and so forth.

Etna's action graphs are reminiscent of knowledge graphs [16, 43] and other task models [17, 62]. However, a main difference is

Table 1: Etna's action knowledge.

Info type	Content
Graph	States, Actions
Action	Name, Inputs, UI script, Parametrizable deeplink
Input	Name, Type, Input-values, UI script
UI script	Interaction type, Target element, Value, etc.
State	Name, State knowledge tree, UI snapshots
UI snapshot	DOM tree, Screenshot, Snapshot knowledge tree
Snapshot/state knowledge tree	URLs, Deeplink parameters, SEO meta-tags, Semantic annotations (Open Graph, Schema.org, HTML data-attrs), OCR-texts, Immutable texts*, Set of interactable UI elements, Error texts*

*Aggregated feature, present only in state knowledge trees.

that Etna's edges do not only represent static relations between states, but instead **parameterizable and executable actions** (execution is explained in §3.4). Each action is formalized as a function with input parameters (e.g., the action Let's go in Fig. 8). From the input-related UI events Etna extracts a schema for the input parameter consisting of name, type, and a set of possible values (if applicable). Parameter values can be texts entered during demonstration or strings scraped from selection-like elements (e.g., available reservation times scraped from a time selection menu).

Each state in a trace timeline will typically contain multiple snapshots, each with a knowledge tree associated. The individual knowledge trees are used to compute an **aggregated knowledge tree** for each state in the model (Fig. 8, upper right). Frequency counts are computed for all features, then aggregation-only features are added. Examples of these include Immutable texts capturing texts that recur in the same UI tree position in most snapshots associated with a state, and Error texts capturing error messages that one may encounter in this state (inferred using keyword matching and font/color properties). Overall, the goal of aggregated knowledge trees is to capture what states are about and which functionality they support, which is useful for state validation (see the RPA scenario in §4.3). Knowledge trees could also be connected to external knowledge bases [16, 41, 73, 106] for richer semantic annotations. Table 1 summarizes the knowledge stored in action graphs, which can be accessed through the graph editor app, queried using a Gremlin API [83], or exported to file system.

As mentioned earlier, instead of recording one long interaction trace, Etna allows users to record multiple short traces and merge their action graphs into one graph. The merge works by first identifying equivalent states across multiple graphs and then merging their actions, inputs, and knowledge trees. To compute state equivalence we introduce a **state similarity** function. This is analogous to the snapshot similarity function of Eq. 1, but, instead of comparing snapshots using features computed based on each snapshot's knowledge tree, it compares states using features computed based on each state's aggregated knowledge tree.

To conclude, Fig. 9 shows three action graphs obtained by merging multiple interaction traces, as reported in the accompanying table. The number of states in a graph varies from 7 to 28, and the number of actions from 25 to 34. In general, the size of a graph varies with the complexity of the website and the modeled tasks.

3.4 Step 4. Executing parametrized actions

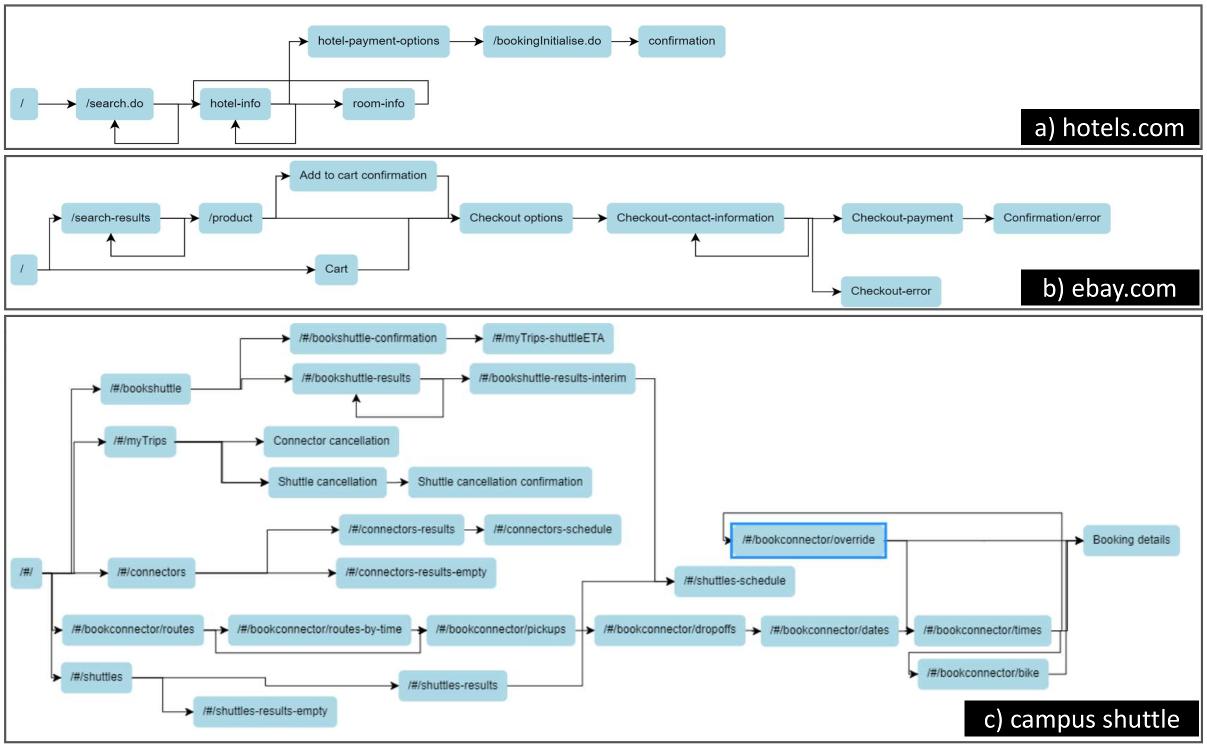
Action graphs capture not only static action knowledge, but also dynamic knowledge by supporting action execution. For execution, Etna automatically extracts **parameterizable deeplinks** and generates **UI scripts** for all actions, and provides an executor service.

Traditional UI automation tools [18, 57, 99] rely on UI scripts for action execution. Etna introduces parameterized deeplinks for action replay. It computes them by comparing path and query string of the captured snapshot URLs to extract a parameter set along with captured values and inferred types. For example, from URLs of various restaurant profile pages using the scheme www.opentable.com/r/[str]?corrId=[str], it infers a deeplink taking a string (extracted from the URL path) and corrId (extracted from the URL query string and whose presence is immaterial to get the correct result). Deeplinks can guarantee a reliable and fast execution of actions, but they may not always exist (many websites hide parameters from their URLs) or may be hard to reverse engineer without a large collection of traces. For example, the search action shown in Fig. 6 which leads to the search results page in opentable.com takes 4 parameters as input; however, the deeplink to the same search results page has 11 parameters including app/session-specific ones such as metroId, latitude, longitude, regionIds, pageType, etc.

Etna complements deeplinks with UI scripts. UI scripts (Fig. 8, lower right) are less reliable but can always be extracted. A UI script specifies all UI interactions necessary to set an action's inputs (if any) and perform the final submit-like interaction (e.g., filling in some data fields and then clicking a "Go" button). Each replay interaction is specified using a set of pre-compiled replay commands (find, child, click, select, ...), similar to Selenium commands [99] but custom to Etna. This makes human-editing of the UI scripts easier.

With this setup Etna's **executor service** executes actions in two possible ways. If the action supports a *deeplink*, then it parametrizes and opens it in the browser. Otherwise, it falls back to *action replay*: it loads the URL of the webpage corresponding to the start state of the action and replays all UI interactions listed in the UI script.

Replaying an interaction involves locating the most probable UI element for the interaction and executing it. Since identifying an element in the UI tree relies on specific attributes, a change to those attributes may result in a failure to locate it. These changes may occur due updates or variability in the website UI caused by different platforms (web browsers), screen sizes (reactive layout) or A/B tests (different users see customized UIs). In general, a single UI element is unlikely to match all the element attributes observed at recording time also because there will be differences between the parameter values specified at execution time and at recording time. Fully solving this problem would require collecting traces from a large set of application environments and conditions, dramatically increasing the demonstration overhead. As in related work [7], Etna locates UI elements using a wide set of both *specific* (identifier, classname, etc.) and *broad* (type, color, etc.) element attributes, so to withstand small changes to individual attributes. Yet, an element's attributes may not be enough for actions that involve selecting one of many similar elements (e.g., a list of business names that share the same text header but have different addresses as sub-field). Hence, Etna also uses the *surrounding context* consisting of text and attributes of nearby UI elements.



	Traces	Snapshots	States	Actions	Input Parameters (Input values)	Size (MB)
hotels.com graph	5	126	7	25	50 (466)	128
ebay.com graph	8	133	10	29	60 (616)	134
campus shuttle graph	16	141	28	34	49 (160)	60

Figure 9: Three action graphs built by merging multiple traces. The table reports the number of traces and UI snapshots, the number of model's states, actions, input parameters/values, and the graph's exported size on disc (excluding snapshot images).

A key novelty in Etna is to use all these attributes to identify the best-guess element through a **semantic ranking** approach. The idea is to *search* for matching UI elements by progressively limiting the search context using element attributes, but search for *text strings* (e.g., a restaurant name) rather than fixed attributes (e.g., a class name). The find command, specified in the UI script, takes a list of *attribute-based filters* and runs them in order against the DOM tree (Fig. 8 shows two examples of filters based on xpath and name attributes). If a filter matches a unique element, that element is selected. If a filter matches multiple elements, subsequent filters in the list are run to attempt to reduce the matching set. This approach allows Etna to use modular filters. An example where this approach is essential is selecting items from dynamic lists. A filter is used to locate the sub-tree that contains the list results (e.g., hotels list), another to locate a sub-tree within the list which contains the desired text (e.g., hotel name), then another to locate the button within the item itself (e.g., room type). A search-based approach like this adds lots of flexibility by not pre-determining where content is within the DOM tree, and makes action replay more robust than with approaches that use fixed selectors (like Selenium, §5.3). Text search is currently based on exact match; it could be improved by supporting fuzzy matching.

4 USE CASES

Etna can work out-of-the-box with existing websites. While it primarily targets developers of UI automation experiences such as robotic process automation (RPA) (§4.3) and AI assistants (§4.2, §4.4), it can also be used for data collection, to generate data for NLP research (§4.4) or to scrape data from websites (§4.1). In the following we give an overview of four use cases that fall into these categories. Fig. 10 summarizes them and highlights which aspects of Etna they leverage. Prior PBD tools for web automation, which focus on generating UI scripts for action replay, would have been able to support only the first use case on data access APIs.

4.1 Data access APIs

We used Etna to generate APIs to access web data for which official APIs either did not exist or did not meet the data formats we needed. A *path* in an action graph, which can be executed through one or multiple UI scripts, can be considered an *API request*. Etna's executor service did not provide an *API response* so we extended it to do so.

We built a simple interface where users can specify API response schemes listing which resources (texts or images) to fetch. Resources are identified using a regex expression or an xpath, and can

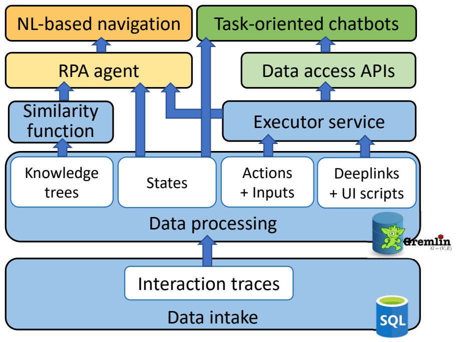
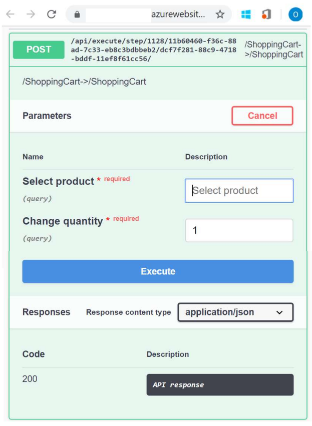


Figure 10: Etna's platform and four use cases built on it.



```
/api/execute/step/1128/11b60460-f36c-88
ad-7c33-eb8c3bdbbeb2/dcf7f281-88c9-4718
-bddf-11ef8f61cc56/
```

/ShoppingCart->/ShoppingCart

/ShoppingCart->/ShoppingCart

Parameters | Cancel

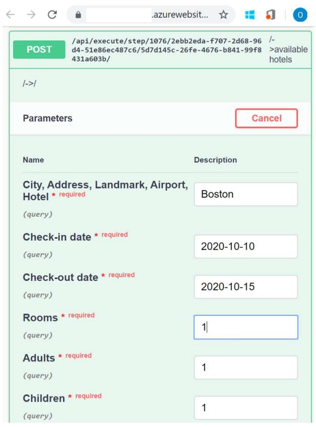
Name	Description
Select product * required (query)	Select product
Change quantity * required (query)	1

Execute

Responses | Response content type | application/json

Code	Description
200	API response

(a) rei.com update cart API.



```
/api/execute/step/1076/2ebb2eda-f707-2d68-96
d4-51e86ec487c6/5d7d145c-26fe-4676-b841-99f8
431a603b/
```

/->available hotels

/->/

Parameters | Cancel

Name	Description
City, Address, Landmark, Airport, Hotel * required (query)	Boston
Check-in date * required (query)	2020-10-10
Check-out date * required (query)	2020-10-15
Rooms * required (query)	1
Adults * required (query)	1
Children * required (query)	1

(b) hyatt.com hotel search API.

Figure 11: Swagger APIs constructed using Etna.

have type "collection" (to scrape sets of items matching a condition, such as a list) or "single" (to scrape individual items). Then, we extended Etna's executor service to launch an API, wait for the resources to be visible, and fetch them. To make our APIs easy to use, we extended Etna's web app to support exporting paths selected in an action graph in the Swagger API format [110]. Fig. 11 shows two APIs generated in this way (the response scheme is not shown). After parametrizing the API, the user can launch its execution directly from the web interface (or invoke it from a script with conditions, loops, etc.).

4.2 Zero-coding task-oriented chatbots

We used Etna's data access APIs not only for general data scraping but also for supporting task-oriented chatbots. Task-oriented chatbots aim to understand a user request in natural language (NL) and execute the associated task. They usually consist of three modules: 1) a control structure, typically hand-designed by developers, which represents the actions (intents) the system supports as well as the inputs (slots) required by the actions for execution; 2) a conversational interface that maps NL requests to intents and slots, and prompts users for missing or incorrect slot values; and 3) an executor that uses intents and slot values to parametrize APIs and executes them. We leveraged Etna to build a framework and a process through which developers can generate chatbots without writing any code and without requiring access to third-party APIs.

As others have observed [62], action graphs implicitly represent the control structure of a dialogue system: they embed entire task flows including states, action dependencies, and parameter schemes. Hence, with Etna, to build the bot's control structure (module 1) and executor (module 3), one can enumerate all one-hop paths in an action graph and wrap each one in an API, as previously described.

To build the conversational interface (module 2), each path must be associated with an intent, and a language understanding model must be trained to map NL queries to every intent. In our framework, we ask developers to write 30–50 utterances for each intent and use luis.ai for training. Then, for each input parameter in the action graph, they need to provide a question clause, and we generate a validator of user answers based on the parameter's schema provided by Etna. Alternatively, as in prior work [62], with an adequate conversational dataset, a seq2seq model could be trained, thus lowering the developer overhead.

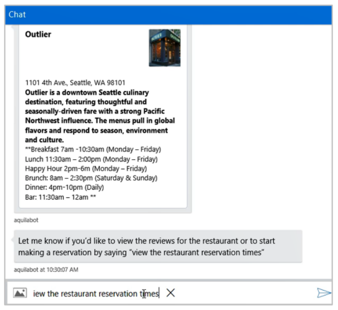
To glue the tree modules together, we built an application-independent runtime on top of the Microsoft Bot Framework SDK [82]. The SDK provides dialogue templates for user conversation, offers primitives to maintain a conversation stack, and handles network connections. Our runtime uses the SDK primitives to connect conversational interfaces to Etna's APIs and executor service, and to build a dialogue interface to converse with the user.

Using this Etna-based platform, we enabled code-free generation of chatbots: by providing an action graph and a conversational interface, developers obtain an executable chatbot. Fig. 12 shows screenshots of four chatbots we built through this process: searching and booking restaurants using OpenTable, searching movies and adding them to a watchlist using IMDB, checking the weather forecast in Foreca, and searching and buying books with Barnes&Nobles.

4.3 Model-driven RPA

In recent years, UI automation has gained remarkable attention thanks to RPA [111], to automate repetitive and error-prone tasks in business applications, such as invoice processing, customer registration, and payroll operations [51]. For RPA, reliability and correctness of task execution are paramount. The current best practice to verify task completion is to specify input/output validation rules, but ultimately many rely on manual inspection. Using Etna we implemented a more reliable approach called "model-driven RPA".

Traditional PBD tools [99] used for RPA have no notion of application state. The assumption is that a task completes correctly if all of its actions are executed. In practice, this is not true. For example, entering all data in a form and clicking a "Submit" button is not an indication of success; a form validation error or a similar message may be displayed indicating that the task is indeed *not* complete. Even an advanced tool like Robofox [50], which generates "assertions" to verify whether HTML elements/data that were available during recording are available during replay, would fail in this scenario. Verifying task completion by inspecting the final application state is not sufficient either because multiple action sequences may



Outlier

1101 4th Ave., Seattle, WA 98101
Outlier is a downtown Seattle culinary destination, featuring thoughtful and seasonally-driven fare with a strong Pacific Northwest influence. The menus pull in global flavors and respond to season, environment and culture.
**Breakfast 7am -10:30am (Monday – Friday)
Lunch 11:30am – 2:00pm (Monday – Friday)
Happy Hour 2pm-6m (Monday – Friday)
Brunch: 8am – 2:30pm (Saturday & Sunday)
Dinner: 4pm-10pm (Daily)
Bar: 11:30am – 12am **

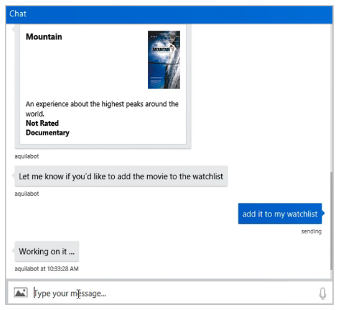
aquilabot

Let me know if you'd like to view the reviews for the restaurant or to start making a reservation by saying "view the restaurant reservation times"

aquilabot at 10:30:07 AM

iew the restaurant reservation times

(a) OpenTable restaurant bot



Mountain

An experience about the highest peaks around the world.
Not Rated
Documentary

aquilabot

Let me know if you'd like to add the movie to the watchlist

aquilabot

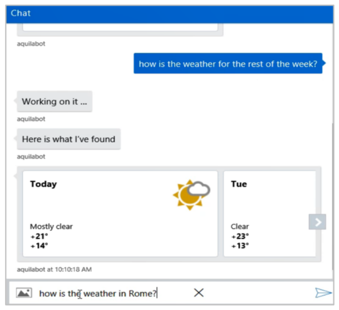
add it to my watchlist

sending

Working on it ...

aquilabot at 10:33:28 AM

(b) IMDB movie bot



aquilabot

how is the weather for the rest of the week?

Working on it ...

aquilabot

Here is what I've found

aquilabot

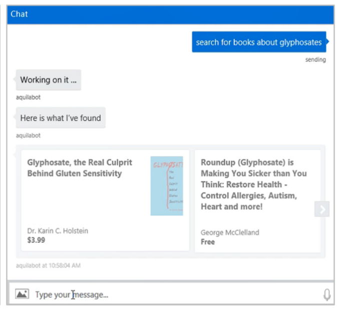
Today
Mostly clear
+21°
+14°

Tue
Clear
+23°
+13°

aquilabot at 10:10:18 AM

how is the weather in Rome?

(c) Foreca weather bot



search for books about glyphosates

sending

Working on it ...

aquilabot

Here is what I've found

aquilabot

Glyphosate, the Real Culprit Behind Gluten Sensitivity
Dr. Karin C. Holstein
$3.99

Roundup (Glyphosate) is Making You Sicker than You Think: Restore Health - Control Allergies, Autism, Heart and more!
George McClelland
Free

aquilabot at 10:58:04 AM

(d) Barnes&Noble book bot

Figure 12: Four chatbots built using Etna's action graphs.

lead to the same final application state. To reliably verify task completion, *every* significant state change which occurred during task replay must be validated.

To this end we leveraged Etna's modeling of application states. We built an **RPA executor agent** that uses action graphs to validate its actions and recognize errors or wrong paths. Inspired by partially observable Markov decision processes (POMDPs), after executing an action, the agent captures an *observation* of the current UI (i.e., a UI snapshot) and compares it with the model's states using Etna's **snapshot-state similarity** function.[2] The state most-similar to the observation represents the agent's *belief* state. The "expected" state is the state the executed action should lead to according to the action graph model. If the belief state deviates from the expected state, the agent recognizes one of the following conditions: *(i) lack of progression*, where the state remained the same whereas the model expected the state to change (e.g., clicking on a button has no effect on the UI); *(ii) branching to a known error or another state*, where the agent transitioned to a known state but not the expected one; or *(iii) unknown state*, where the observation is not similar to any known state so the agent cannot proceed. In all these cases the agent stops, and either logs the error or prompts the user for help.

This model-driven approach provides stronger guarantees on the correctness of a task execution, and documents every step of it thus producing a useful log. In §5.2, we evaluate how well Etna's snapshot-state similarity function can support this scenario.

4.4 NL-guided web navigation

More recently, we used Etna to build a web navigation experience similar to that of Google's Duplex on Web assistant [112]. The user provides a command in natural language with reference to a specific website. The system maps it to a task path in the action graph of that website, and then executes it by driving its UI. Unlike the previously-described chatbots, in this experience, the output returned to the user is the webpage the system navigates to.

To execute task paths we leveraged the RPA agent just described. To train a semantic parser model mapping user commands to task paths, we used Etna to collect a training dataset.[3] We proceeded as follows. First, we enumerated all task paths present in the target action graph in the format <start_state, action_name, input_parameters>. Then, for each triplet we wrote multiple command templates with the input parameters as placeholders, e.g., "*Book me the <room_type> room*". Recall that for list and menu-like elements, Etna collects the set of possible values (e.g., {"1 king bed", "2 king beds", "2 king beds with view", ...} for the "room_type" parameter). We used those values and paraphrases of them (e.g., "single king bed") to instantiate a large number of commands. We applied this approach to 9 websites in the restaurants, hotels and shopping domains, and generated more than 53k command and navigation path pairs. This dataset allowed us to train various semantic parsing models with the aim to understand how to build generalizable models (e.g., training a semantic parser using data from one restaurant website and applying it to different restaurant websites). Further details on our approach and performance results are reported elsewhere [77]. Overall, Etna's key role was to facilitate the creation of a novel dataset by providing definitions of task flows, actions, and input schemes.

5 EVALUATION

Over the past four years we have used Etna to capture over 1,200 traces from 50+ different websites from various categories (shopping, restaurants, hotels, transportation, etc.). We collected traces initially for testing and performance purposes, but later to support the use cases previously described. Etna is a general-purpose tool which can work with most websites. It does not work on websites that block the ChromeDriver and does not capture certain UI interactions (see §6 for details). We were particularly interested in using Etna to model transactional tasks such as reservation, booking or purchasing tasks rather than information retrieval tasks such as weather forecasts or news. Transactional tasks tend to be more complex and consist of longer sequences of actions with interesting dependencies (e.g., filtering, rebooking, editing, canceling, etc.). Because of the popularity of search interfaces in these websites, Etna is optimized to automate multi-level lists, such as those used to display search results. Etna is not suited to represent and automate webpages containing tables, maps or calendars. Etna cannot replay interactions requiring precise timing between subsequent events, such as in games or online trading. We have used Etna both with popular websites such as ebay.com, opentable.com or uber.com and less popular ones, such as our on-campus shuttle booking service.

[2]This function is equivalent to the similarity function of Eq. 1, but instead of using the knowledge trees of individual snapshots, it uses the knowledge tree of the observation's snapshot and the aggregate knowledge tree of each state.
[3]The dataset and code are available at https://github.com/microsoft/flin-nl2web.

Table 2: Effort to manually edit 20 demonstration traces. Changes are classified as create (C), delete (D), rename (R), and move (M) operations. The table also reports the size of each trace as number of UI snapshots, UI events, states, inputs, and actions.

Website	Trace description	Num	Num	Num	Num	Num	State (%)			Inputs (%)			Actions (%)			Events (%)	
		snapshots	events	states	inputs	actions	C	D	R	C	D	R	C	D	R	D	M
opentable.com	Search restaurants & filter	26	26	9	8	8	88.9	0.0	0.0	0.0	0.0	75.0	0.0	0.0	100.0	3.8	11.5
	View restaurant times	22	15	7	5	4	14.3	0.0	14.3	0.0	0.0	100.0	0.0	0.0	50.0	6.7	26.7
	Make a restaurant reservation	31	28	8	10	7	25.0	0.0	25.0	0.0	0.0	60.0	0.0	0.0	42.9	3.6	14.3
rei.com	Add to cart & edit cart	13	9	5	4	4	20.0	0.0	40.0	25.0	0.0	75.0	0.0	0.0	75.0	0.0	0.0
	Checkout	23	21	6	7	5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	4.8	28.6
	Pay	10	9	2	4	1	50.0	0.0	50.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	22.2
hyatt.com	Search hotels	17	17	3	8	2	0.0	0.0	66.7	12.5	0.0	37.5	0.0	0.0	50.0	5.9	35.3
	View hotel info & features	32	21	9	7	8	0.0	10.0	88.9	14.3	12.5	85.7	0.0	0.0	25.0	4.8	19.0
	Search hotels & book	52	55	8	22	7	25.0	0.0	50.0	4.5	4.3	36.4	0.0	0.0	42.9	5.5	23.6
campus shuttle	Book on-demand shuttle	12	11	4	3	3	25.0	0.0	0.0	0.0	25.0	33.3	0.0	0.0	33.3	9.1	27.3
	Search shuttle's schedule	7	5	3	1	2	33.3	0.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	0.0	40.0
	Book fixed-route shuttle	11	9	4	3	3	50.0	0.0	25.0	0.0	0.0	33.3	0.0	0.0	33.3	0.0	22.2
macys.com	Search shirts & filter	21	18	8	1	7	0.0	0.0	87.5	0.0	80.0	0.0	0.0	0.0	42.9	11.1	33.3
	View cart	7	2	3	0	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0
radissonhotels.com	Search hotel & book it	28	29	5	15	4	20.0	0.0	0.0	26.7	6.3	6.7	0.0	0.0	50.0	3.4	34.5
	Search hotels & filter	15	9	4	3	2	0.0	20.0	0.0	33.3	25.0	33.3	0.0	0.0	100.0	0.0	33.3
accorhotels.com	Search & filter hotels	31	26	11	8	10	45.5	8.3	18.2	0.0	0.0	62.5	0.0	0.0	40.0	3.8	11.5
yelp.com	Filter restaurant results	16	9	4	3	3	75.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	11.1
bookatable.co.uk	Search restaurants	4	3	2	1	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	33.3
	Book offer at a restaurant	8	5	3	2	2	33.3	0.0	33.3	0.0	0.0	50.0	0.0	0.0	100.0	0.0	20.0
	Median	16.5	13.0	4.5	4.0	3.5	22.5	0.0	16.2	0.0	0.0	36.9	0.0	0.0	46.4	3.5	22.9
	25th percentile	10.8	9.0	3.0	2.8	2.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	33.3	0.0	13.6
	75th percentile	25.5	21.0	7.5	7.5	6.0	33.3	0.0	45.0	8.5	9.4	61.3	0.0	0.0	87.5	5.1	33.3

After describing how we have leveraged Etna in various application areas, in this section, we provide more insights on its performance. A first goal of this evaluation is to give an intuition on the effort involved in creating action graphs. We do so by describing our experience in collecting action graphs for our use cases. While we do not claim our experience to be representative of a large population of developers, it was driven by real use cases (rather than the evaluation of the tool itself) so we consider it a valuable exploration of the expressivity of Etna. A second goal of this evaluation is to quantify the performance of two of Etna's core functionality: *(ii)* how accurate Etna's approach to snapshot-state similarity is, and *(i)* how reliably it can execute extracted actions.

5.1 Effort in constructing action graphs

To quantify the effort involved in turning demonstrations into useful action graphs, we selected 9 websites and randomly picked 3 traces for each. We discarded trace that overlapped significantly in functionality, thus obtaining a total of 20. We compared each trace before and after being edited. All except 3 traces (on-campus shuttle traces) were recorded and edited to support the NLP use case (§4.4). This choice ensured we evaluated unbiased traces.

We classified the observed trace modifications as related to four entities: *(i)* states, *(ii)* actions, *(iii)* inputs, or *(iv)* UI events. For each entity we considered all the operations that a user can perform using the trace editor. A user can delete (D), create (C) or rename (R) a state, an action or an input. A user can delete (D) or move (M) UI events. Appendix A provides examples of move event (Fig. 13) and rename/create state (Fig. 14) operations.

Table 2 reports the editing effort according to this taxonomy. For each trace and <entity,operation> pair, it reports the percentage of entities that were modified. For instance, in the first row, the number in the last column "Events-M" signifies that out of all 26 UI events contained in the OpenTable "search restaurant & filter" trace, 11.5% were manually moved.

In terms of input-related and action-related changes, we observe that most were renaming operations – input creations/deletions had a 75th percentile of only 9% and action creations/deletions were totally absent. This implies that Etna's UI event processing approach is effective at grouping UI events in input and action clusters, and at automatically removing irrelevant UI events (i.e., we see few UI event deletions). On the other hand, on average, in 23% of the cases ("Event-M" column), users had to manually assign unclassified UI events to an existing input/action cluster. As discussed earlier, Etna takes a conservative approach in case of low confidence by deferring event assignment to the user.

In terms of state-related changes, state deletions occurred in only 3 traces ("State-D" column). In 13 out of 20 traces, users decided to split the trace to add a new state ("State-C" column). Creating a state is usually dictated by a personal preference on how to model the logical task rather than a processing error. For example, in Fig. 14 (Appendix A), the user decided to split the restaurant state and classify the snapshot with the dialog box as a separate "restaurant availability" state. Another user may have chosen to treat these as a single state with a self-loop action called "show next available".

Overall, we conclude that the editing effort was affordable. Most of it dealt with renaming entities ("R" columns). These changes are up to the taste of the user and dependent on the target use case. For example, if the action knowledge is intended to be used for an NLP research project then action and input labels are more relevant than if it is intended to be used for end-to-end testing of websites. On the other hand, we acknowledge that there are various optimizations that Etna could employ to make labeling more precise, such as using computer vision models to associate text labels with nearby UI controls [72] or invoking an entity recognition service [105].

Table 3: Success rates for 6 variations of the snapshot-state similarity function. *Etna-all* is our proposal.

Website	# Model traces	# Test traces (snapshots)	# Snap-model comparisons	Etna-all	Only PageIds	Only InteractEl	PageIds+ InteractEl	PageIds+ SemAnnot	PageIds+ Texts
opentable.com	7	2(23)	161	**95.7**	65.2	4.4	65.2	82.6	**95.7**
ferry reservation	1	1(31)	496	**74.2**	0.0	45.2	**74.2**	0.0	0.0
campus shuttle	16	3(24)	311	**87.5**	8.3	41.7	54.2	8.3	33.3
bookatable.co.uk	8	4(42)	294	**88.1**	38.1	31.0	59.5	54.8	50.0
hotels.com	7	2(62)	372	**83.9**	0.0	40.3	46.8	0.0	74.2
rei.com	10	1(7)	77	**85.7**	0.0	**85.7**	**85.7**	**85.7**	71.4
reserve.com	3	1(5)	20	**60.0**	0.0	40.0	**60.0**	0.0	0.0
yelp.com	8	1(8)	56	**75.0**	62.5	12.5	**75.0**	62.5	**75.0**
Total/Average	60	15(202)	1787	**84.2**	18.8	35.7	59.4	27.2	53.5

5.2 Accuracy of snapshot/state classification

Etna provides three similarity functions for snapshot-to-snapshot comparisons (used to cluster UI snapshots in unique states, §3.2), state-to-state comparisons (used to merge multiple action graphs, §3.3), and snapshot-to-state comparisons (used for reliable action execution, §4.3). All three functions compute similarity using the same features and weights, but use features stored either in single or in aggregated knowledge trees, depending on the usage. A failure in computing similarity can cause false positives/negatives in state identification or lead to a wrong belief about an executor agent's current state and interrupt its execution. We evaluated our similarity approach using the snapshot-state similarity function which compares knowledge trees of single UI snapshots against aggregated knowledge trees of states. Similar observations hold for the other 2 functions.

We selected 75 traces across 8 websites. We used 60 traces to build each website's model (including aggregated knowledge trees for all states in the model), and the remaining 15 to create a set of test snapshots. (The number of traces used for each website varies depending on the length of each trace, see Table 3.) We manually ensured the test traces did not overlap with those used for model building (e.g., different dates, regions, search terms, etc.) and removed snapshot duplicates. Overall, we obtained 202 test snapshots. For each website we compared its test snapshots with all states in its model. The state with the highest similarity score and above a threshold of 0.4 was considered the predicted state. If no state could be identified with enough confidence, it was considered a failure. In total, we performed 1,787 snapshot-state comparisons.

Overall, we tested 6 snapshot-state similarity functions, including 2 baselines replicating state-of-the-art approaches. With reference to the knowledge tree features listed in Table 1 (last row), we tested: (1) *Etna-all*, Etna's similarity function that uses all knowledge tree features, (2) *OnlyPageIds*, a baseline that determines similarity using page identifiers and HTML title annotations, representing state-of-the-art systems such as Kite [62], *(3) OnlyInteractEl*, a baseline that compares the set of interactable UI elements appearing in the snapshot and in the state's knowledge trees, representing state-of-the-art systems such as SwiftHand [21], *(4) PageIds+InteractEl*, a variant of Etna's approach which combines the features used by *OnlyPageIds* and *OnlyInteractEl*, (5) *PageIds+SemAnnot*, a variant of Etna's approach with extends *PageIds* with semantic annotations (e.g., Schema.org), and (6) *PageIds+Text*, a variant of Etna's approach which extends *PageIds* with textual features such as overlapping texts and ImmutableTexts. For all similarity functions we used the same weights: $w_{PageIds}$=2, $w_{Annotations}$=5, $w_{InteractEl}$=1 and w_{text}=2. We measured success rate, defined as the percentage of cases in which given a test snapshot the state most-similar to it was identified (Table 3).

Overall, *Etna-all* achieved a success rate of 84.2%, largely outperforming the *PageIds* and *OnlyInteractEl* baselines. The ablation analysis also demonstrated how different features are most critical to different websites. For example, OpenTable is a website that makes use of semantic annotations, hence page identifier features and semantic annotations are sufficient to achieve a high success rate (82.6%). The same is not true for the ferry reservation website where the use of the InteractEl feature is essential. We have shown before screenshots of this website (Fig. 3b), where states may differ because of one or two different UI controls. We also point out the poor performance of *PageIds+Text*. Despite these websites containing lots of texts, relying only on text similarity can be noisy, unless a large number of traces relative to the number of states exist (e.g., Yelp). Finally, there are complex websites (e.g., campus shuttle) with large action graphs (see Fig. 9) for which only the combination of all features can succeed.

Main causes of failures for Etna-all included: *(i)* misleading naming of interactable UI elements (e.g., date and time widgets named as "Tuesday" and "7:00pm", respectively, resulted in no matches with snapshots collected on different dates and times), *(ii)* limited data affecting aggregated features such as Immutable texts, *(iii)* misleading page titles (e.g., a search page and a booking page with the same HTML title), and *(iv)* missing semantic annotations.

5.3 Reliability in executing actions

We briefly report on our experience with action replay. As in prior work [18], we compare Etna's approach to Selenium IDE [100], currently one of the most popular UI automation tools. From our dataset we selected 12 traces for 8 popular websites including opentable.com, hotels.com and macys.com. We explicitly picked traces that were 6–12 months old to test Etna with worst case scenarios. We parametrized each action's UI script with parameter values different from those observed at record time and executed them using Etna's executor service. To test Selenium IDE, we recorded the same tasks in the same websites using the Selenium recorder, parametrized with new values anything that in the generated UI scripts was auto-labeled as "value" (e.g., input texts) or cases where the exact input string was part of the selector (e.g., a restaurant

Table 4: Success rates of Etna and Selenium IDE based on 94 action executions.

Interaction type	Etna	Selenium IDE
Text input	100.0	100.0
Button/hyperlink click	92.3	92.3
Menu/checkbox select	86.7	80.0
1-level collection select	83.9	9.7
2-level collection select	100.0	0.0
Date pickers	40.0	0.0
Increment/decrement widgets	0.0	0.0
Average	80.9	41.5

name), and replayed them. In case of failure, we replayed the Selenium trace up to 3 times before classifying it as failed. With this setup, during execution, we observed changes in *i)* website content (e.g., list items) due to divergent search inputs and time of day; *ii)* location of UI elements, e.g., a recorded trace used "filter by color" but at replay time we chose "filter by brand" which was located elsewhere; *iii)* sets of UI controls, e.g., a "takeout" button added for COVID-19; and *iv)* UI trees (e.g., element ids).

Table 4 reports the success rates based on a total of 94 action/input interactions. Rather than reporting the results on a per-website basis, we grouped the interactions based on their types. We distinguish between selecting an item in a 1-level collection (outer list) or a 2-level collection (inner list), as defined in §3.2.

Overall, Etna was effective at handling UI changes and input variability, and achieved an average success rate of 80.9%. With both systems, text inputs, button clicks and menu/checkbox selections worked relatively well. However, while Selenium could deal with form filling and single buttons, it completely failed with list selections. It was not able to parse the items in a collection, as it would almost always use a numerical index to pick an item from a list without any relation to its content. In contrast, thanks to semantic ranking (§3.4), Etna was able to replay correctly most list selections – 1-level collection failures were mainly due to timing issues in loading the collection items. Other interactions failed due to missing UI events or failures in locating the UI elements. In general, replaying even the same recorded trace proved unreliable with Selenium due to lots of noisy events (mouse over, scroll, etc.) and due to timing constraints. Etna is optimized to filter out irrelevant UI events and to search and wait for target UI elements to appear.

The most challenging UI elements were date pickers and increment/decrement widgets. Failures with date pickers were expected. Date pickers are hard because of their many implementations and because the apparently simple task of setting a date actually consists of multiple steps (scrolling across months, picking a month and picking a day). Etna currently supports selecting a day from the current month and setting a date in date pickers where dates can be typed (as text inputs). Increment/decrement widgets are not difficult to replay as they are equivalent to button clicks, but they require mapping the specified values to the correct number of clicks to replay, which we do not support yet. Etna does not support also sliding bars, counters, and other customized widgets.

6 LIMITATIONS

We discuss current limitations and future work for Etna.

Computer vision and ML. Etna relies on DOM tree analysis to build application models; it uses OCR and DOM attributes to detect visible content. Ongoing work is exploring how to train deep neural models such as Faster R-CNN [93] and Yolo [92] for screen understanding. A hybrid structural-visual analysis of the webpage can help with state classification, locating UI elements for action replay, and labeling actions/inputs. As we collect more data, it should also be possible to train specialized ML models for detecting pop-up dialogs, error messages, etc.

Action mining. AI techniques could be used to expand the set of inferred actions beyond what appears in user demonstrations. For instance, in a real estate website, a user may demonstrate searching houses by specifying the number of bedrooms and the price range; additional inputs (number of bathrooms, year of construction) could be inferred automatically based on the UI layout. It is also possible to apply Etna to usage logs collected for analytics purposes. However, this would require dealing with a diverse set of environments (different screen sizes and website versions) and very noisy traces.

UI script maintenance. To handle website updates, Etna could automatically test and update UI scripts associated with all extracted actions. A hybrid approach based on demonstration and reinforcement learning [65] could automate the update process.

Failure points. Etna supports most common UI elements and interaction types. It does not model zooming, scrolling, or swiping interactions, and also does not model some customized widgets such as date pickers. Etna does not work on websites that block the ChromeDriver – in our experience, this is frequent with flight and real-estate websites. Action replay can also fail due to delays in loading collections, complex widgets or animations. Etna's produced timelines can be affected by noisy UI events. Etna currently handles noise such as back interactions or clicks outside the main body, but does not handle error-fixing interactions, browsing, or GUI-exploration actions unrelated to the task.

7 CONCLUSIONS

Web automation has moved beyond traditional testing and data scraping scenarios to automate critical enterprise processes and mainstream user experiences such as personal AI assistants. Current PBD tools for UI automation are insufficient to support this new era of application automation. We presented Etna, a system comprising several novel techniques to extract action-related knowledge from web interaction traces and to execute the extracted actions reliably in real-world websites. We also described use cases where Etna has been instrumental. Overall, Etna brings many advantages to developers who are building modern automation experiences, and represents a first step towards the goal of expanding web knowledge graphs to include information on actions and tasks.

ACKNOWLEDGMENTS

We are grateful to the anonymous shepherd and reviewers for their feedback. We thank Sahisnu Mazumder for being one of the early users of Etna and for implementing the NLP use case. We also thank many Microsoft colleagues for their useful feedback over the years.

REFERENCES

[1] The Open Graph protocol, 2021. https://ogp.me/.

[2] Schema.org, 2021. https://schema.org/.

[3] Q. Ai, V. Azizi, X. Chen, and Y. Zhang. Learning heterogeneous knowledge base embeddings for explainable recommendation. *Algorithms*, 11(9):137, Sep 2018. ISSN 1999-4893. doi: 10.3390/a11090137. URL http://dx.doi.org/10.3390/a11090137.

[4] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web Navigation with the WebVCR. In *Proc. of the 9th International World Wide Web Conference on Computer Networks*, pages 503–517, 2000.

[5] Y. Artzi and L. Zettlemoyer. Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013. doi: 10.1162/tacl_a_00209. URL https://www.aclweb.org/anthology/Q13-1005.

[6] Automation Anywhere, 2021. https://www.automationanywhere.com.

[7] S. Barman, S. Chasins, R. Bodik, and S. Gulwani. Ringer: Web Automation by Demonstration. In *Proc. f the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 748–764. ACM, 2016. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984020. URL http://doi.acm.org/10.1145/2983990.2984020.

[8] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 309–321, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380328. URL https://doi.org/10.1145/3377811.3380328.

[9] R. Bhowmik and G. de Melo. Be Concise and Precise: Synthesizing Open-Domain Entity Descriptions from Facts. In *The World Wide Web Conference*, WWW '19, pages 116–126. ACM, 2019. ISBN 9781450366748. doi: 10.1145/3308558.3313656. URL https://doi.org/10.1145/3308558.3313656.

[10] R. Blanco, H. Joho, A. Jatowt, H. Yu, and S. Yamamoto. Overview of NTCIR-13 Actionable Knowledge Graph (AKG) Task. In *Proc. of the NTCIR-13 Conference*, 2017.

[11] Blue Prism, 2021. https://www.blueprism.com.

[12] A. Bordes, J. Weston, R. Collobert, and Y. Bengio. Learning structured embeddings of knowledge bases. In *Proc. of the 25th AAAI Conference on Artificial Intelligence*, AAAI '11, pages 301–306. AAAI Press, 2011.

[13] S. R. K. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. Reinforcement Learning for Mapping Instructions to Actions. In *Proc. of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 82–90, USA, 2009. Association for Computational Linguistics. ISBN 9781932432459.

[14] A. Broder. A Taxonomy of Web Search. *SIGIR Forum*, 36(2):3–10, Sept. 2002. ISSN 0163-5840. doi: 10.1145/792550.792552. URL https://doi.org/10.1145/792550.792552.

[15] P. A. Brooks and A. M. Memon. Automated Gui Testing Guided by Usage Profiles. In *Proc. of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 333–342. ACM, 2007. ISBN 9781595938824. doi: 10.1145/1321631.1321681. URL https://doi.org/10.1145/1321631.1321681.

[16] G. S. Central. Google knowledge graph search api, 2021. URL https://developers.google.com/knowledge-graph.

[17] M. Chang, B. Lafreniere, J. Kim, G. Fitzmaurice, and T. Grossman. Workflow Graphs: A Computational Model of Collective Task Strategies for 3D Design Software. In *Proc. of Graphics Interface 2020*, GI 2020, pages 114 – 124, 2020. ISBN 978-0-9947868-5-2. doi: 10.20380/GI2020.13.

[18] S. E. Chasins, M. Mueller, and R. Bodik. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proc. of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 963–975. ACM, 2018. ISBN 9781450359481. doi: 10.1145/3242587.3242661. URL https://doi.org/10.1145/3242587.3242661.

[19] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 322–334, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380327. URL https://doi.org/10.1145/3377811.3380327.

[20] Y. Chen, M. Pandey, J. Y. Song, W. S. Lasecki, and S. Oney. Improving crowd-supported gui testing with structural guidance. In *Proc. of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13. ACM, 2020. ISBN 9781450367080. doi: 10.1145/3313831.3376835. URL https://doi.org/10.1145/3313831.3376835.

[21] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640. ACM, 2013. ISBN 9781450323741. doi: 10.1145/2509136.2509552. URL https://doi.org/10.1145/2509136.2509552.

[22] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery. Learning to construct knowledge bases from the world wide web. *Artif. Intell.*, 118(1–2):69–113, Apr. 2000. ISSN 0004–3702. doi: 10.1016/S0004-3702(00)00004-7. URL https://doi.org/10.1016/S0004-3702(00)00004-7.

[23] B. B. Dalvi, W. W. Cohen, and J. Callan. WebSets: Extracting Sets of Entities from the Web Using Unsupervised Information Extraction. In *Proc. of the 5th ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 243–252. ACM, 2012. ISBN 9781450307475. doi: 10.1145/2124295.2124327. URL https://doi.org/10.1145/2124295.2124327.

[24] B. Deka, Z. Huang, and R. Kumar. ERICA: Interaction Mining Mobile Apps. In *Proc. of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 767–776. ACM, 2016. ISBN 9781450341899. doi: 10.1145/2984511.2984581. URL https://doi.org/10.1145/2984511.2984581.

[25] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proc. of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 845–854. ACM, 2017. ISBN 9781450349819. doi: 10.1145/3126594.3126651. URL https://doi.org/10.1145/3126594.3126651.

[26] B. Deka, Z. Huang, C. Franzen, J. Nichols, Y. Li, and R. Kumar. ZIPT: Zero-Integration Performance Testing of Mobile App Designs. In *Proc. of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 727–736. ACM, 2017. ISBN 9781450349819. doi: 10.1145/3126594.3126647. URL https://doi.org/10.1145/3126594.3126647.

[27] M. Ermuth and M. Pradel. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *Proc. of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 82–93. ACM, 2016. ISBN 9781450343909. doi: 10.1145/2931037.2931053. URL https://doi.org/10.1145/2931037.2931053.

[28] O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised Named-Entity Extraction from the Web: An Experimental Study. *Artif. Intell.*, 165(1):91–134, June 2005. ISSN 0004–3702.

[29] W. Feng, H. H. Zhuo, and S. Kambhampati. Extracting Action Sequences from Texts Based on Deep Reinforcement Learning. In *Proc. of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4064–4070. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/565. URL https://doi.org/10.24963/ijcai.2018/565.

[30] Y. Gao, J. Liang, B. Han, M. Yakout, and A. Mohamed. Building a large-scale, accurate and fresh knowledge graph, 2018. URL https://kdd2018tutorialt39.azurewebsites.net/. Tutorial at KDD '18.

[31] D. GerÃşnimo and H. KjellstrÃűm. Unsupervised surveillance video retrieval based on human action and appearance. In *2014 22nd International Conference on Pattern Recognition*, pages 4630–4635, 2014. doi: 10.1109/ICPR.2014.792.

[32] D. Gowsikhaa, S. Abirami, and R. Baskaran. Automated human behavior analysis from surveillance videos: A survey. *Artificial Intelligence Review*, 42(4):747–765, Dec. 2014. ISSN 0269-2821. doi: 10.1007/s10462-012-9341-3.

[33] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proc. POPL '11*, pages 317–330, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL http://doi.acm.org/10.1145/1926385.1926423.

[34] I. Gur, U. Rueckert, A. Faust, and D. Hakkani-Tur. Learning to Navigate the Web. In *Proc. of ICLR '19*, May 6–9 2019.

[35] M. Han, P.-H. Wuillemin, and P. Senellart. Focused Crawling Through Reinforcement Learning. In T. Mikkonen, R. Klamma, and J. Hernández, editors, *Web Engineering*, pages 261–278, 2018. ISBN 978-3-319-91662-0.

[36] Z. He, S. Sunkara, X. Zang, Y. Xu, L. Liu, N. Wichers, G. Schubiner, R. B. Lee, and J. Chen. ActionBert: Leveraging User Actions for Semantic Understanding of User Interfaces. In *35th AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 5931–5938, 2021.

[37] I. Hernández, C. R. Rivero, and D. Ruiz. Deep Web crawling: a survey. *World Wide Web*, Jun 2018. ISSN 1573–1413. doi: 10.1007/s11280-018-0602-1. URL https://doi.org/10.1007/s11280-018-0602-1.

[38] D. Hilbert and D. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32, 08 1999. doi: 10.1145/371578.371593.

[39] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec. Strategies for Pre-training Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020*, 2020. URL https://openreview.net/forum?id=HJlWWJSFDH.

[40] D. Hupp and R. C. Miller. Smart Bookmarks: Automatic Retroactive Macro Recording on the Web. In *Proc. of UIST '07*, pages 81–90, 2007.

[41] J. D. Hwang, C. Bhagavatula, R. L. Bras, J. Da, K. Sakaguchi, A. Bosselut, and Y. Choi. (Comet-) Atomic 2020: On Symbolic and Neural Commonsense Knowledge Graphs. In *35th AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 6384–6392. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16792.

[42] T. Intharah, D. Turmukhambetov, and G. J. Brostow. Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions. In *Proc. of the 22nd International Conference on Intelligent User Interfaces*, IUI '17, pages 233–243. ACM, 2017. ISBN 9781450343480. doi: 10.1145/3025171.3025176. URL https://doi.org/10.1145/3025171.3025176.

[43] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on*

Neural Networks and Learning Systems, pages 1–21, 2021. doi: 10.1109/TNNLS.2021.3070843.

[44] L. Jiang, Z. Wu, Q. Feng, J. Liu, and Q. Zheng. Efficient Deep Web Crawling Using Reinforcement Learning. In *Proc. of PAKDD'10*, pages 428–439, 2010. ISBN 3-642-13656-7, 978-3-642-13656-6. doi: 10.1007/978-3-642-13657-3_46.

[45] X. Kang, Y. Wu, and F. Ren. Toward action comprehension for searching: Mining actionable intents in query entities. *Journal of the Association for Information Science and Technology*, 71(2):143–157, 2020. doi: 10.1002/asi.24220. URL https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.24220.

[46] Katalon Studio, 2021. https://www.katalon.com.

[47] M. Khansari, D. Kappler, J. Luo, J. Bingham, and M. Kalakrishnan. Action image representation: Learning scalable deep grasping policies with zero real world data, 2020.

[48] C. Kiddon, G. T. Ponnuraj, L. Zettlemoyer, and Y. Choi. Mise en place: Unsupervised interpretation of instructional recipes. In *Proc. of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 982–992, Lisbon, Portugal, Sept. 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1114. URL https://www.aclweb.org/anthology/D15-1114.

[49] D. Kim and A. Oh. How to find your friendly neighborhood: Graph attention design with self-supervision. In *9th International Conference on Learning Representations, ICLR 2021*, 2021. URL https://openreview.net/forum?id=Wi5KUNlqWty.

[50] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee. Using assertions to help end-user programmers create dependable web macros. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 124–134, 2008. ISBN 9781595939951. doi: 10.1145/1453101.1453119.

[51] M. Lacity and L. P. Willcocks. Robotic Process Automation at Telefonica O2. *MIS Quarterly Executive*, 15(1), 2016.

[52] T. Lau. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Mag.*, 30(4):65–67, 2009. doi: 10.1609/aimag.v30i4.2262. URL https://doi.org/10.1609/aimag.v30i4.2262.

[53] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.*, 53(1-2):111–156, Oct. 2003. ISSN 0885-6125. doi: 10.1023/A:1025671410623. URL https://doi.org/10.1023/A:1025671410623.

[54] T. Lau, J. A. Cerruti, G. Manzato, M. Bengualid, J. P. Bigham, and J. Nichols. A Conversational Interface to Web Automation. pages 229–238, 10 2010. doi: 10.1145/1866029.1866067.

[55] V. Le and S. Gulwani. FlashExtract: A Framework for Data Extraction by Examples. *SIGPLAN Not.*, 49(6):542–553, June 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594333. URL https://doi.org/10.1145/2666356.2594333.

[56] T. Y. Lee, C. Dugan, and B. B. Bederson. Towards understanding human mistakes of programming by example: An online user study. In *Proc. of the 22nd International Conference on Intelligent User Interfaces*, IUI '17, pages 257–261, 2017. ISBN 9781450343480. doi: 10.1145/3025171.3025203.

[57] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proc. of CHI '08*, pages 1719–1728, 2008.

[58] I. Li, J. Nichols, T. Lau, C. Drews, and A. Cypher. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proc. of CHI '10*, pages 723–732, 2010.

[59] T. J. Li, I. Labutov, X. N. Li, X. Zhang, W. Shi, W. Ding, T. M. Mitchell, and B. A. Myers. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018*, pages 105–114. IEEE Computer Society, 2018. doi: 10.1109/VLHCC.2018.8506506. URL https://doi.org/10.1109/VLHCC.2018.8506506.

[60] T. J. Li, M. Radensky, J. Jia, K. Singarajah, T. M. Mitchell, and B. A. Myers. PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proc. of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST 2019*, pages 577–589. ACM, 2019. doi: 10.1145/3332165.3347899. URL https://doi.org/10.1145/3332165.3347899.

[61] T. J. Li, L. Popowski, T. M. Mitchell, and B. A. Myers. Screen2Vec: Semantic embedding of GUI screens and GUI components. *CoRR*, abs/2101.11103, 2021. URL https://arxiv.org/abs/2101.11103.

[62] T. J.-J. Li and O. Riva. Kite: Building Conversational Bots from Mobile Apps. In *Proc. of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 96–109. ACM, 2018. ISBN 9781450357203. doi: 10.1145/3210240.3210339. URL https://doi.org/10.1145/3210240.3210339.

[63] T. J.-J. Li, A. Azaria, and B. A. Myers. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proc. of CHI '17*, pages 6038–6049, 2017. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025483. URL http://doi.acm.org/10.1145/3025453.3025483.

[64] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers. Programming IoT Devices by Demonstration Using Mobile Apps. In *Proc. of the International Symposium on End User Development (IS-EUD)*, volume 10303. Springer-Verlag, 2017. doi: https://doi.org/10.1007/978-3-319-58735-6_1.

[65] Y. Li and O. Riva. Glider: A reinforcement learning approach to extract UI scripts from websites. In *Proc. of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, July 2021.

[66] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proc. of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 8198–8210. Association for Computational Linguistics, 2020. URL https://www.aclweb.org/anthology/2020.acl-main.729/.

[67] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proc. of the 14th International Conference on Intelligent User Interfaces*, IUI '09, pages 97–106. ACM, 2009. ISBN 9781605581682. doi: 10.1145/1502650.1502667. URL https://doi.org/10.1145/1502650.1502667.

[68] T.-Y. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár. Focal Loss for Dense Object Detection. In *Proc. of the 2017 IEEE International Conference on Computer Vision (ICCV '07)*, pages 2999–3007, 2017.

[69] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In *Proc. of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122. IEEE Press, 2015. ISBN 9780769555942.

[70] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In M. B. Rosson and D. J. Gilmore, editors, *Proc. of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007*, pages 943–946. ACM, 2007. doi: 10.1145/1240624.1240767. URL https://doi.org/10.1145/1240624.1240767.

[71] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. *CoRR*, abs/1802.08802, 2018. URL http://arxiv.org/abs/1802.08802.

[72] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar. Learning Design Semantics for Mobile Apps. In *Proc. of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 569–579. ACM, 2018. ISBN 9781450359481. doi: 10.1145/3242587.3242650. URL https://doi.org/10.1145/3242587.3242650.

[73] X. Luo, L. Liu, Y. Yang, L. Bo, Y. Cao, J. Wu, Q. Li, K. Yang, and K. Q. Zhu. AliCoCo: Alibaba E-Commerce Cognitive Concept Net. In *Proc. of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 313–327, 2020. ISBN 9781450367356. doi: 10.1145/3318464.3386132. URL https://doi.org/10.1145/3318464.3386132.

[74] M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the Talk: Connecting Language, Knowledge, and Action in Route Instructions. In *Proc. of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1475–1482. AAAI Press, 2006. ISBN 9781577352815.

[75] S. Maji, L. Bourdev, and J. Malik. Action recognition from a distributed representation of pose and appearance. In *Proc. of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 3177–3184, 2011. ISBN 9781457703942. doi: 10.1109/CVPR.2011.5995631. URL https://doi.org/10.1109/CVPR.2011.5995631.

[76] K. Mao, M. Harman, and Y. Jia. Crowd Intelligence Enhances Automated Mobile Testing. In *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 16–26. IEEE Press, 2017. ISBN 9781538626849.

[77] S. Mazumder and O. Riva. FLIN: A flexible natural language interface for web navigation. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NACCL 2021)*, June 2021.

[78] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval*, 3(2): 127–163, Jul 2000. ISSN 1573-7659. doi: 10.1023/A:1009953814988.

[79] H. Mei, M. Bansal, and M. R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proc. of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2772–2778. AAAI Press, 2016.

[80] A. M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, Sept. 2007. ISSN 0960-0833.

[81] A. K. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. T. Kalai. A Machine Learning Framework for Programming by Example. In *Proc. of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages 187–195, 2013.

[82] Microsoft. Bot Framework, 2021. https://dev.botframework.com/.

[83] Microsoft Azure. Introduction to gremlin api in azure cosmos db, March 2021. https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction.

[84] S. Mori, H. Maeta, Y. Yamakata, and T. Sasada. Flow graph corpus from recipe texts. In *Proc. of the 9th International Conference on Language Resources and Evaluation (LREC'14)*, pages 2370–2377, Reykjavik, Iceland, May 2014. European Language Resources Association (ELRA). URL http://www.lrec-conf.org/proceedings/lrec2014/pdf/763_Paper.pdf.

[85] Moz - SEO Learning Center. Seo meta description, 2021. https://moz.com/learn/seo/meta-description.

[86] B. A. Myers. *Peridot: Creating User Interfaces by Demonstration*, pages 125–153. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262032139.

[87] B. A. Myers and R. McDaniel. Demonstrational interfaces: Sometimes you need a little intelligence, sometimes you need a lot. In H. Lieberman, editor, *Your Wish is My Command*, Interactive Technologies, pages 45–60. Morgan Kaufmann, 2001. ISBN 978-1-55860-688-3. doi: https://doi.org/10.1016/B978-155860688-3/50004-X.

[88] W. M. Newman. A system for interactive graphical programming. In *Proc. of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 47–54. ACM, 1968. ISBN 9781450378970. doi: 10.1145/1468075.1468083. URL https://doi.org/10.1145/1468075.1468083.

[89] P. Pasupat and P. Liang. Zero-shot Entity Extraction from Web Pages. In *Proc. of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 391–401. Association for Computational Linguistics, June 2014. doi: 10.3115/v1/P14-1037. URL https://www.aclweb.org/anthology/P14-1037.

[90] O. Polozov and S. Gulwani. LaSEWeb: Automating Search Strategies over Semi-structured Web Data. In *Proc. of KDD '14*, pages 741–750, 2014. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623761. URL http://doi.acm.org/10.1145/2623330.2623761.

[91] J. Qiu, Q. Chen, Y. Dong, J. Zhang, H. Yang, M. Ding, K. Wang, and J. Tang. GCC: Graph Contrastive Coding for Graph Neural Network Pre-Training. In *Proc. of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 1150–1160. ACM, 2020. ISBN 9781450379984. doi: 10.1145/3394486.3403168. URL https://doi.org/10.1145/3394486.3403168.

[92] J. Redmon, S. Divvala, R. B. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proc. of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*, pages 779–788, 2016.

[93] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:1137–1149, 2015.

[94] J. Rennie and A. McCallum. Using Reinforcement Learning to Spider the Web Efficiently. In *Proc. of ICML '99*, pages 335–343, 1999. ISBN 1-55860-612-2.

[95] H. Reza, S. Endapally, and E. Grant. A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. In *Proc. of the International Conference on Information Technology*, ITNG '07, pages 366–370. IEEE Computer Society, 2007. ISBN 0769527760. doi: 10.1109/ITNG.2007.9. URL https://doi.org/10.1109/ITNG.2007.9.

[96] A. Richard, H. Kuehne, and J. Gall. Weakly supervised action learning with RNN based fine-to-coarse modeling. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1273–1282, 2017. doi: 10.1109/CVPR.2017.140.

[97] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. Examining Image-Based Button Labeling for Accessibility in Android Apps through Large-Scale Analysis. In *Proc. of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '18, pages 119–130. ACM, 2018. ISBN 9781450356503. doi: 10.1145/3234695.3236364. URL https://doi.org/10.1145/3234695.3236364.

[98] A. Safonov, J. A. Konstan, and J. V. Carlis. End-user Web Automation: Challenges, Experiences, Recommendations. In *Proc. of WebNet 2001*, pages 1077–1085, 2001.

[99] Selenium Browser Automation, 2021. https://www.selenium.dev/.

[100] Selenium IDE: Open source record and playback test automation for the web, 2021. https://www.selenium.dev/selenium-ide/.

[101] A. R. Sereshkeh, G. Leung, K. Perumal, C. Phillips, M. Zhang, A. Fazly, and I. Mohomed. VASTA: A Vision and Language-Assisted Smartphone Task Automation System. In *Proc. of the 25th International Conference on Intelligent User Interfaces*, IUI '20, pages 22–32. ACM, 2020. ISBN 9781450371186. doi: 10.1145/3377325.3377515. URL https://doi.org/10.1145/3377325.3377515.

[102] R. K. Shehady and D. P. Siewiorek. A Method to Automate User Interface Testing Using Variable Finite State Machines. In *Proc. of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, page 80. IEEE Computer Society, 1997. ISBN 0818678313.

[103] T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang. World of Bits: An Open-Domain Platform for Web-Based Agents. In *Proc. of the 34th International Conference on Machine Learning*, volume 70, pages 3135–3144, 06–11 Aug 2017.

[104] A. Sil, F. Huang, and A. Yates. Extracting Action and Event Semantics from Web Text. In *AAAI Fall Symposium: Commonsense Knowledge*, 2010.

[105] spaCy, 2021. https://spacy.io/models/en.

[106] R. Speer and C. Havasi. Representing general relational knowledge in ConceptNet 5. In *Proc. of the 8th International Conference on Language Resources and Evaluation (LREC'12)*, pages 3679–3686. ELRA, May 2012. URL http://www.lrec-conf.org/proceedings/lrec2012/pdf/1072_Paper.pdf.

[107] A. Sugiura and Y. Koseki. Internet Scrapbook: Automating Web Browsing Tasks by Demonstration. In *Proc. of the 11th Annual ACM Symposium on User Interface Software and Technology*, UIST '98, pages 9–18, 1998. ISBN 1581130341. doi: 10.1145/288392.288395. URL https://doi.org/10.1145/288392.288395.

[108] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid. VideoBERT: A Joint Model for Video and Language Representation Learning. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV*, pages 7463–7472. IEEE, 2019. doi: 10.1109/ICCV.2019.00756. URL https://doi.org/10.1109/ICCV.2019.00756.

[109] M. Surdeanu, J. Tibshirani, R. Nallapati, and C. D. Manning. Multi-instance multi-label learning for relation extraction. In *Proc. of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 455–465. Association for Computational Linguistics, 2012.

[110] Swagger. API Development for Everyone, 2021. https://swagger.io/.

[111] Tech Crunch. Gartner finds RPA is fastest growing market in enterprise software, 24 June 2019. https://techcrunch.com/2019/06/24/gartner-finds-rpa-is-fastest-growing-market-in-enterprise-software/.

[112] Tech Crunch. Google is bringing AI assistant Duplex to the web, May 7 2019. https://techcrunch.com/2019/05/07/google-is-bringing-ai-assistant-duplex-to-the-web/.

[113] Test Complete, 2021. https://smartbear.com/product/testcomplete/overview/.

[114] UiPath, 2021. https://www.uipath.com/.

[115] w3schools.com. HTML data attributes, 2021. https://www.w3schools.com/tags/att_global_data.asp.

[116] R. C. Wang and W. W. Cohen. Character-level Analysis of Semi-Structured Documents for Set Expansion. In *Proc. of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1503–1512. Association for Computational Linguistics, Aug. 2009. URL https://www.aclweb.org/anthology/D09-1156.

[117] G. Weikum, J. Hoffart, and F. M. Suchanek. Ten Years of Knowledge Harvesting: Lessons and Challenges. *IEEE Data Engineering Bulletin*, 39(3):41–50, September 2016. URL https://publications.cispa.saarland/965/.

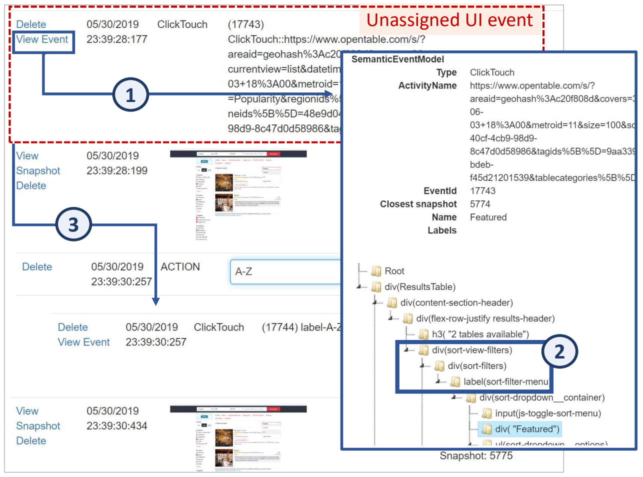
[118] Y. Xie, Z. Xu, Z. Wang, and S. Ji. Self-supervised learning of graph neural networks: A unified review. *CoRR*, abs/2102.10757, 2021. URL https://arxiv.org/abs/2102.10757.

[119] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI Screenshots for Search and Automation. In *Proc. of the 22nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192. ACM, 2009. ISBN 9781605587455. doi: 10.1145/1622176.1622213. URL https://doi.org/10.1145/1622176.1622213.

[120] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach, A. Everitt, and J. P. Bigham. Screen recognition: Creating accessibility metadata for mobile applications from pixels, 2021.

[121] D. Zhao, Z. Xing, C. Chen, X. Xia, and G. Li. ActionNet: Vision-based workflow action recognition from programming screencasts. In *Proc. of the 41st International Conference on Software Engineering*, ICSE '19, pages 350–361. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00049. URL https://doi.org/10.1109/ICSE.2019.00049.

[122] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph Neural Networks: A Review of Methods and Applications. *AI Open*, 1:57–81, 2020.



```
Delete      05/30/2019     ClickTouch   (17743)
View Event  23:39:28:177                ClickTouch::https://www.opentable.com/s/?
                                        areaid=geohash%3Ac20
                                        currentview=list&datetim
                                        03+18%3A00&metroid=
                                        =Popularity&regionids%
                                        neids%5B%5D=48e9d0
                                        98d9-8c47d0d58986&ta

View        05/30/2019
Snapshot    23:39:28:199
Delete

Delete      05/30/2019     ACTION       A-Z
            23:39:30:257

Delete      05/30/2019     ClickTouch   (17744) label-A-Z
View Event  23:39:30:257

View        05/30/2019
Snapshot    23:39:30:434
Delete
```

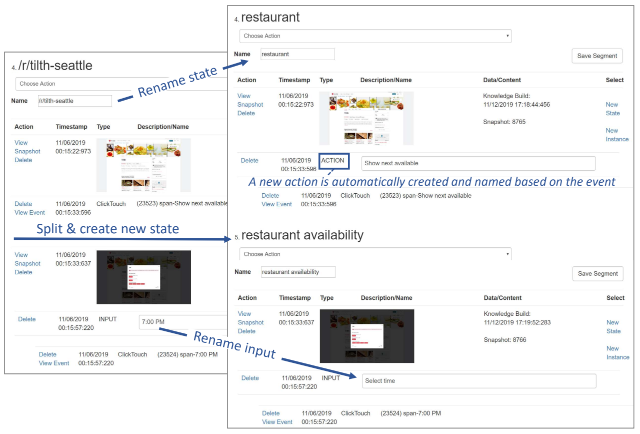
```
SemanticEventModel
            Type  ClickTouch
    ActivityName  https://www.opentable.com/s/?
                  areaid=geohash%3Ac20f808d&covers=
                  06-
                  03+18%3A00&metroid=11&size=100&s
                  40cf-4cb9-98d9-
                  8c47d0d58986&tagids%5B%5D=9aa33
                  bdeb-
                  f45d21201539&tablecategories%5B%5
         EventId  17743
Closest snapshot  5774
            Name  Featured
          Labels

Root
div(ResultsTable)
  div(content-section-header)
    div(flex-row-justify results-header)
      h3( "2 tables available")
      div(sort-view-filters)
        div(sort-filters)
          label(sort-filter-menu)
            div(sort-dropdown__container)
              input(js-toggle-sort-menu)
              div( "Featured")
```

Figure 13: 3-step process to resolve an unassigned UI event. The user "drags" the unassigned event under the correct action cluster.

A TRACE EDITING

Users can assign unassigned UI events to existing or new action/input clusters using a simple process. To illustrate we use


Rename state
Split & create new state
Rename input
A new action is automatically created and named based on the event

4. /r/tilth-seattle

Choose Action

Name /r/tilth-seattle

Action	Timestamp	Type	Description/Name
View Snapshot Delete	11/06/2019 00:15:22:973		
Delete View Event	11/06/2019 00:15:33:596	ClickTouch	(23523) span-Show next available
View Snapshot Delete	11/06/2019 00:15:33:637		
Delete	11/06/2019 00:15:57:220	INPUT	7:00 PM
Delete View Event	11/06/2019 00:15:57:220	ClickTouch	(23524) span-7:00 PM

4. restaurant

Choose Action

Name restaurant Save Segment

Action	Timestamp	Type	Description/Name	Data/Content	Select
View Snapshot Delete	11/06/2019 00:15:22:973			Knowledge Build: 11/12/2019 17:18:44:456 Snapshot: 8765	New State New Instance
Delete	11/06/2019 00:15:33:596	ACTION	Show next available		
Delete View Event	11/06/2019 00:15:33:596	ClickTouch	(23523) span-Show next available		

5. restaurant availability

Choose Action

Name restaurant availability Save Segment

Action	Timestamp	Type	Description/Name	Data/Content	Select
View Snapshot Delete	11/06/2019 00:15:33:637			Knowledge Build: 11/12/2019 17:19:52:283 Snapshot: 8766	New State New Instance
Delete	11/06/2019 00:15:57:220	INPUT	Select time		
Delete View Event	11/06/2019 00:15:57:220	ClickTouch	(23524) span-7:00 PM		

Figure 14: Editing a trace timeline: a state and an input are renamed, and a new state is created which automatically spawns a new action (automatically labeled based on the UI event as "Show next available").

the example in Fig. 13. First of all, in the timeline view a user can easily recognize UI events that do not belong to any cluster (based on horizontal offset). The user can see the event timestamp and previous/subsequent UI snapshots which can help locate the event in the logical flow. For more details about a UI event, the user can open a separate window including a snapshot of the UI tree with highlighted where the event occurred (step ①). In this example, the user can see that the event occurred in the "sort-filter-menu" (step ②). Back in the timeline, the user can see that the subsequent event is associated with the action "A-Z"; they recognize the unassigned event is part of the "A-Z sorting" action and drag it under it (step ③).

Fig. 14 illustrates an additional editing example where the user renamed state #4 by assigning a more general title ("restaurant"), created a new state called "restaurant availability" which triggered the generation of a new transitioning action ("Show next available") in the previous state, and renamed the "7:00 PM" input as "Select time". All these operations are rather simple and accomplished in few clicks.