Accessible Web Automation

A Dissertation Presented

by

Yury Puzis

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2013

Stony Brook University

The Graduate School

Yury Puzis

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. I.V. Ramakrishnan
Department of Computer Science
Dissertation Advisor

Dr. Yevgen Borodin
Department of Computer Science
Dissertation Advisor

Dr. C.R. Ramakrishnan
Department of Computer Science
Dissertation Advisor

Dr. David Warren
Department of Computer Science
Chairperson of Defense

Dr. Jeffrey P. Bigham
Human-Computer Interaction
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Accessible Web Automation

by

Yury Puzis

Doctor of Philosophy
in
Computer Science

Stony Brook University

2013

The Web is far less usable and accessible for people with visual impairments than it is for sighted people. There are only a handful of screen-reader applications that improve web browsing accessibility by reading aloud the content of web pages. However, accessibility does not imply usability; blind people end up spending minutes on browsing tasks that take sighted people seconds. This lack of productivity using the Web poses a significant barrier for people with visual impairments in education, employment, healthcare, etc.

Web automation, a process of automating browsing actions on behalf of the user, has the potential to bridge the divide between the ways visually impaired people and sighted people access the Web. Typical automation interfaces require that the user record or handcraft a macro, a set of instructions for automating a useful sequence of browsing actions, so that these actions can be replayed in the future. In this dissertation, I present a novel approach to Web automation that completely automates those tasks, while simultaneously making the web automation easier to control.

The first pillar of the presented approach is a novel model-based algorithm that uses the past browsing history and the current web page as the browsing context to predict the most probable browsing actions that the user can do. The model is continuously and incrementally updated as the history evolves, thereby, ensuring the predictions are not 'outdated'. A unique aspect of the model is that its construction is 'unsupervised', i.e. does not require any kind of labeled training data.

The second pillar is a novel interface that lets the user focus on the objects associated with the most probable predicted browsing steps (e.g., clicking links and filling out forms), and facilitates automatic execution of the selected steps. Several studies blind participants showed that the proposed approach dramatically reduced the interaction time needed to accomplish typical browsing tasks, and the user interface was perceived to be much more usable than the standard screen-reading interfaces.

Contents

List of Figures

List of Tables

Glossary

Action: A browsing action executed by the user with the help of input hardware, or programmatically. Actions include interaction with the browser, and with a screen-reader via key presses, voice commands, gestures, etc.

Automation Instruction: Data that can be used by a web automation tool to execute a specific browsing action programmatically, on user's behalf.

Browsing Data: A hardcoded macro, a log of environment events (partial or full), or any information learned from the events log or any other applicable information source. Can be used to create automation instruction(s).

DOM: Document Object Model, structured as a tree. Provides a standard, language-independent and implementation-neutral application programming interface (API) for building, traversing, and modifying XML (in particular, HTML) documents.

Environment: The user agents (e.g., browser, media player), the assistive technology (e.g., screen-reader, magnifier), the operating system.

Event: Events are emitted from the environment in response to browsing actions. Can be recorded as browsing data.

History: Sequential record of browsing data.

Information Filtering System: A system that analyzes an incoming information stream, filters out less important or noisy information, and presents the user with the most relevant information from the stream.

Interface Agent: A system that observes user interactions with the user interface, and may be triggered (explicitly or implicitly) to interact with the user interface on user's behalf.

Macro: A sequence of automation instructions.

PBD: Programming by demonstration (also: programming by example).

Webpage Element: A visual primitive of the webpage. Each webpage element corresponds to some DOM node. The semantics of the webpage element can be inferred from the DOM node: its type, its relationship to other nodes, position inside the tree, and the flat list of attributes (optionally) stored within the node.

Acknowledgments

This work would not be possible without many people that helped me on the thorny path of a Ph.D. student. First, I would like to thank my friends, colleagues, and co-authors: Elizabeth Fanning, Spyros Hadjichristodoulou, Song Feng, Faisal Ahmed, Andrii Soviak, Muhammad Asiful Islam, Rami Puzis, Vasily Tarasov, and Tatsiana Mironava for their help and friendship over the years. I would like to offer my special thanks to Valentyn Melnyk for his countless hours of technically excellent work and insightful discussions. I would also like to extend thanks to Glenn Dausch, another co-author and a usability expert at Charmtech Labs. Glenn's insightful observations and deep understanding of the complexities of non-visual browsing were invaluable in putting this dissertation on a firm practical ground.

I would like to extend my greatest thanks to my advisors, I.V. Ramakrishnan, Yevgen Borodin, and C.R. Ramakrishnan for their infinite patience and guidance over the years. Their help, the friendly atmosphere they helped create, and the insightful discussions were of great importance to my professional growth and this research. I also deeply thank Yevgen Borodin for suggesting to join the research on web accessibility and for his unwavering friendship.

No good research is possible without financial support. This work was supported in part by National Science Foundation award IIS-0808678 and Small Business Innovative Research Grants H133S110023 and H133S120067 from the National Institute of Disability and Rehabilitation Research.

Chapter 1

Introduction

1.1 The Challenge of Non-Visual Web Access

The Web plays an important role in our lives and has become a vital infrastructure for participating in our fast-paced electronic society. A report [18] by the Internet World Stats shows that Internet usage has skyrocketed by more than 566% since 2000, to include almost a third of the global population as of June 2012 (over 2.4 billion people). We use the Web to obtain and exchange information, shop, pay bills, make travel arrangements, apply for college or employment, communicate with others, participate in civic activities, etc. The utility of the Web is even greater for disabled users who once required human assistance with many of these activities, which can now be done using the Web. At the same time, disabled users have to deal with significant limitations of existing Web technology in the area of accessibility.

Each type of disability requires special attention from developers for developing technology that can truly leverage the enabling potential of the Web. Different types of disabilities include motor (limited or slow motor control), auditory, neurological, speech, cognitive (e.g., learning, focus disabilities), and visual impairments (blindness, low vision, color blindness). This work will concentrate on improving Web accessibility for blind users and users with low vision (for simplicity referred to below as "visually impaired").

While browsing, users with low vision can use magnifiers (e.g., MAGic [6] and ZoomText [1]) that enlarge a small portion of the screen and require a lot of panning to see the rest of the screen. Magnifiers often have some text narration functionality as well. However, completely blind users (and some low-vision users) rely on screen readers (e.g., JAWS [4], NVDA [11], Window-Eyes [7], VoiceOver [2], Super Nova [3]) for web browsing. Screen-readers are assistive tools that narrate the screen content and allow users to interact with it (e.g., fill and submit forms, follow links and press buttons, etc).

An accessible webpage is a webpage that can be used by a visually-impaired user in all the ways intended by its authors: perceiving its content (e.g., reading an article), timely perceiving webpage updates (e.g., user notifications), interacting with controls (e.g., clicking buttons, filling out forms), making your own contributions (e.g., leaving comments), and navigating to other webpages. The accessibility of a webpage for a specific user depends on the user's disabilities, technical savviness, and experience, as well as on concerted work of a number of technologies: the webpage design and adherence to Web Accessibility Initiative (WAI) guidelines [20]; the user agent: browser, media player, etc.; the available input and output hardware; and, most importantly, the screen-reader.

In contrast to sighted users, visually-impaired users cannot just "glance" over a webpage whenever there is a need to find quickly the right button to click and text to read, or identify an important webpage update due to a serial mode of interaction with the webpage imposed by the screen-reader. Regrettably, the existing assistive technologies are often not sufficient to bridge the accessibility divide between visually-impaired users and users without disabilities. Among the underlying causes of the situation, I single out the three: website authors often do not follow the WAI guidelines and do not use accessibility checkers to verify the accessibility of their websites; browsers often do not expose all the accessibility information required by the assistive technologies; and screen readers often offer users only partial and inefficient access to webpages.

In order for a screen-reader to provide an adequate representation of a webpage and enable access to its functionality, the screen reader needs to interpret the semantics of the webpage correctly. Providing adequate webpage representation requires narrating the content in the order most reflective of webpage designer's intention, as well as the user's preferences - a hard problem due to the fact that the webpages are designed for visual consumption as a two-dimensional canvas, rather than a one-dimensional stream of information. Providing adequate webpage representation also means enabling easy access to all the webpage's functionality, despite the fact that webpages are rather often optimized for users of pointing devices e.g. the mouse, which cannot be efficiently used by blind users. Unfortunately, screen-readers are generally ignoring layout, graphics, and user's priorities, instead reading aloud all the textual content in webpages. Screen readers designed for keyboard-equipped devices enable sequential navigation over the webpage content and provide numerous shortcuts that can speed up the navigation by 'jumping' to specific sections of a webpage. Screen readers designed for touch-screen-equipped devices enable touch-based spatial navigation, at the expense of a significantly reduced arsenal of "shortcuts", replaced with gestures.

In most cases, however, even expert users fall back to the most basic navigation, in which they go through items sequentially, one by one. Most advanced users become experts in tens and hundreds of specialized shortcuts, and sometimes are even modifying the behavior of screen readers via custom, handcrafted scripts.

Accessibility shortcomings in the webpage design, the sequential mode of interaction with webpages using screen readers, and the inability of visually-impaired users to determine whether a piece of content is important before they listen to it often result in a high cognitive load that, in turn, impacts users' browsing speed and their overall browsing experience. As a result, visually-impaired users spend significantly more time on seemingly simple online task than users without disabilities. In order to improve their browsing experience, users are developing efficient behavioral strategies [55], but are unable to circumvent the fundamental shortcomings of the technology in their use.

Attempts to improve the situation are continuing in multiple directions. *Spatial navigation* [44, 59] uses 3D audio to provide additional information about the webpage layout. *Haptic navigation* [75] uses specialized hardware, such as a haptic mouse, to commutate additional information about the webpage content. Different *text summarization* techniques [45, 10, 63] have been proposed to help the user to get a quick overview of continuous text present on a webpage, and *text skimming* [5, 22, 23, 24] has been developed to skim quickly through large amounts of text similarly to how it is done by the sighted users. *Webpage transcoding* (automatic webpage modification) has been used to improve the accessibility of poorly designed webpages, and/or to customize a webpage to the user's needs and preferences [70]. Finally, *Web automation* [8, 33], a process of automating browsing actions on behalf of the user, has been explored with the aim of reducing the amount of user's interaction with the webpage.

Web automation has the potential to bridge the web accessibility divide between the ways people with and without vision impairments use the Web. The traditional approach to Web automation is via *macros*, pre-recorded sequences of instructions that can automate browsing steps. Macro-based automation tools are finding their way into mainstream technology as part of screen-readers, browser extensions, and browser features (e.g., form filling). The adoption of macro-based Web automation has been stifled by a number of challenges: (a) it requires that the user make an effort to create, manage, and then find and replay macros, (b) it lacks the flexibility necessary to allow the user to deviate from the prerecorded sequence of steps, or to choose between several options in each step of the macro, and (c) it does not address the need of visually-impaired users to find non-interactive content, e.g., an article, an error notification in a form, etc. Those difficulties make macro-based approaches too limiting to be useful for people with vision impairments; for example, none of the 19 blind participants of the experiment detailed later in this work (Section 3.4) reported using any kind of web automation technology.

1.2 Contributions

In this dissertation, I present a novel approach to accessible, non-visual web automation. Its salient aspects are embodied in the prototype system - *Automation Assistant* [69].

The contributions of this dissertation are as follows. First, I systematize and analyze complexities and constraints faced by designers of accessible web automation tools. Second, I present a novel, accessible, and usable non-visual web automation user interface (UI). The UI is designed to focus the user only on a limited number of browsing steps that are most useful at a given time. This is accomplished by continuously monitoring the user activity, and, when requested, guiding the user through browsing tasks step by step, providing relevant contextual suggestions, and enabling the user to confirm each step before the suggestions are executed. Third, I describe a novel method that enables the web automation UI. The method is based on a modified dynamic programming sequence alignment algorithm, and is used to predict the most probable actions the user can take at a given time when browsing the Web. The method uses "unsupervised", continuous, and incremental modeling of user browsing actions, i.e. the model is silently and efficiently updates after every user action, does not require any kind of labeled training data, and does not require re-training. The effectiveness of the approach was validated in experiments evaluating the prediction accuracy, in a user study evaluating the UI usability, as well as in a longitudinal study with the Automation Assistant prototype.

It is worthwhile pointing out that Automation Assistant can be viewed as an *interface agent* [47, 54] - a system that observes user interactions with the UI and interacts with the user or with the UI. This work is also related to *adaptive user interfaces*. An adaptive interface tailors the presentation of functionality to better fit an individual user's tasks, usage patterns, and abilities [41]. However, while automating interaction with the Web is an important aspect of the system, it is not its main strength. The most important aspect of the proposed approach, in the author's view, is a seamless integration of regular browsing with the ability to filter out the vast amounts of information present in webpages and laser-focusing the user on what is most important.

Chapter 2

Complexities of Practical Non-Visual Web Automation

In this chapter I will discuss different approaches to web automation (Section 2.1), outline the complexity of creating an accessible web automation tool from the computer-human interaction perspective (Section 2.2), and finally from a technical perspective (Section 2.3).

While the focus of this chapter is on *non-visual* web automation, it should be noted that, first, many of the points apply to web automation tools that do not impose such a constraint. And second, ‘non-visual’ does not imply ‘useful exclusively for blind and low-vision users’. Sighted users often fall into the category of *situationally visually impaired* for many temporary reasons: the desire (or the need) to rest their eyes, usage of devices with a small-screen form factor (e.g., smartphones), preference for auditory consumption of information, etc.

2.1 Approaches to Web Automation

We define the *environment* with which the user and the web automation tool (for example, Automation Assistant) interact to be the web browser coupled with the screen reader and the operating system (Figure 2.1). The user and the web automation tool interact with the environment by executing *browsing actions* (for brevity referred to here as “actions”). For example, the user can press a keyboard shortcut; the web automation tool can simulate a keyboard shortcut press on the user’s behalf. In response to the browsing actions, the environment triggers *events*, e.g., browser’s JavaScript events, screen reader’s virtual cursor movement, etc.

An *automation instruction* is defined as the instruction that is needed to execute a specific browsing action programmatically on user's behalf. Web automation can be viewed as two distinct processes: generation and execution of automation instructions. In the generation phase, the sequence of actions or a single action to be automated is specified, optionally parameterized, and stored as some form of *browsing data*. Browsing data can be a hardcoded macro, a log of environment events (partial or full) triggered by user's actions, or any information learned from the events' log or any other applicable information source. Actions include interaction with the browser and with a screen reader via key presses, voice commands, gestures, etc. In the execution phase, the browsing data is used to infer concrete, parametrized actions that are subsequently automated.

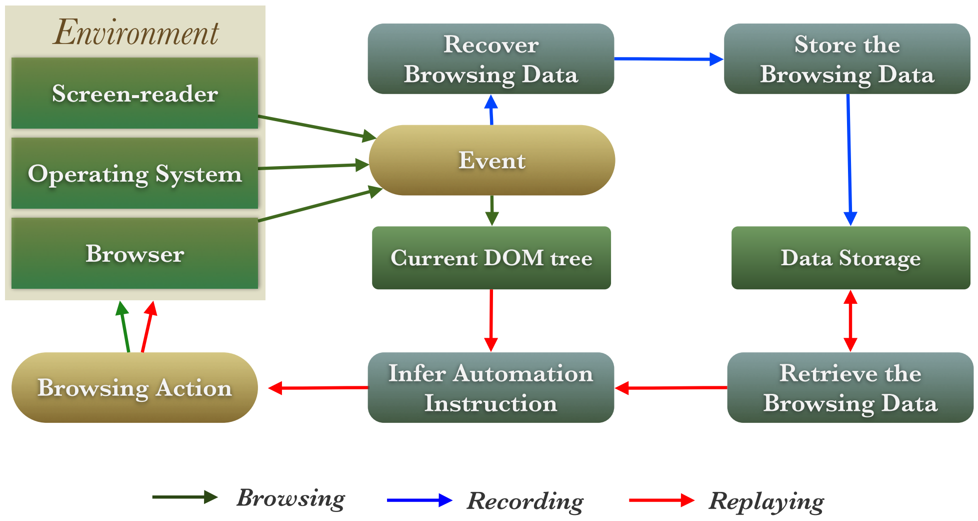

Figure 2.1: Overview of Web Automation Process

2.1.1 Generation of Automation Instructions

The two main approaches to learning or recording browsing data are handcrafting and *Programming by Demonstration* (PBD) [37].

The first approach to handcrafting requires writing a script to customize the behavior of the browser / screen reader, or hardcoding the automation instructions into a specific feature in the browser / screen reader. The second approach to handcrafting is employing a graphical user interface (UI) to specify automation instructions such as in Montoto atel. [58]. The handcrafting approach requires learning the tool(s) used to create the automation instructions: the syntax, the semantics, and the necessarily sophisticated UI (introducing additional accessibility and usability issues). The learning curve can be significant. For instance, handcrafting scripts for the JAWS screen reader [4] requires the user to learn to program in the script language and follow the 180+

page macro creation manual, while the instructions for Window-Eyes screen reader [7] scripting extend over 1000 pages.

The PBD approach enables an end-user to perform and record a macro. This is the approach taken by the majority of end-user automation tools. Automatic form filling in modern browsers is an example of PBD, hardcoded for a specific feature (filling textboxes and drop-down lists). To take this further, a recording made for one webpage can sometimes be dynamically adapted to be used for a similar task on a different webpage, or even a different website [30]. PBD can be further classified into three categories:

1. Most existing PBD approaches are *explicit*: the user needs to start and/or stop the macro recording process manually, which means that the user needs to manipulate the browser and the screen reader into a state when the recording can be started, execute the actions flawlessly, terminate the recording at an appropriate time and save it under a recognizable name.
2. With the *implicit* approach the macro is recorded completely automatically, which, in turn, brings forward the challenge of automatic detection of the beginning and the end of a macro.
3. Finally, the tools taking the *hybrid* approach require that the user specify only the start or the end of the recording, and the counterpart is inferred automatically. This design allows the web automation tool author to make some tradeoffs between the first two approaches, but does not address the fundamental issue: automatically or not, the macro needs to be managed.

2.1.2 Execution of Automation Instructions

Automation may be (a) initiated on-demand, e.g. with a shortcut, a predefined command, or a natural-language query, (b) recommended to the user by the system (followed by an explicit execution), (c) triggered with an environment event (e.g., a webpage-load), (d) following a user-specified schedule, or (e) executed from another macro.

Each of the above approaches has its disadvantages. For example, on-demand execution requires that the user know: which macros are available, how and where to find the right macro, how to put the browser in a state that will allow the macro to execute correctly (e.g., if the macro has to be started on a specific webpage), etc.

Recommendation-based systems need to provide useful and timely recommendations, and do it without interrupting the user's workflow, or creating a cognitive overload. System event-based triggers may initiate a macro at an inappropriate time and require additional user interaction.

2.2 Computer-Human Interaction Perspective

As can be seen from the discussion above, designing a usable web automation tool may involve making tradeoffs. However, when the web automation tool needs to be designed for non-visual use, the challenges become even more pronounced.

Listed below are observations made during studies of web automation tools with visually impaired users (conducted both by this and other [50, 58] authors). The common thread of most of those observations can be summarized in two words: *trust* and *cost*. The system's ability to gain and keep the user's trust in predictable and reliable functioning emerges as the cornerstone property of any usable web automation tool, but especially the one designed for non-visual use. At the same, time trustworthiness is a property that is hard to be maintained in an ever changing environment (the Web) that was never designed to be controlled or monitored by an interface agent. Moreover, even the most trustworthy system will not be practical unless it is less 'expensive' to use than a screen reader in terms of cognitive load, and time efficiency.

Of course, well-known principles of the human-interface design apply here as well and include (a) making the interface feel immediately familiar, (b) keeping things simple and most important functionality readily available, (c) encouraging, and enabling the users to discover functionality, (d) giving users sufficient control over the system's actions, and (e) overall user interface consistency. By following these principles, developers of web automation tools can significantly reduce the learning curve for novice users and make their interfaces more intuitive and usable.

2.2.1 Initiating Execution

The user needs to be able to create or procure useful automation instructions

As an obvious prerequisite to using a web automation tool, the user needs to be able to either create (using handcrafting, or PDB), or procure (from a shared database) useful automation instructions. The former approach is more conducive to the creation of highly personalized automation, but needs to be usable and accessible in its own right. For example, when using PBD to record a macro, the user may need to answer some or all of the following questions: (a) when to start recording a macro? (b) when to stop recording a macro? (c) was a mistake made during recording? and (d) how to store the recording to make it discoverable at a later time?

Procuring automation instructions from a shared database may be a simpler solution when personalization is not required, but it involves centralized database management and quality control (e.g., see [52]).

The user needs to know when and how the automation tool can be useful

It is unlikely that even a very sophisticated automation tool will be useful in all circumstances, and will be able to completely replace the screen reader and browser shortcuts. Therefore, the user needs to be able to switch easily between using the web automation tool and manipulating the screen reader/browser directly. Proactive (but not intrusive) notification of the user by the web automation tool about automation options available in the current context can reduce the number of unnecessary switches, and simplify discovery of useful automation.

The user needs to know what will happen

In order to automate a specific task, the user needs to find and execute the right automation instructions. Not knowing with certainty which buttons will be automatically pressed, and which form fields will be filled might be both disconcerting and dangerous (especially when money is involved). This is especially true with respect to irreversible operations (e.g., form submissions). The situation is exacerbated by the facts that (a) webpages change, and once correct automation instructions might no longer work correctly, and (b) the original automation instructions may contain an error. While all those considerations may prevent a sighted user from using a web automation tool, visually impaired users are even less likely to execute automation instructions without completely understanding their effect.

Whereas metadata can be used to provide clues about the intended function of a specific set of automation instructions, it is likely to be insufficient. A more reliable approach is to enable the user to easily review each automation instruction to be executed. This means that the automation instructions have to be presented in a way that is easy to understand without additional training, i.e. as close to human language as possible. Further, understanding the exact effect of executing automation instructions involves understanding the browsing context in which they will be executed (webpage and its state).

2.2.2 Managing Execution

The user needs to be able to make corrections to the automation instructions

One possible way to reduce the uncertainty regarding the possible effect of executing automation instructions is to allow the user to review and/or modify some or all of the instructions before, or while they are being executed. This includes changing the parametrization of the instructions (e.g., setting a value of a textbox to ‘John’ instead of ‘Tom’), executing only a portion of a predefined sequence of automation instructions, and reverting already executed actions (when possible). If the user is not able to intervene easily to stop the replaying, modify one or more steps, or completely deviate from the original recording, the smallest change in user preferences, or webpage design can render painstakingly crafted instructions completely useless.

The best approach to enabling corrections depends on the system design. If the automation tool is based on handcrafted scripts, then fixing them before execution might be as easy or as hard as writing them in the first place, but likely to be much harder during the execution phase. Enabling user corrections will be easier with a PBD-based system.

2.2.3 Completing Execution

The web automation tool needs to fail gracefully

As already mentioned, due to a number of reasons beyond the user’s control, there is always a possibility that the automation will fail. When and if this happens, the automation tool needs to (a) detect failure and (b) fail gracefully. For example, if one automation instruction out of a sequence fails to execute, the system should be able to estimate if the best course of action is (a) to terminate the execution and notify the user about it, or (b) ignore the failed action as inessential, or (c) if possible, automatically take corrective action. For example, in a case when a webpage is missing a form that needs to be submitted, the remaining actions in a sequence are likely to be impossible to execute, and the system should terminate. On the other hand, if the system cannot find a specific textbox to fill, but there are other textboxes on the webpage, the system may try to determine which textbox to fill instead.

The user needs know what has happened

Similarly, the user needs sufficient information about the effect of executing the instructions, the information about which can be provided either in a form of step-by-step feedback, or as a summary provided at the very end. In order to allow the user to manually recover from those mistakes, the feedback has to include information about a possible failure of execution, and the nature of the failure. However, even if the instructions were executed perfectly, the user still needs to understand the current browsing context: what webpage is now open, where the screen reader's virtual cursor is, what the form field values are, etc.

2.2.4 Minimizing End-to-End Cost

The web automation tool needs to minimize cognitive load for the user

Visually impaired users of screen readers are burdened with much greater cognitive load than sighted users. This stems from a number of factors: (a) the need to listen to a lot of unnecessary information before finding important information, (b) missing important information that is inaccessible via screen reader (for example, webpage color cues or layout, custom JavaScript based controls without ARIA annotations, etc.), (c) the need to either use a very basic set of shortcuts (e.g., moving 'up' and 'down'), or plan and execute a more elaborate combination of steps (e.g., jump to the 'bottom' of the webpage, then jump to the third button from the 'bottom', then go 'down' step by step to find a specific textbox), (d) the need to remember screen reader's virtual cursor position on the webpage, and relative position of other important parts of the webpage, (e) the need to remember the current state of the webpage, (e.g., the values of all important fields in a form before submitting it), and (f) the need to remember the changes applied to the webpage, and how to find the modified form fields and pressed buttons, to review and/or undo those changes.

In this context any additional piece of information related to the web automation tool that the user needs to remember may be a burden. This includes using the tool satisfying all the needs outlined above, and even switching between the web automation tool and the screen reader (the importance of this specific cost will be particularly obvious in the experiment described in Section 3.4). It is essential that the costs of using web automation be lower than the cost of the same task executed using a screen reader.

The web automation tool needs to be more time-efficient than using shortcuts

Screen-readers and browsers have hundreds of shortcuts that attempt to provide a solution to every possible need of the user. Of course, as mentioned before, this is an inherently inefficient, time-consuming mode of interaction. Moreover, the efficiency of using shortcuts is greatly dependent on the level of the user's expertise in shortcuts, in non-visual navigation in general, and in the particular webpages being visited (memory of 'where things usually are' on the webpage, or of which combination of shortcuts can help find some specific part of the webpage can be very useful when the webpage cannot be ‘glanced’ at). All those factors make up the ‘cost’ of using shortcuts.

Similarly, any web automation method has a ’cost’, which includes not only the cost of using the method itself (e.g., press a shortcut to run a macro), but also the overhead of preparing the automation instructions, discovering and initiating the right automation instructions at the right time, verifying the results of the automation, correcting the automated actions, recovering from mistakes, etc. Once again, the end-to-end cost of using web automation has to be lower than that of using shortcuts for the same task.

2.3 Technical Perspective

Listed below are the technical complexities of developing a practical, accessible web automation tool that accounts for the varios computer-human interaction constraints discussed in the previous section. Here, the operative word and the fundamental reason behind most of complexity is *uncertainty*. This includes the uncertainty of user’s intentions, the uncertainty of reaction of a webpage to automation, and the uncertainty of the meaning of events emitted by the environment. To a large extent, those constraints stem from the already mentioned fact that the web has never been designed to be controlled or monitored by an interface agent. Nonetheless, it is in the presence of those constraints that the system needs to be able to earn the user’s trust.

2.3.1 Classification of Environment Events

While specifying a macro using PBD, or handcrafting a macro using a graphical interface, every user browsing action may trigger *multiple* events in the environment, including transitively: the browser, e.g., setting form fields; the screen reader, e.g., the virtual cursor movement; and/or the operating system, e.g., keyboard presses (see Figure 2.1). While some of those events can be used to generate browsing data, others can be ignored as “noise”. The choice of which events should be considered noise depends on the abstraction level chosen by the tool designer. For instance, should we record that a shortcut has been pressed, or that a JavaScript function call was made (triggered by the aforementioned shortcut), or that the DOM tree was modified (by the aforementioned JavaScript call), or that the virtual cursor changed its position (as a result of the aforementioned DOM modification)? Once the abstraction level has been chosen, the noise can be avoided if the tool can ignore (a) events that are triggered by the recorded events, (b) events that trigger the recorded events, and (c) events that are unintentional, or unnecessary (such as a step-by-step exploration of the webpage in order to find the “submit” button).

Identification of semantically equivalent, but otherwise different events is another important contributing factor to the reliability of automation. First, two different actions may have the same semantics, i.e. lead to the same result, and the same result may not always be achieved by triggering the same events: the website author may change the events triggered by an action. For example, the user may submit a form by pressing an “enter” key inside a form field, or by pressing it over a “submit” button. Second, specific browsing actions may change their semantics (lead to different results), or implementation (same results will be achieved by triggering different events), e.g. the website author may replace the standard HTML “submit” button with a custom control that submits the form using JavaScript. Unfortunately, this problem cannot be solved in general: there is no way to infer *reliably*, for example, that two different JavaScript functions lead to the same result, or that a JavaScript function called twice will always lead to the same result.

2.3.2 Scarcity of Useful Browsing Data

Cold start is a problem of insufficient up-to-date information needed to automate actions on a specific webpage. Either the handcrafted macro has not been designed for this website, or the user has never recorded a macro using PBD (either because it was too ‘expensive’ or because the user did not realize the macro would be useful), or the macro data has decayed, i.e. the webpage changed too much and it is no longer possible to infer executable and sound automation instructions (this can happen as a consequence of server-side changes, new content added by website visitors, etc.)

The following three complementary solutions may reduce the effect of this problem. First, whenever possible, generate automation instructions without user involvement by observing user’s behavior, removing the need to handcraft a macro, or record it using PBD.

Second, in some scenarios the automation instructions can be generated by analyzing the webpage on the fly. The following is an incomplete list of possible pre-configured ‘automation scenarios’:

1. Edit the form field that is automatically assigned focus by the webpage
2. Edit the first form field in the most prominent (biggest) form
3. Edit the form field succeeding the one being currently edited

4. Submit the form being edited

5. Move the virtual cursor to the article title identified by a string most similar to the tab title or the label of the link that opened this page

6. Move the virtual cursor to the article title identifier by main content landmark (ARIA, HTML5 specifications)

7. Move the virtual cursor to the most visually prominent webpage elements

8. Move the virtual cursor to the first webpage element in a DOM subtree that was dynamically added after the first user interaction with the webpage since the initial DOM was loaded (such dynamic changes may be a result of silent error notifications, content skipped because it didnt have time to load, etc)

9. Move the virtual cursor to the first webpage element that is different from the previous webpage (useful to review the difference between webpages when the new webpage is not significantly different from the previous one)

Third, it is possible to collect data on one website, and reuse it on another website, as long as semantics of the data can be correctly inferred. This approach is explored, for example, in [30] as well as in the form filling feature employed by modern browsers, designed to fill basic details of forms with the user's personal information.

Fourth, impersonal information collected by observing one user (or a script handcrafted by one user) can be used to automate actions of another user. For example, if one user executed the steps necessary to check out a product from amazon.com, the automation tool can remove all personal data from the browsing data and use the resulting automation instructions for another user.

2.3.3 The Rise of Irrelevant Browsing Data

The second problem associated with data decay lies in the fact that, as the amount of old "broken" browsing data grows, it will interfere with the user looking up useful automation instructions. Moreover, this data will consume the disk space, and may be accidentally used to generate and execute "broken" automation instructions. In this situation, the user will have to either start managing manually (deleting, tagging, updating) "broken" browsing data, or have to stop generating (or procuring) it altogether. This manual work involves extra overhead: figuring out which automation instructions require management in the first place. For example, figuring out which automation instructions are “broken” by simply executing them and verifying the result is a significant effort, and may lead to undesirable, and possibly irreversible results (e.g. accidentally making a purchase). Management of macros, in particular, can quickly become a prohibitively ‘costly’ task for visually impaired users.

As discussed in the previous section, it is important to enable the user to update the automation instructions manually (both before and during the execution). At the same time, the management burden can be lessened with automatic detection and (a) permanent removal of outdated browsing data, (b) ‘hiding’ from the user contextually irrelevant browsing data, and (c) proactively updating, where possible, browsing data that can be ‘fixed’. For example, if the automation tool is using a machine learning algorithm, its model has to be designed to be easy to correct without retraining.

2.3.4 Inference of Automation Instructions

Inference and subsequent parametrization of automation instructions can be done at the time the browsing data is generated, or from the browsing data immediately preceding execution. The latter approach can benefit from (a) analysis of information not available at the time the browsing data was generated, e.g., the webpage on which the instructions will be executed and (b) a new user feedback. The type of inference and parametrization that is possible or required depends on the type and availability of the browsing data:

1. Generation of automation instruction (e.g., if a specific JavaScript event was recorded, infer that the user voiced the first paragraph of an article, and generate instructions to voice the first paragraph of the currently open article).
2. Parametrization of automation instruction (e.g., choosing the form field value to fill in).
3. Generation of a *sequence* of automation instructions (a macro) for a set or a list of instructions.
4. Identification of the specific webpage element on which to execute the automation instruction (if applicable).

2.3.5 Webpage Element Addressing

The logical structure of an HTML document is defined by the Document Object Model (DOM) [16], which provides a standard, language-independent and implementation-neutral application programming interface (API) for building, traversing, and modifying HTML documents. The DOM is structured as a tree, with each tree node corresponding to some webpage element, e.g., a button, a textbox, a table, a form, a group of text paragraphs, etc. The semantics of a webpage element can be inferred from the corresponding DOM node: its type, its relationship to other nodes, its position inside the tree, and the flat list of attributes (optionally) are stored within the node.

In order for a browsing action that acts upon some webpage element to be replayed correctly, the automation system needs to identify uniquely its target DOM node. The standard approach is to identify the target node using the XML Path (XPath) [12] query language. Unfortunately, (a) webpages change over time, either due to the author's server-side modifications, or contributions by its visitors (e.g., comments), or dynamic DOM mutations, and (b) different browsers may generate slightly a different DOM tree for the same webpage (e.g., Internet Explorer and Firefox). As a consequence, the node returned by an XPath query may no longer have the same semantics as the original node. The node with the "correct" semantics may be found elsewhere in the DOM tree, or not exist at all.

The fundamental problem of using XPath, or similar techniques (see Section 6.5 for more details) for reliable addressing, is that those approaches were designed to *query a DOM tree for element(s) at the specified address*, as opposed to *query for the DOM tree address of the specified element*.

To illustrate, consider the following two queries: (a) the person occupying 1600 Pennsylvania Ave., Washington, D.C., and (b) the President of the United States. While both 'addresses' *may* return the same result, their semantics are very different: if the President changes his/her residence, or the house numbering scheme changes, the first address will remain correct only if we care about who lives at the specified address, rather than who is the president, or who occupies the building located at the specified address at the time the query was formulated. If what we care about is locating the President, the only query that can potentially return the correct result is (b).

Consider a more specific example of an XPath query for "DOM node with attribute id='buy' ": //*[@id='buy'], referencing the webpage element "that will execute the current purchase order". Let us assume, that at the time the XPath query was generated, it returned a node corresponding to the given description. Let us also assume that the author of the webpage followed the HTML standard of not introducing multiple nodes with the same 'id' attribute (i.e. the XPath will return only one node). When the webpage author decides to change the “address” of the referenced webpage element, e.g., rename the ‘id’ attribute of the node, or assign it to a node serving a different function, the XPath will be “broken”. In general, no single XPath expression is immune to DOM mutations, and therefore XPath and similar query languages cannot be used for reliable webpage element addressing. Specifically, an address may “break” in one of the following ways:

1. Non-uniqueness: the XPath may be non-unique in the new DOM because it was not sufficiently specific to account for changes in the DOM tree.
2. False Positive: the XPath may be pointing to the wrong target node from the perspective of the action semantics (e.g., the author replaced a form “submit” button with the “resend” button).
3. False Negative: the XPath may not exist in the new DOM because it was not sufficiently general to account for changes in the DOM tree.

2.3.6 Detection of Action Completion

When replaying browsing actions on the user’s behalf, the automation system needs to determine when the previously automated browsing action has completed execution before executing the next one. This is important in case the effect caused by the preceding action might be a prerequisite for the following action, as well as in case the system needs to notify the user about completing execution of a sequence of automation instructions. There is no standard way to specify when the action has completed execution: a browsing action can trigger one or more environment events which in turn can trigger other events (possibly asynchronously). An action has completed when all the events it triggered have completed, which can only be detected at run-time.

A naive solution is to wait for some predefined amount of time from the moment a browsing action was initiated. A more sophisticated approach is taken by Montoto et al. [58] who suggest using JavaScript event listeners to detect which events were triggered by the browsing action (including transitively), and using callback functions as an event completion notification mechanism. This approach can potentially handle almost all possible cases, including XMLHTTPRequest, webpage redirects, iframes, and any JavaScript calls. Notable exceptions are (a) events that never finish, such as events triggered at specified time intervals and (b) data that did not finish loading into the HTML document, e.g., from MySql database. Those cases can be handled with a timeout.

2.3.7 Detection of Action Failure

Detection of action failure is required for the automation tool to implement properly the mechanism of graceful failure, mentioned in the previous section, as well as to inform the user that a failure has occurred, if necessary. Action may fail for a number of reasons. First, the action may not complete in a reasonable amount of time. This can cause the user to interfere with manually executed browsing actions (unless the user controls are blocked at the time of execution), which may in turn make the continuation of the automation process pointless, impossible, or both. This type of failure can be detected with a simple timeout. Second, the action may not be executed because the targeted webpage element address is "broken", which in some cases, can be detected with certainty (false negative or non-unique result), while the case of false positive result identification of failure can be harder. Third, even if the action is executed, it may not cause the desired effect, and/or lead to undesirable side effects. In some cases, this type of failure is the hardest to detect because it involves semantic understanding of the automation instructions. The failure may be a result of an automation tool error, webpage error, user error, and data decay. For example, if a user changed his login credentials on a website, the credentials recorded by the automation tool at an earlier time will no longer work, even if the automation tool filled all the necessary form fields and submitted the form.

In some cases, websites provide error messages containing useful information intended for user consumption (for example, notification of failure to submit a form, or a failure to choose a reasonable value in a date picker control). However, automatically detecting and interpreting such error messages may be a challenging task that requires dealing with the many non-standard approaches taken by web authors to bring such messages to the user's attention.

2.4 Conclusion

In this Chapter, I described the approaches to and the complexities of practical non-visual web automation. The three operative words that arise from the discussion are *trust*, *cost*, and *uncertainty*. The first is the user's ability to trust the system to do the right thing most of the time, and to recover from mistakes the rest of the time. The second is the need for the system to minimize the cost of using it: the cognitive load and the time. And the third is the uncertainty in which a web automation tool is required to operate, stemming from the need to interpret intentions of the user and the webpage author from clues, rather than from communication protocols designed with web automation tools in mind.

Chapter 3

Non-Visual Web Interface Agent

Automation Assistant [69] is an accessible, non-visual web automation tool. Automation Assistant can be described as an Interface Agent [47, 54], that is defined as a system that observes user interactions with the UI and interacts with the UI on the user's behalf when it is triggered explicitly or implicitly. An Agent may ask the user to confirm a set of actions before execution, validate the set of executed actions, or it may act completely autonomously. The UI of a *Web* Interface Agent can be either integrated into a webpage (via transcoding), the browser (via scripting), any other Web technology in use, or function independently. A Web Interface Agent needs to be *situated* in the Web environment, i.e. learn about the websites that could be visited, data sources that could be queried (the so called "Deep Web"), as well as be able to interact with one or more of technologies in the Web environment: websites, data sources, browsers, screen-readers, etc. One of the oldest (now deprecated) examples of Web Interface Agent technology is The Microsoft Agent [9].

In this chapter I am going describe a use scenario that can benefit from a system like Automation Assistant (Section 3.1), describe the workflow enabled by Automation Assistant (Section 3.2), discuss the merits of the proposed design (Section 3.3), and report the results of evaluating the system in a user study (Section 3.4).

3.1 Use Scenario

The following use scenario illustrates how Automation Assistant can be used to make web browsing a much better experience than it is possible with today's technologies.

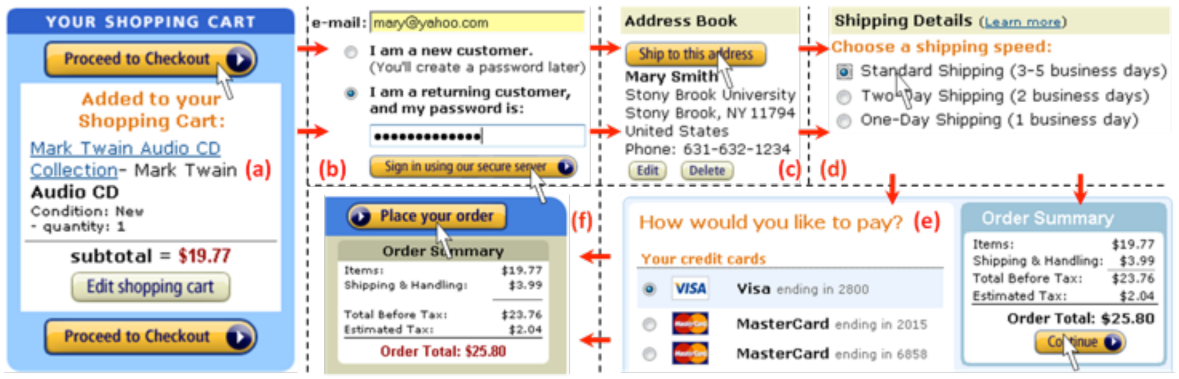

Figure 3.1: Making a Purchase on Amazon.com

Mary is legally blind, but her friends perceive her as computer-savvy because she uses email and chat applications, does shopping, pays bills, and manages her own finances online. Mary still remembers how tedious surfing used to be with screen-readers. But once she started using Assistant, that seems to be taking her always to the right place on the web page, doing things online has become much easier.

When Mary is shopping for a new audio book on Amazon.com, after adding it to her shopping cart, she presses a keyboard shortcut to ask Assistant to help her complete shopping. Assistant focuses on the Proceed to Checkout button (Figure 3.1-a) and Mary's screen reader reads "Proceed to Checkout, button". If there are more suggestions available, Mary can easily examine and choose among the available options. All she needs to do is press 'enter' to activate the button. And so, Mary goes through the checkout confirming every step.

Assistant helps Mary enter her e-mail and password and sign in (Figure 3.1-b). If she were already logged in, Assistant would skip over those steps and, instead, suggest that Mary select her regular shipping address (Figure 3.1-c) and then "shipping speed" (Figure 3.1-d). At any point, Mary can divert from the steps offered by Assistant and do whatever she wants, e.g., select faster shipping. Assistant helps Mary select the credit card (Figure 3.1-e), and, finally, click the "Place your order" button (Figure 3.1-f) the complete 6-step checkout sequence that used to take 10-15 minutes can now be done in just a few!

3.2 Automation Assistant Workflow

In designing Automation Assistant I started by considering what the ideal workflow for a visually impaired user of a web automation tool would be. Starting with the workflow instead of an algorithm was essential to avoid making the wrong tradeoffs before having a clear understanding of how the tool was to be used to be effective and practical.

To illustrate how screen reader users accomplish browsing tasks, let us consider the task of buying a product, and a simple scenario in which the last step remaining is to find and press the "checkout button" on an already open webpage. First, the user will have to either recollect, or guess, or discover through exploration of the webpage that pressing the button to checkout the product is indeed the correct next step. Knowing that, the user will adopt a strategy to find the button (Figure 3.2a).

The strategies employed by any given user to execute this single-action task will depend on the level of expertise of the user and his/her familiarity with the task and/or web page. Beginning users often listen to the content from the top of the page continuously until they hear the button label read to them (if they recognize that it is the button they need). They can also expedite the process by pressing the "Down" key to make the screen reader skip to the next line. This unfortunately is not much faster because one has to hear at least some of the content before realizing that it is not relevant. More advanced users may employ the search feature to find the button by searching for its label, assuming they know what the label is, and the label is a text string.

Expert users would employ element-specific navigation allowing them to jump back and forth among elements of certain HTML type: buttons, headings, edit fields, etc. Expecting to find an HTML button, they may press "B" to jump only among the buttons narrowing down their search space and reducing the amount of information they have to listen to. If they are familiar with the web page, they sometimes find quicker ways to get around, e.g., if the page has many buttons, instead of pressing "B" multiple times to get to the right button, the user may go to the end of the page and go in reverse order with "Shift+B", or press "H" to go to the next heading, and then "Shift+B" to go to the button that precedes the heading. Unfortunately, if the web page gets a slight make over, or if the user cannot remember the structure of the page, s/he has to fall back on the least efficient navigation with arrows. For instance, the 'check out' button may turn out to be an image-link with the label "Go to Cart", in which case neither pressing "B" nor searching for the label will help.

The key idea behind the design of Automation Assistant is that very few actions are applicable in a given browsing state/context, and even fewer are reasonable/useful, i.e. they can lead to a meaningful/ desirable result [68]. The state, or context in which the user currently operates, can be characterized by many parameters fitting one of the following three broad categories:

- Virtual context, such as a sequence of visited URLs, loaded cookies, recent user action his-

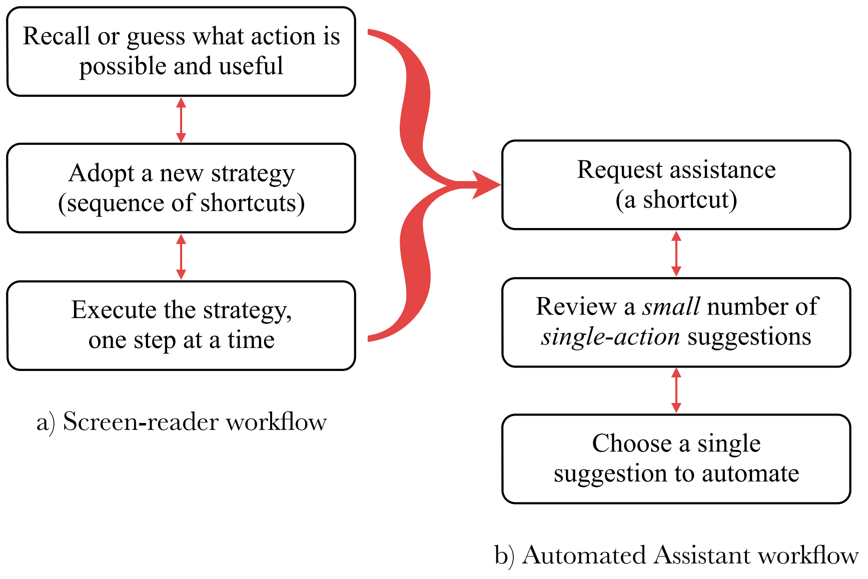
Recall or guess what action is possible and useful
Adopt a new strategy (sequence of shortcuts)
Execute the strategy, one step at a time
a) Screen-reader workflow
Request assistance (a shortcut)
Review a *small* number of *single-action* suggestions
Choose a single suggestion to automate
b) Automated Assistant workflow

Figure 3.2: Workflow of Executing a Single-Action Task

tory (e.g., filling out a form field, checking a checkbox, submitting a form, reading content, etc.), values of form fields in a webpage, webpage content, etc.

- Physical context, such as time of the day, day of the week, season, geographic coordinates, type of location (e.g., coffee shop, home, desert, etc.), speed, altitude, etc. Some of this information is easy to obtain (e.g., time), some is supported by the emerging standards (e.g., geolocation services in HTML5), some can be obtained using free web services (e.g., weather reports), and some can be obtained from specialized sensors (e.g., from the accelerometer or gyroscope in smartphones).
- User input, such as a (pseudo-) natural language string, a set of keywords, a hand / finger gesture, etc.

While browsing, the user can modify the state with user input or utilize predefined keyboard shortcuts to review a pseudo-natural language description of a suggested set of actions relevant in a particular browsing state; such set should be minimal (preferably a single action). The user can then select and confirm execution of a single action from that set, replacing the need to choose, execute, adapt, and restart (in case of a failure), different strategies. This is the approach taken by the Automation Assistant (Figure 3.2b), as explained below:

1. While browsing, the user issues a request for suggestions (e.g., a shortcut, a gesture). Automation Assistant will respond by providing a *small* number of *contextually relevant* sug-

gestions, each for a *single action* on a specific webpage element. The suggestions are provided by filtering out ('hiding') the irrelevant (not suggested) webpage elements.

2. The user examines the suggestions by moving the screen-reader's virtual cursor from one webpage element associated with a suggestion to another, while Assistant is providing voice feedback where appropriate. The virtual cursor could be moving between webpage elements either in topological order of the webpage (standard for screen-readers), or apply some other criteria (e.g., likelihood of the suggestions).

3. The user optionally examines the webpage elements around the suggestion, gaining better understanding of the context and meaning of the suggestion.

4. The user either (a) confirms the execution of exactly one of the suggestions, or (b) executes the suggestion 'manually', by using standard screen reader shortcuts (e.g., press 'enter' to follow a link) or (c) ignores the suggestions and continues browsing using the screen reader, starting from the current position of the virtual cursor.

5. The user optionally verifies that the action was executed correctly by listening to the feedback provided by Assistant, and/or examining the webpage.

6. The user optionally repeats the process again, this time (having changed the state of the webpage with some actions) receiving and examining a new set of suggestions.

The voice feedback provided to the user by Assistant is as similar as possible to what the user would hear from the screen reader (including taking the screen reader settings into account). Buttons, links, and radio buttons are voiced exactly like the screen reader would voice them. For example, a button 'submit' could be voiced as 'button submit'. No additional information is provided in those cases because the suggestion is implied by the type of the webpage element. Voicing of checkboxes, listboxes, comboboxes, and textboxes is augmented with a specific value of the suggestion. For example, the suggestion to flag an unchecked checkbox 'yes' would be voiced as 'yes checkbox unchecked suggested check', while a suggestion to fill a blank textbox labeled 'first name' with a value 'John' would be voiced as 'first name textbox blank suggested John'. Accepting a suggestion, or acting upon it using screen reader shortcuts will trigger the standard, minimal voice feedback expected from a screen reader, e.g., accepting a suggestion to modify a form field will voice its new value.

In contrast to the screen reader, Assistant allows the user to examine the most likely action-objects, e.g., a button that needs to be pressed or a form-field to be filled. When the user looks through the suggestions, there may only be the "checkout" and "keep shopping" buttons, because

Assistant will remember that those are the buttons the user usually presses after adding an item to cart. Furthermore, Assistant is likely to find the button even if its label and/or the position on the page changes. Thus, Assistant will allow even a novice user to be as efficient as an expert user, while also relieving the expert user of having to remember the web page structure.

3.3 Discussion

The previous section described the workflow of a user of Automation Assistant, and contrasted it with the typical workflow of a screen reader user. In this section I am going to discuss the advantages, and the disadvantages of Automation Assistant design with respect to the needs of a user of a non-visual web automation tool.

3.3.1 Initiating Execution

The prerequisite to using an automation tool is having access to useful and personalized automation instructions. Automation Assistant solves this problem by completely automating the process of collecting and storing browsing data. As an interface agent, Automation Assistant constantly monitors user activity and stores all relevant browsing data generated as a result of user's browsing actions, without any need to know when to start/stop a recording, if a macro is recorded correctly, and where to store it. Finding useful,contextually relevant automation instructions is also one shortcut away, removing the need to start/stop macro replaying.

A big emphasis in the workflow of the Automation Assistant is placed on providing the user with as much information as possible about the suggested actions, and doing it in a way that is both familiar and minimally intrusive. This is accomplished by (a) restricting the suggestions to a single action at a time, (b) providing voice feedback that is very similar to voice feedback of a screen reader, with only minimal, necessary modifications, and (c) allowing the user to easily switch between reviewing suggestions and exploring the 'neighborhood' (nearby webpage elements) of a specific suggestion to better understand its semantics.

Finally, the step-by-step workflow makes it rather uncomplicated to incorporate suggestions from preconfigured scenarios, such as those listed in Section 2.3.2.

3.3.2 Managing Execution

The step-by-step workflow enabled by the Automation Assistant facilitates the introduction of corrections to the automation instructions. The user can skip any number of suggested steps, continue browsing with a screen reader, and start using Automation Assistant when its suggestions become useful. Since the suggestions are not executed without a confirmation, the user can similarly stop using Automation Assistant, or modify a suggested action at any step. One could hypothetically use a macro to help guide the user step by step through the same exact transaction; however, macros are also a lot less flexible and cannot support some of the scenarios possible with Assistant: (a) presenting the user with a choice of several possible actions and (b) guiding the user starting from an arbitrary step of the sequence of automation instructions.

3.3.3 Completing Execution

Detection of action completion in Automation Assistant is less critical than with macro-based automation tools because automation instruction is never executed without the user's confirmation: the user is likely to wait a sufficient amount of time for the previous action to complete. Detection of action completion is also not necessary for updating the user with new suggestions because Automation Assistant can refresh the suggestions continuously, after every webpage modification, including dynamic DOM mutations (the Automation Assistant prototype is efficient enough to not introduce any perceivable latency into the UI).

Detecting a failure and failing gracefully with Automation Assistant is once again made easier by its step-by-step workflow. First, there is no multi-step automation process to terminate. Second, the likelihood that an action will fail is relatively small since (a) action on an element with a 'broken' address will not be suggested, and (b) the user will have an opportunity to make sure that the action is correct before executing it. Finally, the feedback provided to the user by the Automation Assistant after the executed action will keep the user up to date on the state of the webpage.

3.3.4 Minimizing End-to-End Cost

Automation Assistant minimizes the user's cognitive load in a number of ways: (a) by filtering out all the irrelevant content on the webpage, and helping the user focus on what is most important; (b) by reducing the need for complex browsing strategies and replacing them with a few simple shortcuts; (c) by removing the need to know if a macro is available for a given webpage, how to find it, if the macro has finished replaying and, if it did, what actually happened; (d) by replacing the need to figure out what each macro does, whether it is designed to do what needs to be done, and whether it will work - with a mode of execution were all those decisions are deferred for as long as possible, and are made for each individual automation step: i.e., figuring what each individual suggested action does immediately prior to deciding if it should be executed; and (e) by allowing the user to easily switch between the screen reader and the Automation Assistant, and to perform each task, and each step, with the tool best suited for that purpose.

At the same time, using Assistant, the user can quickly go through a long, complicated transaction spanning multiple steps and webpages: enter the user name and password to log in, select the shipping address, shipping method, billing information, and finally complete the purchase. Whenever Automation Assistant can provide useful suggestions, it is likely to be more time-efficient than the regular screen reader. However, if the suggestions provided are not useful, the time to explore them will be wasted. Further, a multi-step macro is likely to execute faster than when using Automation Assistant to complete the same task, even if the suggestions are always useful. However, the end-to-end execution time of a macro may be much greater than that of Automation Assistant, if we account for the time needed to find the right macro, later verify the results of the execution, and possibly backtrack to recover from mistakes.

3.4 Evaluation

The effectiveness of Assistant was tested in a user study with 19 visually impaired screen reader users performing tasks on webpages with and without the help of Assistant. We formulated the following hypotheses:

H1 Visually impaired users can complete tasks significantly faster when using Assistant than when using a standard screen reader.

H2 Usability of the screen reader is significantly higher when using Assistant than when using a standard screen reader.

3.4.1 User Interface

In consultation with web accessibility experts, I have designed several versions of Assistant's interface with two versions making the final cut in a pilot user study. Since even small details of the UI design have potential to influence results, we decided to test both versions of Assistant UI.

Assistant A. The S and Shift+S keys are used to move the screen reader's virtual cursor to respectively the next and the previous webpage element for which an action is suggested. This interface is based on the standard screen-reading interface for navigating among webpage elements of a particular type, e.g., B and Shift+B for buttons, A and Shift+A for links.

Assistant B. A single shortcut is used to toggle on/off the "suggestions" mode, in which the user can use standard screen reader shortcuts, but navigation is only allowed among the suggestions of Assistant, making the rest of the content "disappear". If the current screen reader's virtual cursor position is not associated with a suggestion (because the mode was just turned on or the suggestions changed), the cursor position is moved to the suggestion topologically following the current position. If there is no such suggestion, the user is taken to the suggestion topologically preceding the current position. Otherwise, "no suggestions" is voiced.

It is imporant to note that both Assistant A and B interfaces are designed to coexist in the same system with a screen reader. If none of the suggestions are useful for the user, he/she can use the standard screen reader shortcuts to accomplish their immediate goal, and then continue using Assistant. Both Assistant A and B interfaces can also coexist in the same system with each other.

For example, imagine that Assistant is used on a web page containing a large number of elements, of which the 5th one, a textbox, and the 32nd one, a button, are associated with suggestions; the user's current reading position is on the 10th element. With Assistant A, pressing the 'S' key will navigate to the button, while pressing 'Shift+S' will navigate to the textbox. With Assistant B, when the suggestion mode is turned on, the user will be taken to the button, and pressing the 'Up' key will take the user to the textbox.

3.4.2 Participants and Methodology

The 19 participants of our study consisted of 58% males, and were on average 54 ($\sigma = 12$) years old. The age group represents the typical age of a screen reader user, as many people lose sight later in life. The participants were White/Caucasian (42%), Black/African-American (26%), Latino/Hispanic (12%), Asian (5%), Indian (5%), Central-American (5%), and 5% declined to respond.

The participants rated their level of computer experience at “not comfortable” (0%), “mildly comfortable” (0%), “comfortable” (26%), “very comfortable” (58%), and “expert” (16%). About 90% of the participants used Internet Explorer as their primary browser, 5% used Firefox and another 5% used Safari. About 90% of the participants used JAWS as their primary screen reader, 5% used ZoomText and 5% VoiceOver. All the participants used JAWS before. The number of hours per week that participants regularly spent using the Web was 1-5h (32%), 6-10h (16%), 11-20h (26%), and more than 20h (26%).

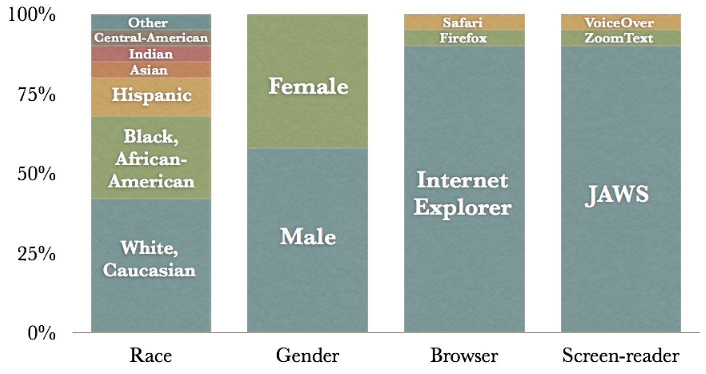

Figure 3.3: Study Participants

The study was conducted using Capti Web browsing application that has a screen-reading interface very similar to the standard screen-readers, with the Firefox browser and IVONA [15] Text-To-Speech (TTS) engine with the voice “Eric”, speech rate of 180 words per minute.

In this study, we asked the participants to perform 6 browsing tasks, each associated with a different website. The 6 chosen tasks were (1) registering a new website account (ebay.com) (2) searching for and booking a hotel (hilton.com) (3) website registration and product checkout (tigerdirect.com) (4) product checkout (amazon.com) (5) job search (monster.com) and (6) searching for and sharing a restaurant (yelp.com). Of the chosen websites 80% of the participants had prior experience with amazon.com and 20% with ebay.com. None of the participants used any of the other websites before.

Subjects were asked to perform the tasks using 3 different systems: the baseline screen reader without automation (**N-A**), Assistant A (**A-A**), and Assistant B (**A-B**). Each system was evaluated with 2 consecutive tasks. We counterbalanced the task order, the system order, and the task-to-system assignment.

Prior to performing the tasks, the participants were explained how to use the system they were about to evaluate, and given an opportunity to practice on a sample web site until they felt comfortable to proceed. Since all the participants had prior experience with the JAWS screen reader, the interface of our Capti screen reader, which provides the commonly used features and shortcuts of JAWS, was immediately familiar.

The profiles of the statistical models for Assistant were created in advance, i.e. for each user all initial suggestions contained the correct action that the user was expected to execute, or confirm. During the evaluation, Assistant updated the model and changed its suggestions based on the subject's behavior when s/he deviated from the initial suggestions. In practice, during the study, the participants ended up with at most 2 suggestions to choose from.

For each task, we measured the completion time, or recorded a time-out if the subject exceeded 10 min. (Table 3.1, Figure 3.4). The tasks that timed-out were not included in quantitative results computation. After each system evaluation (2 consecutive tasks), the participants answered the System Usability Scale (SUS) [36] questionnaire. After the completion of all tasks, the participants had to compare the 3 systems (Table 3.2).

3.4.3 Quantitative Results

Task success. All participants were able to complete all tasks with A-A and A-B within the given time constraint of 10 minutes. Four participants were not able to complete 1 task in the time allotted (a different task in each case, a total of 4 tasks) using N-A.

	A-A		*A-B*		*N-A*	
Task	μ	σ	μ	σ	μ	σ
eBay	187	87	**173**	33	448	158
Hilton	**69**	3	92	29	476	55
TigerDirect	214	67	**193**	49	526	89
Amazon	107	26	**101**	26	342	179
Monster	**94**	35	112	27	411	164
Yelp	174	36	**171**	48	422	127
Average	153	71	**142**	52	426	132

Table 3.1: User Interface Evaluation: Task Completion Time (sec.)

Task completion times are shown in Table 3.1 and Figure 3.4. One-way ANOVA test ($\alpha = 0.001$) shows statistically significant result ($p < 0.0001$) and Tukey's Multiple Comparison Test showed statistically significant difference between A-A and N-A, as well as between A-B and N-A ($p < 0.0001$). The one-tailed t-test ($\alpha = 0.001$) for all tasks showed that both A-A ($t = 8.9, df = 41$) and A-B ($t = 11, df = 45$) provide statistically significant ($p < 0.0001$) speed improvements when compared to N-A. This corroborates hypothesis **H1**.

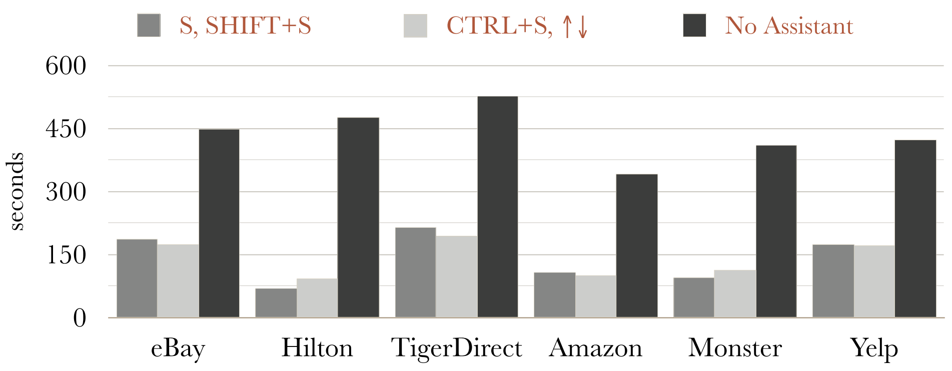

Figure 3.4: User Interface Evaluation: Task Completion Time (sec.)

Of the 6 websites used in the study, only amazon.com was very familiar to most of the participants - 80% used it on a regular basis (the only other familiar website was ebay.com, with 20%). This resulted in an insignificant improvement in the performance time when using a screen reader without automation (Table 3.1, task 4, column 5). However, even in this case both A-A and A-B provided statistically significant speedups when compared to N-A. This reinforces hypothesis **H1**.

3.4.4 Post-Completion Questionnaire

The combined scores for the SUS questionnaire are shown in the first two rows of Table 3.2. According to Bangor et. al. [27] the results for Assistant B (80), and the screen reader without Assistant (78) can be considered "good", while the results for Assistant A (88) can be considered "excellent". Overall, when asked if they would like to use Automation Assistant in the future the participants responded with an average grade of 4.47 on Likert scale (0.96 SD).

The ANOVA test between all three systems, as well as the one tailed paired t-test between the A-B and N-A ($p = 0.33, t = 0.44, df = 18$), showed no statistically significant difference. The one tailed paired t-test between the A-A and N-A showed that the improvement was statistically significant ($p = 0.0109 < 0.05, t = 2.5, df = 18$). This corroborates hypothesis **H2**.

After evaluating all three systems, the participants were asked to answer 10 questions comparing the three systems to each other (Table 3.2). The participants could answer by naming one of the evaluated systems or saying "none of the above". The results showed that the A-A was strongly preferred by most participants in all questions (shown in bold: the highest values for positive questions 1,3,5,7,9, and the lowest values for negative questions 2,4,6,8,10). This reinforces the results of the SUS questionnaires. It is notable that the participants scored A-A higher than A-B despite the fact that, in most cases, the participants completed tasks with A-B only marginally faster.

	Assistant A	*Assistant B*	*No Assistant*
SUS (A)	**88.16**	80.13	78.03
SUS (SD)	**11.24**	17.21	22.62
Which system would you prefer to use?	**84.2%**	15.8%	0%
Which system did you find the most complex?	**0%**	42.1%	31.6%
Which system did you find the easiest to use?	**78.9%**	15.8%	5.3%
Which system would need the most support from a technical person?	**0%**	31.6%	26.3%
Which system was the most well integrated?	**63.2%**	21.1%	15.8%
Which system was the least consistent?	**5.3%**	31.6%	15.8%
Which system do you imagine most people would learn to use most quickly?	**78.9%**	15.8%	5.3%
Which system did you find the most cumbersome to use?	**0%**	42.1%	21.1%
Which system did you feel the most confident using?	**78.9%**	10.5%	10.5%
Which system did you feel required you to learn the most before using it?	**0%**	42.1%	26.3%

[1] The answer "none of the above" is omitted for brevity
[2] The questions are based on SUS, reformulated for comparative evaluation

Table 3.2: User Interface Evaluation: Post-Completion Questionnaire

The high deviation values (σ_{SUS}) are, mostly likely, a result of counterbalancing the order of systems, and also because SUS questions were asked immediately after each system was evaluated (to ensure the experience was still fresh in the participants' minds). For example, if the subject evaluated A-A before N-A then the latter received much lower scores, reflecting participants' tendency to score each system in the context of prior experience. Likewise, when N-A was evaluated first, the participants assigned it very high scores because, in this case, they compared it to the screen reader they were used to the most: JAWS. A possible reason for this favorable comparison is that blind users strongly associate a screen reader with the synthesized voice, and we used IVONA, a high quality voice. In one specific example of contextual scoring, one of the participants, having already given N-A the highest scores possible, expressed her desire to score A-A higher than was permitted by the Likert scale.

3.4.5 Qualitative Results

Our observations show that, when performing tasks using N-A, the participants adopted one of three strategies: (a) moving through the webpage, top down, until finding the element they searched for, or, if not successful after some time, (b) jumping to the end of the webpage and sequentially moving bottom up, or (in about 20% of all cases) (c) using advanced shortcuts such for searching the webpage, or iterating through predefined element types (e.g., buttons, links). When successful, the last strategy provided a significant speed improvement; when not, the subject had to fall back on one of the two former strategies.

When using A-A or A-B, the participants initially showed signs of hesitation, often pausing before pressing the next shortcut. However, by the second task the participants were becoming substantially more confident. A common mistake made by the participants when using the A-B was to try to use the “up” and “down” keys to navigate away from the suggested element (this is impossible by design, since those shortcuts only navigate between suggestions when the suggestion mode is on). When using A-A, the participants sometimes used those keys to become familiar with the neighborhood of the suggested element before proceeding with the task.

A number of users explicitly stated that they preferred to remain always in the same mode (i.e. not to have a special suggestion mode), because this meant they did not need to learn subtle differences between the way the mode behaved. In other words, and introduction of multiple states may be increasing cognitive load, and may be the main reason for significantly higher usability scores of the A-A. At the same time, A-B may still be a better choice for power-users, because after a suggestion is executed, the A-B moves the cursor to the next suggestion, allowing the user to press fewer shortcuts. Also, in case of multiple suggestions, the user could use element-specific navigation shortcuts in the suggestion mode to iterate through suggestions of a particular type. For example, pressing B in the suggestion mode will iterate only through suggested buttons.

The participants made a large number of suggestions and provided ideas for improving Assistant interface. First, it quickly became apparent that suggesting a value for a form field that is already set to that value is redundant, and should be avoided.

Second, when suggesting to change the state of a checkbox, voicing the suggested value after the current value may be confusing because, in this case, users are used to listening to the last spoken word, to determine the checkbox status (the same, however, was not confusing for other form fields).

Third, radio buttons belonging to the same group differ from other form fields in that they are logically mutually exclusive: only a single radio button can be ‘checked’ at any time. At the same time, each radio button is a separate webpage element that can be suggested by Assistant, creating two problems: (a) Assistant may suggest more than one radio button from the same group, and (b) suggesting a specific radio button provides no easy facility to switch between radio buttons in the same group, using Assistant’s shortcuts (using screen reader shortcuts is still possible however). Therefore, some participants suggested that the radio button group be treated as a single webpage element (analogous to a listbox with several possible options).

Finally, it is likely that the first suggestion will always be ‘topologically’ below the current position, and therefore pressing an S for the first time will take the user to the fist suggestion. However, this assumption may not always hold. As a consequence, the user may either get used to always checking for a suggestion ‘above’ the current position, or risk missing out on useful suggestions. One possible solution is to make sure that the first press on the S key will always take the virtual cursor to the top-most suggestion first (and a subsequent press on Shift + S will take the virtual cursor back to its original position).

3.4.6 The Need for Web Automation

In addition to evaluating Automation Assistant, the participants were also asked a number of questions about their attitude towards using web automation tools in general (Table 3.3).

There are several observations that can be made about the answers. First, the participants were definitely interested in using web automation on regular computers (4.74), but much less so on mobile devices (3.68). High SD for the latter (2.0) question shows that there was substantial disagreement between the participants on this issue. This is possibly explained by the fact that not all people are interested in using mobile devices for browsing in the first place.

Executing more than a single automation instruction at a time, as well as executing automation on new websites was also a welcome idea. It is the author’s opinion however, that neither of those two features will be used before a user can build up his or her trust in the system not to fail in a step-by-step mode of operation. Moreover, it will be easy to loose the user’s trust by failing to execute actions without user supervision - an even harder technical challenge than not failing in a step-by-step mode of operation.

The participants also expressed strong interest in sharing their browsing actions with others, but did not feel as strongly about the targeted (and therefore inherently more private) mode of sharing.

3.4.7 Testimonials

We received no negative comments from the participants about the evaluated user interface. Below are a few quotations that exemplify participants’ overall reaction:

	Average	*SD*
I wish I could have a cleaner view of the web page as I am doing transactions	4.32	1.29
I wish I could automate browsing on regular computers	4.74	0.56
I wish I could automate browsing on mobile devices	3.68	2.0
I wish I could do the entire transaction automatically in one go, as opposed to doing them step by step	4.32	1.0
I wish I could also automate websites that I never visited before	4.21	1.03
With the understanding that Assistant learns to automate from your browsing actions, I dont mind sharing the learned automation with others, as long as it doesnt include my personal information and if I can benefit from automation learned from other people	4.84	0.37
I wish I could choose with whom to share the learned automation	3.47	1.71
I wish Automation Assistant remembered my passwords, as long as they are encrypted and saved only on my computer	4.26	1.15
I often find myself overwhelmed when I do online transactions such as shopping and reservations	3.26	1.52

Table 3.3: The Need for Web Automation: User's Perspective (Likert Scale)

"It's like automatic bookmarking on steroids."

"If all what people were doing is browsing the web, this would be perfect."

"I haven't seen anything as unique as this program."

"With one tap it brings you right to where you want to go."

"I hope it is coming soon, many blind people could use it; especially when you go to college it can help students with research."

"Pretty innovative idea I am wondering why even sighted people wouldn't find this product useful."

3.5 Conclusion

The reason for devising and executing various browsing strategies when using a screen reader is that screen-readers do not provide a "magic shortcut" that would do exactly what the user wants. Automation Assistant's workflow enables just such a "magic shortcut" that is missing from the vast arsenal of shortcuts made available to the users by the browser and the screen reader. The user study described in this Chapter provides supporting evidence that Assistant's workflow is indeed more efficient at some tasks than the screen reader. The ability to easily switch between using the screen reader enables workflows that the user is accustomed to, and the workflow enabled by Assistant allows the user to choose the right tool for each problem, making integration of Assistant into user's everyday browsing a matter of choice, rather than a compromise.

Chapter 4

Practical Model-Based Web Automation

In this Chapter, I will describe the technical aspects of Automation Assistant. Section 4.1 introduces new terminology, Section 4.2 describes the predictive model and the inference algorithm used to learn automation instructions and infer new browsing action recommendations, Section 4.3 describes the method of addressing webpage elements used in Automation Assistant, while Section 4.4 reports on results of the method evaluation.

4.1 Technical Preliminaries

An *automation instruction* is defined as the instruction that is necessary to execute a specific browsing action programmatically on the user's behalf. We define 3 types of automation instructions: *Value Change* (change the value of an HTML form field such as textbox, radio button, checkbox, selection list, etc.), *Invocation* (following a link, pressing an on-screen button, except form submission), and *Form Submission* (separated for the purpose of combining multiple ways of form submission). Automation instructions are automatically inferred from observed events (e.g., a JavaScript onsubmit event can be used to generate a Form Submission instruction specifying which form needs to be submitted). An action may trigger different events at different times and, hence, may or may not always generate equivalent automation instructions. Similarly, different actions may result in generating equivalent automation instructions. In the remainder of this work, I will use browsing actions and automation instructions interchangeably when it will not lead to confusion in the context.

We define *history* as a sequence of automation instructions that appear in the order in which they were generated from events. An example of history is: <a Value Change for setting the "First Name" textbox to "John", followed by a Value Change for setting the "Last Name" textbox to "Doe", followed by a Form Submission>.

The *query* denotes the set of all suffixes of the browsing history. It should be observed that suffixes denote here recent browsing actions.

4.2 Model-Based Web Automation

4.2.1 Overview

The approach taken in Assistant [69] is based on the idea that if certain subsequence of the history matches the most recent browsing actions, then the action immediately following the matched subsequence can be predicted as the user's next action, and hence it is a possible candidate for suggestion. To illustrate, let

```
...ABCD...AECE...ACE...ABECF...
```

be a part of the history capturing prior actions, and let `ABC` be one of the suffixes in the current query. Each letter represents a single browsing action and the same letters are used for equivalent actions. Then, we can align `ABC` to every subsequence of the browsing history and predict the next most likely user action (denoted by the '`?`' symbol). In our example we obtain the following four local sequence alignments:

$\texttt{ABC}\underline{\texttt{D}}$ $\texttt{AEC}\underline{\texttt{E}}$ $\texttt{A-C}\underline{\texttt{E}}$ $\texttt{ABEC}\underline{\texttt{F}}$ `...`

```
ABC? ABC? ABC? AB-C?
```

First, the suffix may align to more than one subsequence in history. In some cases, different actions may follow each of the aligned subsequences (e.g., `D, E, F` above). This can be a result of the user executing the same task on a modified webpage, or of the user making different choices within the browsing task, or of the user switching to a different browsing task altogether. Second, the suffix may partially match a subsequence in the history (e.g., `AECE, ACE, ABECF`), in which case the alignments will have different *edit distances* (the minimal number of insertions, deletions, and replacements needed to modify one sequence into another). The larger the edit distance, the smaller is the likelihood that the alignment is useful for making a prediction. However, the ability to find partial matches is very important for the algorithm to be useful whenever the user deviates from his/her historic record.

The time interval between actions in history may have high variance, either due to periods of user inactivity, or because a subsequence of history was deleted (or never recorded). However, it is not obvious that accounting for differences in time intervals will significantly improve the performance of the statistical model, since user's previous actions may be indicative of his/her next action even if they are separated by a large time span. Similarly, a very short (or very typical) time span between two actions is not a guarantee that the user did not make an unpredictable 'switch' to a completely new browsing task. Preliminary exploration of time interval statistics is done in a longitudinal study presented in Chapter 5. However, an investigation of usefulness of a sophisticated handling of time intervals is beyond the scope of this work.

4.2.2 Model Construction

The basis of our web automation model (overviewed above) is the Smith-Waterman [74], a dynamic programming sequence alignment algorithm. Smith-Waterman builds a Sequence Alignment Table T of size $m+1$ columns by $n+1$ rows. Each row i, and column j in the Sequence Alignment Table corresponds to a single automation instruction q_i, and h_j respectively. The automation instructions in the rows and columns are generated from the user's browsing history. Column m and row n represent the most recent automation instruction. $T[i][j]$ denotes the score of the alignment of two sequences composed of automation instructions from row 0 to i and column 0 to j.

In building the Sequence Alignment Table, we are interested in the alignment of the query to each subsequence of history. Those alignments are represented by the cells in the bottom row of the Sequence Alignment Table. Given a table of $n+1$ rows, cell $T[n][j-1]$ scores the alignment of the query with the subsequence of history that ends with action h_{j-1}. We say that the subsequent action h_j is predicted by the alignment $j-1$ with the score of $T[n][j-1]$.

Consider the example in Figure 4.1, in which we align $V_1V_2S_1$ (recent actions) to $\ldots V_1V_2I_1I_2S_1V_3 \ldots$ (subsequence of history), where V are Value Change instructions, I are Invocation instructions, S are Form Submission instructions. The score modifier for a match or a mismatch of q_i, and h_j, is 1 and -1, respectively:

The positive scores in the bottom row indicate that there are three equally likely alignments, each scored 1: (a) rows V_1V_2 match to columns V_1V_2, row S_1 is "deleted", and the predicted action is I_1 (b) rows V_1V_2 match to columns V_1V_2, row S_1 is (mis)matched with column I_1, and the

	...	V_1	V_2	I_1	I_2	S_1	V_3
	0	0	0	0	0	0	0
V_1	0	1	0	0	0	0	0
V_2	0	0	2	0	0	0	0
S_1	0	0	**1**	**1**	0	**1**	0

Figure 4.1: Example of a Sequence Alignment Table

predicted action is I_2, and (c) row S_1 is matched to column S_1, and the predicted action is V_3.

4.2.3 Model Optimization

Our sequence alignment is based upon the following definition, drawn from the standard Smith-Waterman algorithm:

$$T_o[i][j] = \max \begin{cases} 0 & \text{case 1} \\ T_o[i-1][j-1] + S_o[i][j] & \text{case 2} \\ T_o[i-1][j] + S_o[i][j] & \text{case 3} \\ T_o[i][j-1] + S_o[i][j] & \text{case 4} \end{cases}$$

Let $\equiv$ denote a match of q_i and h_j, let $\equiv_{vc}$ denote a match of q_i and h_j such that both are Value Change, and have the same value, and let $\not\equiv$ denote a mismatch of q_i and h_j. As already mentioned, Smith-Waterman defines the score modifier, denoted by S_o, as 1 if $q_i \equiv h_j$ and -1 otherwise. We depart from Smith-Waterman by redefining the score modifier as follows:

$$S_o[i][j] = \begin{cases} 1 & \text{if } q_i \equiv h_j \\ 2 & \text{if } q_i \equiv_{vc} h_j \\ P_o[i][j] & \text{if } q_i \not\equiv h_j \end{cases}$$

where P_o is a progressive penality for a mismatch of two actions, and is defined as follows:

$$P_o[i][j] = \begin{cases} P_o[i-1][j-1] - 1 & \text{if case 2} \\ P_o[i-1][j] - 3 & \text{if case 3} \\ P_o[i][j-1] - 2 & \text{if case 4} \\ 0 & \text{otherwise} \end{cases}$$

Value Change represents a modification to the HTML form fields, and, in some use scenarios, the chosen value of a form field may hint at future user actions. This scenario is accounted for by setting $S_o = 2$ if $q_i \equiv_{vc} h_j$.

I introduce progressive penalty P_o into the Model for the following reason. The necessary length of the suffixes in the query is hard to predefine. Ideally, this length should be large enough to include all the actions that can influence the user's choice of the next action, but, at the same time, small enough to exclude the actions that are "irrelevant" when predicting the next action. The local sequence alignment approach taken in this work allows us to examine the mapping of all possible suffixes to all subsequences of the history simultaneously. For instance, the example in Figure 4.1 examines the suffixes S_1, V_2S_1, and $V_1V_2S_1$; the length of positively scored alignments is 3 for cases (a) and (b) and 1 for case (c). However, this implies that some of the suffixes might be too long (contain "irrelevant" actions). We deal with this by penalizing the mismatch of two actions on a progressive scale, under the assumption that actions become "irrelevant" when they are followed by too many mismatches.

Finally, we want to prioritize case 2 over case 4, and case 4 over case 3. Therefore, the score modifier for aligning mismatching q_i and h_j (case 2) is smaller than the modifier for inserting an item from history (case 4), which is in turn smaller than the modifier for deleting an item from the query (case 3).

4.2.4 Generation of the Prediction List

As was discussed above, there may be multiple local sequence alignments between the query and the history. Each such alignment can result in a potential prediction. Let the *prediction list* be defined as a list of predicted browsing actions, or, equivalently, automation instructions, ordered by likelihood. The process of inferring a new prediction list is illustrated in Figure 4.2.

The model is updated on the fly, every time a new environment event is logged. First, unless the event can be ignored (as discussed in Section 4.2.5), a new automation instruction is inferred. Second, if Value Change is generated, and it has already been executed on the same page since the last page load, then its value is assigned to the first occurrence of the instruction, and the new occurrence is discarded. Third, the model is incrementally updated with the new automation instruction by generating a new column and a new row of alignment scores in the Sequence Alignment Table.

It should be noted that the model is updated incrementally: the new alignment of the updated query and history is computed by generating a single table row and column (instead of recomputing

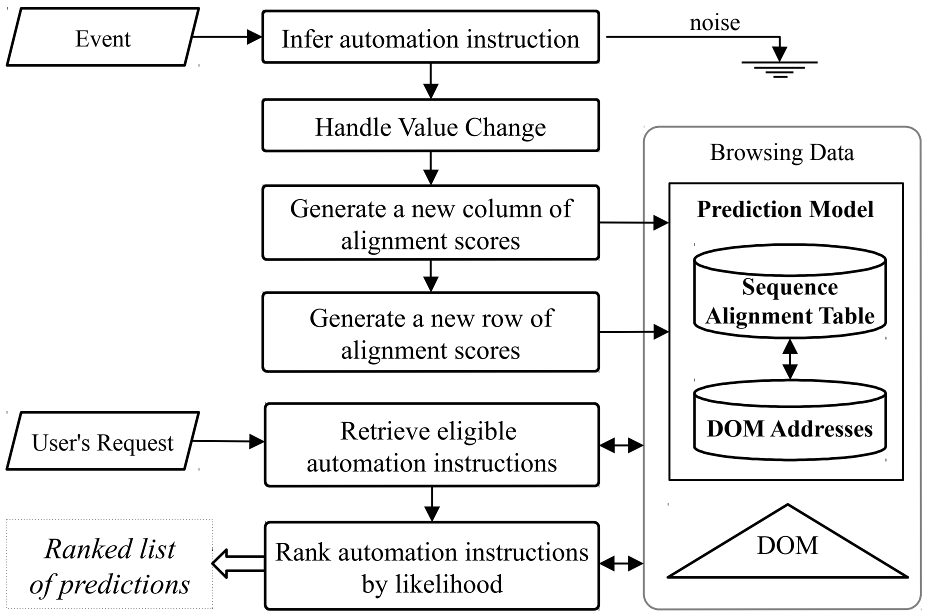
Event
Infer automation instruction
noise
Handle Value Change
Generate a new column of alignment scores
Generate a new row of alignment scores
User's Request
Retrieve eligible automation instructions
Ranked list of predictions
Rank automation instructions by likelihood
Browsing Data
Prediction Model
Sequence Alignment Table
DOM Addresses
DOM

Figure 4.2: Process Flow for Predicting User Actions

the Sequence Alignment Table from scratch). This is possible because the Sequence Alignment Table that needs to be computed at step i is a subset of the Sequence Alignment Table that needs to be computed at step $i + 1$. Moreover, adding a new row (column), only involves examining the row above (column to the left), and the generation of prediction list only involves the examination of the bottom row. Therefore, only the bottom row and the rightmost column of the Sequence Alignment Table need to be stored (see [60] for further details).

The algorithm generating the prediction list (or, equivalently in this context, automation instructions) is executed upon user's request for suggestions (for example, a shortcut). First, each prediction is evaluated for eligibility to be scored. A prediction is considered non-eligible and is discarded if at least one of the following is true: (a) the targeted webpage element (visual primitive of a webpage) does not exist, (b) it is hidden, (c) it is a disabled or a read-only form field, or (d) the action has been executed since the last page load. For instance, actions I_1 and I_2 in Figure 4.1 would be discarded if the Form Submission action S_1 triggered a new page load. The hidden, disabled, and read-only form fields are discarded because their value is usually assigned by the webpage itself, as a function of user's interaction with the enabled webpage elements, or factors beyond user's control (e.g., the date).

The eligible predictions are scored as follows. A sliding window of size k is used to collect k top scored, *unique* predictions, i.e., the sliding window always contains k most likely predictions so far. Since predictions need to be unique, each prediction's score is defined as the maximum of scores assigned to it by different alignments. If the scores of two different predictions are equivalent, the prediction with the score provided by the more recent (larger column j index) alignment is ranked higher (discussed in Section 4.2.5). For instance, the ranking order in Figure 4.1 would be V_3, I_2, I_1. Furthermore, by ignoring the *number* of predictions for each action and tie breaking, we aim to minimize the effects of data decay.

The time complexity for updating the model and recomputing the prediction list after each action is $O(m \log k + n)$, where k is the size of the sliding window, $O(m)$ is the time to add a new row, $O(n)$ is the time to add a new column, and $O(m \log k)$ is the time to generate a new prediction list. However, since k is a small constant - providing the user with too many suggestions would be counterproductive - the complexity is $O(m + n)$. The space complexity is also $O(m + n)$.

4.2.5 Generation of Automation Instructions

Automation instructions are generated from environment events. The mapping between events and instructions is:

- Value Change: triggered by JavaScript *onchange*, or *onreset* events: change of value of an HTML form field such as textbox, radio button, checkbox, selection list, etc.
- Invocation: triggered by (a) JavaScript *onclick* event: a mouse click (except on a form 'submit' button), or (b) JavaScript *keypress* event: a press of the 'enter' key (except in a form field, or a form 'submit' button).
- Form Submission: triggered by a JavaScript *onsubmit* event.

We say that two automation instructions are equivalent if and only if they have the same type and:

- Both are an Invocation of a URI element (e.g., following a link), and refer to the same URI. Otherwise,
- Both refer to the same unique webpage element (see Section 4.3).

When comparing automation instructions, we need to consider the semantics (the actual effect) of the instructions, rather than the specific events from which the instructions were inferred. A mouse click and a press on the 'enter' key are assigned the same type because more often than not the semantics of a mouse click and a press on the ‘enter’ key is the same (for instance, clicking a link with a mouse, or, focusing on the link and then pressing ‘enter’). By not making any assumptions about the specific input device, we make it easier for the model to compare and identify equivalent instructions and ensure that the model created by observing keyboard events will be usable when the user switches to the mouse and vice versa. The mouse can be useful for people with low vision and it can enable sighted people to easily create models for people with vision impairments. This generalization will also enable transparent support for other input modalities (such as Braille devices, touch screens, or speech input) in the future.

A Value Change instruction is parametrized with any input in the corresponding form field, such as entering value into a textbox, (un)checking a checkbox, changing the selection in a combobox, etc. Value Change is recorded only when the user exits the form field, i.e. the value is final. The value of all the subsequent modifications of the same form field (i.e. if the user comes back to update the value) are stored in the original instruction, i.e. no new instruction is created. For instance, if the user (a) sets “First Name” field to “John”, then “Last Name” field to “Smith”, and then updated the “First Name” field to “Sam”, this generates a sequence of (a) Value Change ”Sam” for ”First Name”, followed by (b) Value Change “Smith” for “Last Name”.

Form submission is always related to a specific form being submitted, and therefore the semantics of submitting a form differs from Invocation. Different Invocation instructions that can be used to submit the same form include pressing ‘enter’ or clicking the mouse on the form ‘submit’ button (of which there can be more than one for the same form, including the outside of the form element, as per HTML5 specification), pressing ‘enter’ inside a form field, and interacting with a custom form submission mechanism. We want to make sure that the semantically equivalent but otherwise different ways of submitting the same form are considered equal.

However, the general case of comparing instructions by their semantics is not reliable (except form submission, invoking a URI, and possibly other special cases). If a webpage has two controls that cause the same effect (e.g., two buttons that open the same shopping cart) by running custom JavaScript, the corresponding automation instructions will not be equivalent. This is mitigated by the fact that if the user chooses to use both controls interchangeably, the system will learn both variations and will provide useful suggestions regardless of user’s choice.

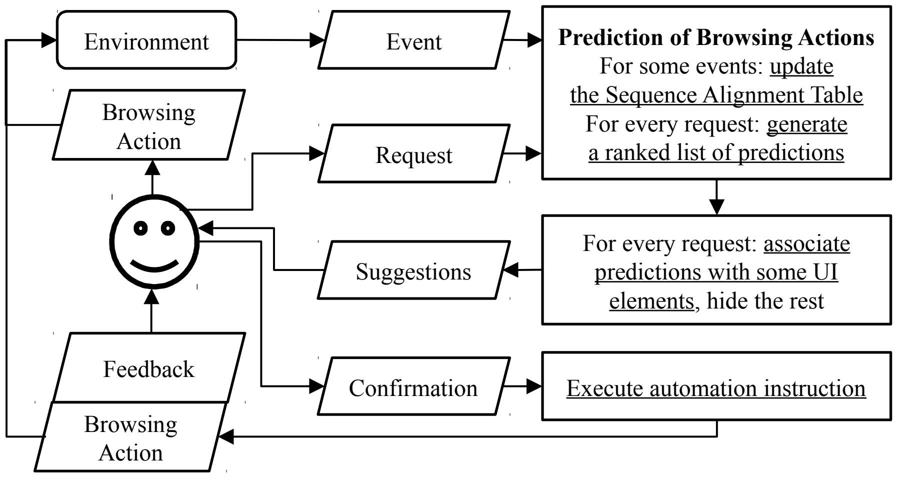
Environment
Event
Prediction of Browsing Actions
For some events: update the Sequence Alignment Table
For every request: generate a ranked list of predictions
Browsing Action
Request
Suggestions
For every request: associate predictions with some UI elements, hide the rest
Feedback
Confirmation
Execute automation instruction
Browsing Action

Figure 4.3: Web Automation Assistant Architecture

4.2.6 Putting it All Together

Chapter 3 provided a detailed description of the user workflow that Automation Assistant was designed to enable. Here we return to it once again, but this time from the Automation Assistant perspective (Figure 4.3):

1. Assistant constantly updates the Sequence Alignment Table by listening to events triggered by browsing actions. Some of those events are ignored, while others turned into automation instructions and added into the SAT.

2. In response to user's request for new suggestions (shortcut) Assistant generates a new, short, fixed-length ranked list of automation instructions representing predictions of the next user action.

3. The predictions are exposed in the UI in a form of suggestions associated with the webpage elements they act upon. If the webpage address of the prediction is 'broken', the corresponding suggestion will not be provided.

4. Once the user has examined some or all the suggestions, a confirmation for one of the suggestions may be issued, triggering Assistant to execute the corresponding automation instruction. As a consequence the state of the webpage changes (or a new webpage loads), and Assistant updates the SAT, ready to produce the next set of suggestions.

4.3 Resilient Addressing

The complexity of addressing a DOM element in a way that would be resilient to DOM changes is described in Section 2.3.5, and prior work on the subject can be found in Section 6.5. In Assistant the issue of resilient addressing is resolved in a way different from the existing approaches. A quantitative comparison with existing approaches was not performed, and remains a work in progress. That said, Assistant's choice of addressing scheme has proven to be sufficiently resilient in practice to make it possible to conduct a 6-week longitudinal evaluation of Assistant (Section 5). In this section I outline the addressing scheme taken in Assistant.

Each address in Assistant is represented by an *address descriptor*. Descriptor is a set of 'nodes', where each node is described by a set of it's attributes. The nodes are chosen by applying one or more *relative* XPaths to the node who's address we want to store in the Descriptor (function f). Those XPaths include, but are not limited to (a) the node's left-most sibling, (b) the node's right-most sibling, (c) a sequence of nodes on the path to the root of the tree, (d) the left-most sibling of the parent, (e) the right-most sibling of the parent, (f) the label of the node (same as sloppy programming), etc. When an automation instruction is initially stored in the SAT, it is associated with the Descriptor of the referenced page element stored in a separate shared addresses data structure. Automation instructions that refer to the same webpage element also refer to the same Descriptor object, providing for an efficient storage and comparison of SAT entires.

The Descriptor is used to locate the addressed node in a DOM tree in the following way. Given some descriptor $d_0 = f(n, t_0)$, generated from node n in a DOM tree t_0, and a new version of a DOM tree t_1, we need to choose a node m in t_1 such that the descriptor $d_1 = f(m, t_1)$ is the best possible match to d_0 of all descriptors generated from t_1. Two descriptors completely match if and only if the sets of nodes in each (and the XPaths with which they were generated) are identical. Matches of less than 80% of nodes are discarded to prevent false positive matches. Once d_1 is chosen, it replaces d_0 in the shared address storage, allowing Assistant to "track" changes of the webpage with an on-the-fly update of the Descriptor.

4.4 Evaluation

In this section, we describe the evaluation of the model's prediction accuracy.

4.4.1 Methodology

The data for model evaluation was collected by observing 8 Firefox users performing 6 different browsing tasks, each associated with a different website. The browsing tasks were chosen to be comparable in difficulty and to be representative use cases for people with vision impairments (according to our accessibility consultants). The 6 chosen tasks were: (1) registering a new website account (ebay.com); (2) searching for and booking a hotel (hilton.com); (3) website registration and product checkout (tigerdirect.com); (4) product checkout (amazon.com); (5) job search (monster.com); and (6) searching for a restaurant and sharing the information with friends (yelp.com).

The data was collected by sighted, rather than blind users, for pragmatic reasons - it takes a blind user on average 5-10 times longer to complete a single task compared to a sighted person, which makes data collection prohibitively expensive. However, as we confirmed in a pilot study, the model works the same way for sighted and blind users because they would have to take the same exact steps to complete the tasks. The actions predicted by the model are limited entirely by the functionality of a website; the actions are not specific to non-visual browsing (e.g., movement of the virtual cursor), and are input-modality independent (mouse and keyboard). Hence, the accuracy of the underlying model can be evaluated with sighted, as well as with blind users.

The users generated 5 "noisy" browsing sequences per website by performing a task on each website 5 times, each time varying the browsing actions as they would normally do, e.g., choosing a different shipping method while shopping, reviewing product specifications, changing the mind and choosing a different product, filling out form values in different order, re-entering (updating) data in fields, using different methods to submit a form, etc. The users were not constrained by a time limit and completed all the browsing tasks. The resulting dataset contained a total of 240 browsing sequences (30 per person) and 3074 browsing actions (65% Value Changes, 19% Invocations, and 16% Form Submissions).

The goal of the evaluation was to measure the ability of the model to predict a *gold-standard* (minimal) sequences of actions from a noisy browsing history. That is, we considered a scenario in which the user's intention was to complete the remainder of any given browsing task using Assistant with a minimal number of steps. A gold-standard sequence was independently handcrafted for each of the 6 websites; the resulting sequences contained a total of 51 actions, of which 72% were Value Changes, 17% were Invocations, and 11% were Form Submissions.

The evaluation was done in iterations, where in each iteration $j = 1...6$: 1) the model was updated with randomly selected noisy browsing sequences one by one (from the same website, and the same subject), 2) a new sequence of prediction lists ξ_j was generated, and 3) the accuracy of the prediction was evaluated by comparing the model prediction to the gold standard sequences. That is, on the first iteration, the model contained a single browsing sequence, on the second - two browsing sequences, and so on. The accuracy for each iteration j, denoted μ_j^ξ, is then averaged across all users and websites. The gold-standard sequence was not used to update the model.

Given a gold-standard sequence $h_i \in H$, the prediction list $L_i \in \xi$ was evaluated with the metric *precision at* k, denoted by P@k, and the *mean reciprocal rank* metric, denoted by MRR, of the correct prediction h_{i+1} (sliding window size is unbounded). MRR is defined as follows:

$$MRR(H, \xi) = \frac{1}{|H| - 1} \sum_{i=1}^{|H|-1} \psi_i$$

If $h_{i+1} \in L_i \in \xi$ then the utility function $\psi_i = 1/L_i(h_{i+1})$, and $\psi_i = 0$ otherwise; $L_i(h_{i+1})$ is the rank of h_{i+1} in L_i.

It is important to note that the predictions (in form of suggestions) might not be reviewed by the user in the order produced by the ranking algorithm. In a user study with people with vision impairments (Section 3.4), we presented the suggestions in the screen-reading order (preorder DOM traversal) to ease transitions between reviewing suggestions and regular browsing. However, MRR is useful for comparing prediction algorithms, and for producing a rough estimate of a threshold k below which the predictions can be discarded because they are unlikely to be correct. This threshold is the size of the sliding window used to generate the prediction list (see Section 3.4), and is also used for the P@k metric. The smaller is the value of k the more preferable it is for the user, and the less likely it is that a useful prediction will be part of the list made avaliable to the user. In our experiments we set $k = 5$, as a compromise between the two competing considerations, based on empirical obsevations of system results and on the pilot user study.

4.4.2 Results

Table 4.1 compares the accuracy of 3 methods for gold-standard sequence prediction: (a) $P(h_{j+1}|h_j)$: bigram frequency count, where each bigram represents two consecutive user actions h_j, h_{j+1}, i.e. predict action $j + 1$ based on a single most recent action j, (b) $T[i][j]$: the unmodified Smith-Waterman [74], and (c) $T_o[i][j]$: the optimized Smith-Waterman. $S(\cdot)$ denotes discarding of non-eligible predictions (as described in Section 4.2.4).

The results show that (i) looking back at more than one preceding user action, (ii) optimization

	P@k		MRR	
Algorithm	μ_1^ξ	μ_6^ξ	μ_1^ξ	μ_6^ξ
$P(h_{j+1}\|h_j)$	0.23	0.27	0.19	0.36
$S(P(h_{j+1}\|h_j))$	0.43	0.46	0.45	0.30
$T[i][j]$	0.36	0.27	0.17	0.15
$S(T[i][j])$	0.45	0.33	0.27	0.20
$T_o[i][j]$	0.52	0.54	0.35	0.41
$S(T_o[i][j])$	0.62	0.69	0.45	0.49

Table 4.1: Accuracy of Predicting the Gold-Standard Action Sequence from User-Generated Action Sequences

of the Smith-Waterman parameters, and (iii) skipping non-eligible actions all contribute towards better performance; although not shown here, each optimization was also tested independently. Of course, the disadvantages of making predictions just based on a single most recent action will be erased if the user never deviates from his/her previous steps, which is a very uncommon scenario in web browsing.

4.5 Conclusion

The design of the algorithm described in this section attempts to address the needs of the workflow described in Section 3, as well as the issues outlined in Section 2. The fact that the model can be updated continuously (after every user action), and without user supervision relieves the user from the responsibility to manage any aspect of recording the browsing data. At the same time, the incremental update is efficient enough to make continuous update practical, without an increase in the latency of the system response, which is rather important for screen-reader users. The automatic management of the browsing data, as well as the fact that the model is not ‘compressed’, makes it easy to introduce heuristics for discarding outdated browsing data by simply removing corresponding ‘rows’ and ‘columns’ from the table. Finally, this section describes approaches to classification of environment events and resilient DOM addressing: the two issues that need to be addressed in any web automation tool.

Chapter 5

Longitudinal Study

This study was conducted in order to explore the long term practicality of Automation Assistant and establish trends in user behavior that go beyond the first few hours of usage. A longitudinal study helps reduce the effects of system learnability, system feature discoverability, and user expertise, giving the system's long term usability a more prominent effect on the results. In this Chapter, I will describe the methodology of the study (Section 5.1) and discuss the study results (Section 5.2).

5.1 Methodology

The study was conducted with a single, blind, male participant. The participant's level of computer experience is "expert", selected out of the possible choice of computer expertise: "not comfortable", "mildly comfortable", "comfortable", "very comfortable", and "expert".

The study was conducted using Capti screen reader integrated with the Firefox web browser, running on Mac OS X operating system. The Text-To-Speech (TTS) engine used was the standard OS X speech engine, configured to use the voice 'Alex'. The speech rate was not fixed, and the participant was allowed to change it at any time. This choice was made to accommodate the participant's workflow that required frequent manual adjustments of speech rate.

The study lasted for a total of 6 weeks, of which the first 3 weeks the participant used Capti, and during the last 3 weeks the participant was allowed to start using Automation Assistant features integrated into Capti. The participant was instructed to use Capti/Automation Assistant for daily browsing. In the end of the study, the participant reported the usage of Capti/Automation Assistant for about 80% of all browsing needs, with the other 20% dedicated to an iPhone device, and its integrated VoiceOver screen reader. The participant reported this as a typical usage pattern, dictated by the need to access the Web on the go.

Capti enables the user to browse the Web much like popular screen-readers JAWS and VoiceOver. Despite being less configurable and exposing fewer shortcuts than those screen-readers, Capti is a fully featured screen reader and is capable of handling complex state-of-the-art webpages, including ARIA annotations, dynamic DOM mutations, and HTML5 (see Appendices A and B for additional details).

When using Automation Assistant, the participant was given access to 4 additional shortcuts (Table 5.1) not available in the "standard" screen reader mode of Capti. The shortcuts UI were modeled after the Assistant A interface described in Section 3.4. The choice was made based on the results of the experiment described in that section, which showed that Assistant A is more usable than Assistant B. The differences between the UI used in this study and Assistant A are summarized in the following points:

1. In this study the shortcut S was replaced with Alt + Down Arrow, while the shortcut Shift + S was replaced with Alt + Up Arrow. This was done to allow the user to request new suggestions even when editing a textbox, or selecting an item from a listbox (pressing S while editing a textbox results in typing S; pressing S while editing a listbox results in jumping to an element beginning with the letter S).
2. A new shortcut, Alt + Space, was added to enable the participant to 'train' Assistant without actually having to interact with a webpage element. For example, pressing Alt + Space on a textbox that was pre-filled *by the browser* (and therefore not stored in the SAT), would actually add an automation instruction to the SAT, as if the user entered the value manually. Similarly, pressing Alt + Space on a button would add an automation instruction to press the button, without actually pressing it.

During the study we recorded a log of the participant's keystrokes. For each keystroke we recorded what internal system command it maps to (if any), e.g., a key stroke on the letter X while not editing a textbox will map to a specific internal command that can be described as 'go to the next checkbox'. In addition, for each keystroke we recorded the system time of its execution.

The model (Sequence Alignment Table) was continuously updated, including during the first 3 weeks of the study. This allowed the participant to start benefiting from Assistant from day one of the study's second phase. The type of automation instructions recorded into the SAT slightly differs from the ones described in Section 4.2.5, as the changes were made to improve system reliability:

1. Two types of automation instructions were recorded: Invocation and Value Change. The Form Submission instruction was merged with this Invocation instruction.
2. If a radio button was selected by the participant, and the SAT contained a record of another radio button from the same radio group having been selected since the last webpage load, then the old record is updated with the new radio button (instead of creating a new automation instruction).

Shortcut	Explanation
Alt + Down Arrow	Move the virtual cursor to the next suggestion
Alt + Up Arrow	Move the virtual cursor to the previous suggestion
Alt + Space	Accept the current value in the form field and refresh suggestions
Shift + Space	Accept (execute) a suggestion, and refresh suggestions

Table 5.1: Automation Assistant Shortcuts

5.2 Results

The participant visited 30 unique domains (117 non-unique webpage visits) during the first 3 weeks, and 28 unique domains (168 non-unique webpage visits) during the last 3 weeks of the study. In that time period, the participant made a total of 20019 keystrokes, and recorded a total of 809 automation instructions into the SAT (4% of the total number of keystrokes). Of the 809 unique SAT records 80.5% were Invocations, and 19.5% were Value Changes.

5.2.1 Requesting and Confirming Suggestions

Table 5.2 details the statistics collected during the period of the study. The table contains 2 columns: the absolute values and values normalized per non-unique webpage visit.

The three bottom rows in the table require a special mention. The row titled 'Accepted current value (Alt+Space)' shows an insignificant number of uses of the model 'training' mechanism, the reason for which was that the participant did not feel motivated to use the feature, being aware that while it might be useful in the long term, in the relatively short span of the studys last 3 weeks during which this feature was enabled, it will be of only limited benefit.

The row titled ‘Accepted suggestion’ refers to the number of suggestions accepted by the user, which includes the suggestions accepted by pressing Shift + Space, as well as those accepted by using the standard screen reader command: pressing ‘enter’ on buttons, links, checkboxes, and radio buttons after asking for a suggestion (‘enter’ must be pressed on the suggested webpage element). The participant reported that the ability to freely choose the method of interaction with the webpage Assistant’s shortcuts or the screen readers shortcuts, noticeably increased the system’s ease of use.

The last row, titled ‘Accepted suggestion (up to 2 steps)’ is measuring, in addition to the number of immediately accepted suggestions, the number of times the following scenario happened: (a) the user asks for a suggestion, (b) the user decides to explore the neighborhood of the suggested element, (c) the user moves away from the suggestion by no more than 2 steps, (d) the user returns to the suggested element and, finally, (e) accepts the suggestion. The documented increase in the number of suggestions accepted after a brief ‘search’ around the suggested element indicates that Automation Assistant has an impact on user’s browsing strategies that goes beyond the basic workflow of request/review/accept suggestion.

	Total	Per Webpage
Key presses (Capti)	8748	74.77
Key presses (Assistant)	11271	67.09
Invocations (Capti)	361	3.09
Invocations (Assistant)	291	1.73
Value changes (Capti)	69	0.59
Value changes (Assistant)	88	0.52
Requested suggestion (Alt+Up)	181	1.08
Requested suggestion (Alt+Down)	364	2.17
Accepted current value (Alt+Space)	3	0
Accepted suggestion	68	0.40
Accepted suggestion (up to 2 steps)	90	0.54

Table 5.2: Browsing with Capti and Automation Assistant: Suggestions

5.2.2 Time

Table 5.3 compares the time the participant spent while browsing with Capti, and Automation Assistant, in total and normalized per non-unique webpage visit.

The time spent browsing was computed by measuring and adding up time intervals between individual browsing actions. Time intervals were categorized into 3 non-mutually exclusive tiers: up to 200ms, up to 1 min. and up to 2 minutes, which was done because the size of the interval depends on several factors, and not all of those factors can be easily mitigated:

1. The time interval during which a webpage load event was detected is likely to be caused by waiting for the webpage to load. Those intervals were ignored.
2. The time interval can be extended because the user is waiting on dynamic DOM updates (i.e. the webpage is not reloading, by loosing responsiveness) that are caused by asynchronous server requests. The magnitude of this type of 'interference' with the browsing activity is hard to measure.
3. The size of the interval also depends on the amount of text that the user wants to listen to before moving on to the next webpage element / executing the next action.
4. Consequently, the size of the interval is also affected by the chosen speech rate, which is allowed to be changed on the fly by the participant, as already mentioned before.
5. The size of the interval between actions may also depend on factors unrelated to browsing, for instance, the user might decide to walk away from the computer, return some time later, and continue browsing.

	Total	Webpages	Per Webpage
Intervals of up to 200ms			
Time (Capti)	141465	103	1373.45
Time (Assistant)	204905	163	1257.09
Assistant, as % of Capti			**91.53%**
Intervals of up to 1 min.			
Time (Capti)	2032789	116	17524.04
Time (Assistant)	2848331	168	16954.35
Assistant, as % of Capti			96.75%
Intervals of up to 2 min.			
Time (Capti)	3136634	117	26808.84
Time (Assistant)	4318116	168	25703.07
Assistant, as % of Capti			95.88%

Table 5.3: Browsing with Capti and Automation Assistant: Time

Table 5.3 shows that the time spent by the participant on a webpage when using Assistant is lower than when using Capti. The values computed for time intervals up to 200ms long are the most representative of the three, since they are likely to include less noise from unrelated factors (listed above). Those results are reinforced by rows 1 and 2 in Table 5.2, showing that the number of the key presses per webpage visit done during the first part of the study (when Assistant was disabled) is greater than the number of key presses per webpage done during the second part of the study (when Assistant was enabled). Also, the results can be compared with the time measurements in the study described in Section 3.4.3, where the time difference between the baseline screen reader, and Assistant is more pronounced, due to a much more focused experiment (with limited time, and specific tasks).

In a post-completion discussion the participant emphasized that the *perception* during the course of the last 3 weeks of the study was that the time saved with Assistant was substantial. Further, the participant reported that he carefully verifying the values entered into the form fields by Assistant. The verification always returned a positive result, slowly building the participant's trust in the system. The participant opined that once the system gained his trust he would significantly reduce time and effort on verification, resulting in additional gains from using Assistant.

5.2.3 Post-Completion Questionnaire

At the end of the study the participant was asked to answer the standard SUS [36] set of questions in order to determine the usability grade of Automation Assistant, and compare it to the usability of Capti, VoiceOver, and JAWS screen-readers. Note that although JAWS and VoiceOver were not included in this study, the participant is an expert user of both screen-readers, and therefore is able to qualify their long-term usability without the need for a separate experiment. The results of the questionnaire are shown in Table 5.4.

The participant was also asked to compare systems by answering ten comparative questions derived from the SUS questionnaire (the same questions were used in the study in Section 3.4). To answer each question the participant needed to choose one of the four aforementioned systems, or 'none of the above'. The results of the questionnaire are shown in Table 5.5.

As can be seen from both tables, the participant strongly preferred Automation Assistant over all other systems. At the same time, the baseline of the study, Capti, also performed very favorably when compared to the two other screen-readers.

Each system had disadvantages that impacted their overall SUS score. JAWS is very configurable (e.g., users can hide a section of a webpage, disable announcements of tables or headings, etc), which unfortunately also means that JAWS *needs* to be configured to function properly, which

	JAWS	*VoiceOver*	*Capti*	*Assistant*
I think that I would like to use this system frequently	4	3	3	4
I found the system unnecessarily complex	5	5	4	2
I thought the system was easy to use	2	1	4	4
I think that I would need the support of a technical person to be able to use this system	1	1	1	1
I found the various functions in this system were well integrated	2	4	4	4
I thought there was too much inconsistency in this system	2	3	2	3
I would imagine that most people would learn to use this system very quickly	1	1	4	3
I found the system very cumbersome to use	3	4	2	1
I felt very confident using the system	4	4	4	4
I needed to learn a lot of things before I could get going with this system.	5	5	3	2
SUS score	42.5	37.5	67.5	75

Table 5.4: Screen-Readers vs. Automation Assistant: SUS

	Response
Which system would you prefer to use?	Assistant
Which system did you find the most complex?	VoiceOver
Which system did you find the easiest to use?	Capti
Which system would need the most support from a technical person?	JAWS
Which system was the most well integrated?	Capti
Which system was the least consistent?	VoiceOver
Which system do you imagine most people would learn to use most quickly?	Capti
Which system did you find the most cumbersome to use?	VoiceOver
Which system did you feel the most confident using?	JAWS
Which system did you feel required you to learn the most before using it?	VoiceOver

Table 5.5: Screen-Readers vs. Automation Assistant: Direct Comparison

requires both effort and expertise. In contrast, Capti has nearly no configuration options, apart from a few necessities, such as controlling the speech rate; however its default feature set and configuration work well.

VoiceOver is harder to control because of its navigation model, where VoiceOver segments the webpage into different groups. The participant explained that the navigation model of VoiceOver is less consistent in use than the standard navigation model employed by JAWS and Capti.

5.2.4 Discussion

Probably the most important observation about the disadvantages of Assistant is *unpredictability* in providing suggestions, which may not be offered for a number of previously discussed reasons (e.g., see Chapter 2). However, from the user's perspective, the systems behavior is hard to predict, and so the user is forced to choose which action to take: ask for a suggestion after every browsing action (and waste his/her effort every time there is no suggestion), not to ask for suggestions at all, or try to guess when asking for a suggestion will make sense. In the future, this shortcoming can be mitigated by providing the user with a non-intrusive notification about the availability of suggestions. At the same time, this observation once again reinforces the need for any web automation tool to minimize the cost of using automation, including cognitive load and time.

Another observation made during the study is that Assistant, in addition to its ability to direct the user to relevant form fields, buttons, and links, also needs to be able to direct the user towards important parts of the webpage not represented by any control, e.g., plain text. For example, when making a purchase the user might need to review the shipping address that is already on file in the website database. This review *might* cause the user to decide to modify the address, but before that determination can be made the user needs to find and read the address on file. The ability to identify, store, and later suggest webpage areas that have no special action associated with them can thus be a very useful addition. One possible approach would be to adapt the Alt+Space shortcut used in this study to allow the user to 'flag' such areas manually.

A third observation made during the study is that the legitimate absence of feedback on the part of Assistant can be perceived as a problem with the system, rather than an indication of a problem with a webpage, or a mistake on the part of the user. For example, the user is visiting a product page on a shopping website, and the product is 'coming soon'. As a consequence, the Assistant will not suggest pressing the missing "add to cart" button, which would be suggested on other product pages, and which would be expected by the user. While this situation may be easy to recognize for a sighted person, a visually impaired user will waste time trying to figure out what happened and how to purchase the product. This problem might/would be mitigated if Assistant offered some explanation why a suggestion was not made (e.g., even though the model generated "add to cart" suggestion, the post-processing step could not find the relevant button on the webpage).

The fourth observation made about Assistant is the need for a shortcut that would allow the user to backtrack to the previously modified form field, or, more generally, the previously visited webpage element. This feature is not web automation specific and can be imagined as a part of any screen reader. Integrated with a web automation tool, however, it can be very useful in enabling the user to double-check modifications made with the help of a web automation tool, revisit previously reviewed suggestions in chronological order, and return the visual cursor to its original position before jumping through suggestions.

Fifth, is that Assistant needs to allow the user to review (and confirm) from more than one suggestion for a single webpage element. The simplest use case is analogous to the common feature of some webpages and browsers: let the user choose from multiple suggestions for a textbox (e.g., when searching on google.com). With Automation Assistant however, this can be significantly expanded: choose from several possible values for a listbox, choose from several possible *combinations* of values for a multi-choice listbox, choose between submitting a form, and filling out an entire form with suggested values, etc. Moreover, Assistant has the potential to provide better suggestions than a browser or a webpage because Assistant is able to take into account both the content of the webpage, and the user's prior actions on the webpage. This data can be used to determine the conditional probability of a specific form field value to be accepted by the user. For example, if the user is planning a trip, a suggestion to fill in a date on a hotel website may be useful, but using *any* previously entered date may be undesirable. The switching between alternative suggestions for the same webpage element would need to be done with a separate shortcut(s), to provide a clear distinction between jumping through suggested webpage elements and suggestions for one element.

5.3 Conclusion

In this Chapter, I described the longitudinal study conducted with a single participant over a course of 6 weeks. The study showed that Automation Assistant is practical in the long-term use, and provides both perceived and measurable benefits to the system usability, as well as user experience.

The study also revealed a number of improvements that have the potential to improve the usability of the approach described in this dissertation.

Chapter 6

Prior Work

6.1 Overview

As already mentioned, Automation Assistant can be described as an Interface Agent [47, 54]. Web Agents are most often used for transcoding existing websites or automatically generating new websites in order to improve overall user browsing experience, e.g., to facilitate the perception of, or interaction with other visitors of the same or nearby webpages, to increase user's awareness of webpages on a specific topic, to suggest useful links to follow [25, 32, 64], to find relevant Web documents, build personalized document collections (e.g., a newspaper) [43], etc.

Another related branch of research focuses on PBD systems. Automation Assistant was inspired by the ideas described in [65, 66, 67, 68]. In contrast to the standard PBD approaches, our approach does not use macros altogether; instead, it uses the history of past browsing actions to predict future actions. The automatic form-filling feature in modern web browsers is another example of PBD that learns from what the user types into web forms and then helps automate form-filling. The proposed model-based approach, however, subsumes form-filling and even makes it more powerful; specifically, it can propose different values for the same form-fields depending on the browsing context of the user, e.g., if the user enters the first name, the model can predict the related last name(s). Furthermore, the proposed model-based approach can automate other browsing actions such as clicking links and buttons, and, in contrast to macros, it can suggest several possible alternatives at each browsing step. Other approaches to learning/recording, inference and execution of automation instructions are summarized in Table 6.1.

The feasibility of the step-by-step automation of browsing tasks without macros was also ex-

	Learning/Recording	Execution
WebVCR [26]	Explicit PBD	On-demand
WebMacros [71]	Explicit PBD	On-demand
iMacros [8]	Explicit PBD	On-demand
Robofox [49]	Explicit PBD	On-demand
CoScripter [52, 53]	Custom script	On-demand
	Explicit PBD	
	Hybrid PBD	
Smart Bookmarks [48]	Hybrid PBD	On-demand
CoCo [51]	Explicit PBD	On-demand (nat. lang.)
Creo [39]	Explicit PBD	Recommended
Montoto et.el. [58]	Graphical UI	On-demand
Chickenfoot [31]	JavaScript	On-demand
		Event-triggered
	Explicit PBD	
Web data extraction [42, 28]	Custom script	On-demand
Screen-readers [4, 11, 7, 3]	Custom script	On-demand
Trailblazer [30]	CoScripter-based	On-demand
Hearsay [34]	Explicit PBD	On-demand
Automatic form-filling	Implicit PBD	Recommended
Automated Assistant	Implicit PBD	Recommended (one action)

Table 6.1: Comparison of Web Automation Tools

plored in [56, 57, 72], where web automation relied on process models constructed from sequences of actions using machine learning techniques such as clustering, classification, and automata learning. The resulting process models had browsing states as nodes and actions as transitions. However, the approach used to construct the process models was not incremental, requiring that the model be rebuilt in order to accommodate any changes, such as learning new transition (or unlearning old ones). Moreover, the approach implied the system's ability to split clickstreams into sequences corresponding to sessions, cluster similar session sequences into similar groups, and then learn the automata, potentially introducing errors in each of those steps (e.g., it is not clear what is a good indicator of a splitting point, and if one exists at all in our domain). Finally, to predict the next browsing step, this approach required the exact match between the current browsing state and a state in the model (which represents a first-order Markovian process), thus limiting the predicting power of the approach in case of deviations from the process.

The model described in this paper is modeling a higher-order Markovian process, making it possible to predict the next step even if the user deviates from previously learned sequences of browsing actions. Furthermore, the model learns to automate browsing steps incrementally, making it possible to insert new and to remove outdated data on the fly without rebuilding the model. The model takes the history of browsing steps as they are without the need to segment the history into sessions.

Related work on the prediction of user browsing behavior also exists in other application domains such as prefetching [62, 38], e-commerce [73], content personalization [29], etc. These applications are only somewhat related because they pursue different goals, e.g., prefetching files or adjusting the webpage layout. These applications often construct probability graphs (similar to process models) and tend to do that offline; they do not handle form filling, etc. Notably, in [35], sequence alignment is used to find the next link the user could click in a website directory (e.g., Yahoo!). Apart from the difference in the application domain, the described approach also differs in that it aligns the current user session to multiple past sessions that were previously clustered and organized into clickstream trees. The ensuing limitations are similar to those of the process-model approach: errors of session segmentation, as well as in clustering, and inefficiency of off-line model construction. In contrast, the approach proposed in this paper avoids the complexity and errors of clustering and segmentation, constructs the model incrementally, automates both form-filling and clicking, introduces modifications to the alignment algorithm, and, rather than using the final result of sequence alignment, it uses the inner-workings of the algorithm (i.e. the sequence alignment table) to predict multiple possible steps the user could take.

6.2 Non-Visually Accessible End-User Web Automation

Most of the popular screen-readers including JAWS [4], NVDA [11], Supernova [3], and Window-Eyes [7], enable users to automate tasks by writing scripts. In addition, the JAWS' ResearchIt tool automates a lookup of information, e.g., dictionary definitions for a particular word. ResearchIt can be accessed via hotkeys, and also exposes a scripting API that can be used to add additional information sources.

Trailblazer [30] is a CoScripter-based tool that interfaces with the JAWS screen-reader to make the CoScripter interface accessible. Trailblazer also provides access to a macro database created by the users of CoScripter. The novelty of Trailblazer is in the ability to adapt a macro recorded for one website and use the same macro to accomplish the same tasks on other websites.

The HearSay non-visual web browser [34] includes a prototype of accessible web automation for recording and replaying of macros. Hearsay also has speech-enabled interface and parameterized macros. To the best of the author's knowledge, it is the first PBD web automation tool designed specifically for visually impaired users.

6.3 Web Automation for Sighted End-Users

Tools described in this section are designed for sighted end users, and as such are not accessible for the visually impaired users. While some of these tools can theoretically be accessed with the help of a screen reader, they are far from being usable by screen-reader users, as they require a lot of interaction.

WebVCR [26], WebMacros [71], iMacros [8], and Robofox [49] systems all use the explicit PBD approach. WebVCR is designed as a tool for recording "smart bookmarks": bookmarks to webpages without a static URL, and hence can only be addressed by a sequence of steps (e.g. following a sequence of links). WebMacros is similar to WebVCR, but is implemented as a web proxy. This allows WebMacros to easily determine when a webpage has finished loading, which is critical for the correct timing of action replaying, and for handling webpage redirects during recording. Robofox makes use of automatically generated assertions to help detect and correct macro failures.

CoScripter [52] is the most prolific and versatile web automation tool in use. CoScripter enables both PBD and manual (collaborative, using web wiki) creation and editing of macros. Co-

Scripter Reusable History (also known as ActionShot [53]) is a tool that records every user action (without the need to initiate/terminate recording) and allows the user to modify the list of actions using the same collaborative interface. ActionShot can convert user actions into pseudo-natural language strings for easier understanding and relate each action to a screenshot of the webpage. ActionShot requires that the user manually indicate the actions to be turned into a macro.

CoCo [51] is a tool that provides an abstraction layer on top of a browser by leveraging CoScripter and ActionShot. CoCo takes a natural-language query, maps the query to a macro, executes the macro (possibly asking the user to evaluate parameters), and extracts a part of a webpage as a response to the user. The macros are retrieved both from CoScripter's collaborative database and from user's web history logs that have been continuously recorded by ActionShot. The history log is automatically split into sets of consecutive steps (macros) using several rules based on URL changes and the amount of time between steps. CoCo's approach of implicit creation of automation raises the issue of reliability of its heuristics and the cost of identifying and fixing potential errors. Furthermore, in the case of errors in the recording, the user has to edit the recording manually.

Creo [39] is another PBD tool that uses the content of a visited webpage to automatically suggest that the user initiate one (or more) of the prerecorded macros. Creo uses knowledge-bases to map webpage content to specific macros by generalizing words used on the webpage and in the macro. For example, a macro may contain a search query for a "cat", which can be generalized to an "animal". Creo will suggest that the user run that same macro when s/he visits a page with any other word that can be generalized to an "animal" (e.g., "dog"). The recommendation-based execution approach used by Creo eliminates some of the challenges of the on-demand and event-based execution; however, Creo's automatic macro suggestion mechanism is built on the premise that the content of the visited page can suggest to the user which macro to run.

Chickenfoot [31] is a browser plugin that extends JavaScript to enable easier handcrafting of webpage automation and customization scripts. Chickenfoot uses pattern matching for DOM addressing. For example, the following command will fetch the Google homepage in the background, and click the button labeled "I am feeling lucky": `with (fetch('www.google.com')) { click('I am feeling lucky') }`

Montoto et al. [58] propose a handcrafting automation with a GUI: the user explicitly specifies each action by right-clicking the target element, and choosing the action to be recorded. This allows the system to make a more accurate recording of an action sequence. In addition, the authors propose a method for detection of action completion (described in Section 2.3.6), and an approach for handling webpage element addressing: represent an address as a partial absolute XPath, extending it from the node towards the root of the tree only far enough to ensure that the address will return a single node.

6.4 Web Data Extraction

Tools for Web data extraction (OXPath [42], Lixto [28],Visual Web Ripper (visualwebripper.com), Web Content Extractor (newprosoft.com/web-content-extractor.htm), Chickenfoot [31] (discussed above)) are designed for the purpose of automatic information extraction from human oriented Web interfaces. This requires automation of user browsing actions much the same way as is done in tools designed specifically for the purpose of Web automation. However, while Web automation tools are focused on navigation, Web data extraction tools are focused on large scale data extraction.

A good example of a web data extraction language is OXPath [42], an extension of the XPath language. OXPath is making it possible to write expressions that will instruct the executing engine to (a) address DOM nodes (like XPath), (b) fill forms (c) trigger DOM events, and (d) extract and return information for the specified DOM nodes (e.g., form field values, style attributes, etc.). For example, the following expression will extract current articles, titles and sources from Google News [42]:

```
doc('news.google.com')//div[@class ='story']:<story>
[.//h2:<title=string(.)>]
[.//span[style::color='#767676']:<source=string(.)>]
```

6.5 Addressing of Webpage Elements

XPath is a language for querying parts of DOM tree, with the query expressed as an address within the document. While XPath is the primary technique for DOM querying, there exist a number of related technologies. The latest specification of the XPath (2.0) is a subset of XQuery 1.0 language [21], and is also used in Extensible Stylesheet Language Transformations (XSLT) language [13]. Both XQuery and XLST are designed to express arbitrary XML to XML data transformations, and have the same expressive power. XQuery extends XPath with the query prolog, element and attribute constructors, the ”FLWOR” clauses: FOR, LET, WHERE, ORDER BY, RETURN (of which XPath has only FOR), and the typeswitch expression. Other standardized methods for querying a DOM tree are XPointer [14], and XLink [17]. The relationship of the above technologies is depicted in Figure 6.1.

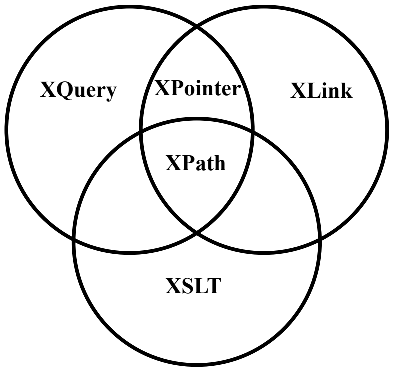

Figure 6.1: DOM Addressing Methods

CSS Selectors [19] is another standardized addressing method very similar in expressiveness to XPath. A comparative example of XPath and CSS is in Table 6.2.

Query	CSS 3	XPath 2
All Elements	*	//*
All Child Elements	p > *	//p/*
Element By ID	#foo	//*[@id='foo']
All P with an A child	Not possible	//p[a]
Next Element	p + *	//p/following-sibling::*[0]

Table 6.2: Comparison of XPath and CSS queries

In addition to the standardized query and transformation languages there are also specialized extensions of XPath, such as OXPath [42] (also see Section 6.4) and SXPath [61] (extending XPath with semantic information processing).

More sophisticated methods of addressing that are resilient to *some* possible webpage changes were developed in [58, 46, 40, 53, 31]. Most of them have already been described in the previous Section. However, the idea explored in [46, 40] deserves a special mention: the use of machine learning to train a model of a node based on structural and visual features.

Chapter 7

Conclusion

In this dissertation, I presented a novel approach to accessible, non-visual web automation embodied in the prototype system *Automation Assistant*: an interface agent with an accessible non-visual user interface for step-by-step automation of repetitive web browsing tasks.

The three main contributions of this work are systematization and analysis of the complexities and constraints faced by designers of accessible web automation tools (Chapter 2); a novel, accessible, and usable non-visual web automation user interface (Chapter 3); and a novel method for predicting the most probable actions the user can take at a given time when browsing the Web (Chapter 4).

The design of Automation Assistant was evaluated by testing the performance of the prediction algorithm, and conducting two user studies. The evaluation of the workflow (Section 3.4) showed the feasibility of the approach. The fact that one of the two user interfaces evaluated in Section 3 turned out to be much more usable than the other serves as a reminder of the importance of attention to detail in interface design in general, and accessible interface design in particular. The longitudinal study (Chapter 5) provided additional valuable feedback that would be impossible to obtain in a short-term, focused study.

Automation Assistant was designed to be computationally efficient, nonintrusive, robust, and trustworthy. The approach described in this thesis has shown the potential to meaningfully improve the usability and the accessibility of non-visual browsing, and web automation in general. This is, however, only a single step towards making truly accessible Web a reality.

7.1 Future Work

With further R&D effort, Assistant has the potential to morph into an eyes-free, voice-controlled system, benefiting people both with and without vision impairments.

There are several broad directions for the advancement of the approach to web automation described in this dissertation: the workflow / the user interface, accuracy of predictions, efficient management of the browsing data.

Some of the possible directions for user interface improvement were listed in the results of the longitudinal study. This includes making it easier for the user to figure out when new suggestions are available; focusing the user on webpage elements that do not have any explicit actions associated with them, such as plain text, in order to enable the user to easily find and review content; providing better feedback to the user about the reasons that certain suggestions are not offered (or conversely, are offered); introducing the ability to 'backtrack' one's steps chronologically to simplify the review process of the modified form fields. Other possible improvements include re-introducing the (possibly limited) ability to automate a sequence of actions, but without the inherent disadvantages detailed in Section 2; generalizing Assistant beyond the browser window, and making it possible to automate all software running on the operating system.

Progress in prediction accuracy is another important direction. The accuracy can be improved by using additional contextual and personal information: what time of the day it is, what day of the week it is, if the user is accessing the web via a mobile device or via a personal computer, what the current physical location of the user is, etc. The time interval between user actions can serve an another important signal: a specific time interval pattern can provide insight into the current user workflow and intent. The dynamic DOM updates present another technical challenge: how to interpret those updates correctly, and how a dynamic update should affect the predicted user actions. Another direction is in clustering profiles (browsing data) of different users, using profiles of some users to predict actions of others without jeopardizing privacy. Yet another direction is to automatically purge the model from decayed data.

Finally, the exploration of input modalities other than the keyboard may be of interest as well. The Assistant's simple interface lends itself beautifully to simple input modalities that include only a few controls. Further, browsing data collected while browsing the web using one type of device (e.g., a personal computer) should be useful on a different device (e.g., a smartphone). Most importantly, the user interface revolution, started in 2007 with the re-introduction of the touch-based devices, has created the demand for non-visual, and small-screen-friendly user interfaces, making every person with a smartphone a potential beneficiary of advances in Web automation. This quickly growing group of users, together with the visually impaired users of the Web should provide a strong incentive for new research in this area.

Bibliography

[1] AI Squared: ZoomText Magnifier/Reader. http://www.aisquared.com/zoomtext/more/zoomtext_magnifier_reader/. (p. 1).

[2] Apple: VoiceOver screen-reader. http://www.apple.com/. (p. 2).

[3] Dolphin Computer Access: SuperNova screen-reader. http://www.yourdolphin.com/. (pp. 2, 60, and 62).

[4] Freedom Scientific: JAWS screen-reader. http://www.freedomscientific.com/. (pp. 1, 6, 60, and 62).

[5] Freedom Scientific: JAWS skim reading. http://www.freedomscientific.com/Training/Surfs-up/Skim_Reading.htm. (p. 3).

[6] Freedom Scientific: MAGic screen magnification software. http://www.freedomscientific.com/products/lv/magic-bl-product-page.asp. (p. 1).

[7] GW Micro: Window-Eyes screen-reader. http://www.gwmicro.com/. (pp. 1, 7, 60, and 62).

[8] iopus: imacros. http://www.iopus.com/download/iMacros-Manual.pdf. (pp. 3, 60, and 62).

[9] Microsoft agent. http://msdn.microsoft.com/en-us/library/ms695784. (p. 19).

[10] Mindtools: Speed reading. http://www.mindtools.com/speedrd.html. (p. 3).

[11] NonVisual Desktop Access: NVDA screen-reader. http://www.nvda-project.org/. (pp. 1, 60, and 62).

[12] World Wide Web consortium (W3C): XML path language (XPath), 1999. http://www.w3.org/TR/xpath. (p. 16).

[13] World Wide Web consortium (W3C): XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt/. (p. 64).

[14] World Wide Web consortium (W3C): XML Pointer Language (XPointer), 2002. `http://www.w3.org/TR/WD-xptr`. (p. 64).

[15] IVONA software: IVONA multi-lingual speech synthesis system, 2005. `http://www.ivona.com/`. (p. 28).

[16] World Wide Web consortium (W3C): Document Object Model (DOM), 2005. `http://www.w3.org/DOM/`. (p. 16).

[17] World Wide Web consortium (W3C): XML Linking Language (XLink) version 1.1, 2010. `http://www.w3.org/TR/xlink11/`. (p. 64).

[18] Internet usage statistics: The internet big picture world internet users and population stats. 2011. `http://www.internetworldstats.com/stats.htm`. (p. 1).

[19] World Wide Web consortium (W3C): Selectors level 3, 2011. `http://www.w3.org/TR/css3-selectors/`. (p. 65).

[20] World Wide Web consortium (W3C): WAI guidelines and techniques, 2011. `http://www.w3.org/WAI/guid-tech.html`. (p. 2).

[21] World Wide Web consortium (W3C): XQuery 1.0: An XML query language, 2011. `http://www.w3.org/TR/xquery/`. (p. 64).

[22] Faisal Ahmed, Yevgen Borodin, Yury Puzis, and I. V. Ramakrishnan. Why read if you can skim: towards enabling faster screen reading. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, W4A '12, pages 39:1–39:10, New York, NY, USA, 2012. ACM. (p. 3).

[23] Faisal Ahmed, Yevgen Borodin, Andrii Soviak, Muhammad Islam, I.V. Ramakrishnan, and Terri Hedgpeth. Accessible skimming: Faster screen reading of web pages. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 367–378, New York, NY, USA, 2012. ACM. (p. 3).

[24] Faisal Ahmed, Andrii Soviak, Yevgen Borodin, and I.V. Ramakrishnan. Non-visual skimming on touch-screen devices. In *Proceedings of the 2013 International Conference on Intelligent User Interfaces*, IUI '13, pages 435–444, New York, NY, USA, 2013. ACM. (p. 3).

[25] Corin R. Anderson, Pedro Domingos, and Daniel S. Weld. Adaptive web navigation for wireless devices. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 2*, IJCAI'01, pages 879–884, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. (p. 59).

[26] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. Automating web navigation with the webvcr. *Comput. Netw.*, 33(1-6):503–517, June 2000. (pp. 60 and 62).

[27] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the System Usability Scale. *Int. J. Hum. Comput. Interaction*, pages 574–594, 2008. (p. 30).

[28] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 119–128, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. (pp. 60 and 64).

[29] J. Bian, A. Dong, X. He, S. Reddy, and Y. Chang. User action interpretation for online content optimization. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1, 2012. (p. 61).

[30] Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: enabling blind users to blaze trails through the web. In *Proceedings of the 14th international conference on Intelligent user interfaces*, IUI '09, pages 177–186, New York, NY, USA, 2009. ACM. (pp. 7, 14, 60, and 62).

[31] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 163–172, New York, NY, USA, 2005. ACM. (pp. 60, 63, 64, and 65).

[32] Andrea Bonomi, Marcello Sarini, and Giuseppe Vizzari. Engineering environment-mediated multi-agent systems. chapter Combining Interface Agents and Situated Agents for Deploying Adaptive Web Applications, pages 103–114. Springer-Verlag, Berlin, Heidelberg, 2008. (p. 59).

[33] Yevgen Borodin. Automation of repetitive web browsing tasks with voice-enabled macros. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*, ASSETS '08, pages 307–308, New York, NY, USA, 2008. ACM. (p. 3).

[34] Yevgen Borodin, Faisal Ahmed, Muhammad Asiful Islam, Yury Puzis, Valentyn Melnyk, Song Feng, I. V. Ramakrishnan, and Glenn Dausch. Hearsay: a new generation context-driven multi-modal assistive web browser. In *Proceedings of the 19th international conference on World Wide Web*, WWW '10, pages 1233–1236, New York, NY, USA, 2010. ACM. (pp. 60 and 62).

[35] Amit Bose, Kalyan Beemanapalli, Jaideep Srivastava, and Sigal Sahar. Incorporating concept hierarchies into usage mining based recommendations. In *Proceedings of the 8th Knowledge discovery on the web international conference on Advances in web mining and web usage analysis*, WebKDD'06, pages 110–126, Berlin, Heidelberg, 2007. Springer-Verlag. (p. 61).

[36] John Brooke. SUS: A quick and dirty usability scale. In P. W. Jordan, A. Thomas, B. Weerdmeester, and I.L. McClelland, editors, *Usability evaluation in industry*. Taylor and Francis, 1996. (pp. 29 and 54).

[37] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993. (p. 6).

[38] Josep Domènech, Bernardo de la Ossa, Julio Sahuquillo, José-Antonio Gil, and Ana Pont. A taxonomy of web prediction algorithms. *Expert Syst. Appl.*, 39(9):8496–8502, 2012. (p. 61).

[39] Alexander Faaborg and Henry Lieberman. A goal-oriented web browser. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 751–760, New York, NY, USA, 2006. ACM. (pp. 60 and 63).

[40] Ruslan R. Fayzrakhmanov, Christoph Herzog, and Kordomatis Iraklis. Web objects identification for web automation: Objects and their features. In *Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems*, IHC, pages 292–295, Porto Alegre, Brazil, Brazil, 2013. Sociedade Brasileira de Computa. (p. 65).

[41] Leah Findlater and Krzysztof Z. Gajos. Design space and evaluation challenges of adaptive graphical user interfaces. *AI Magazine*, 30(4):68–73, 2009. (p. 4).

[42] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Sellers. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, 2013. (pp. 60, 64, and 65).

[43] Daniela Godoy, Silvia Schiaffino, and Anala Amandi. Interface agents personalizing web-based tasks. *Cognitive Systems Research*, 5(3):207 – 222, 2004. Special Issue on Intelligent Agents and Data Mining for Cognitive Systems. (p. 59).

[44] Stuart Goose and Carsten Möller. A 3D audio only interactive web browser: using spatialization to convey hypermedia document structure. In *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, MULTIMEDIA '99, pages 363–371, New York, NY, USA, 1999. ACM. (p. 3).

[45] Simon Harper and Neha Patel. Gist summaries for visually impaired surfers. In *Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility*, ASSETS '05, pages 90–97, New York, NY, USA, 2005. ACM. (p. 3).

[46] Christoph Herzog, Iraklis Kordomatis, Wolfgang Holzinger, Ruslan R Fayzrakhmanov, and Bernhard Krüpl-Sypien. Feature-based object identification for web automation. 2013. (p. 65).

[47] Z. Huang, A. Eliens, A. van Ballegooij, and P. de Bra. A taxonomy of web agents. In *Database and Expert Systems Applications, 2000. Proceedings. 11th International Workshop on*, pages 765 –769, 2000. (pp. 4, 19, and 59).

[48] Darris Hupp and Robert C. Miller. Smart bookmarks: automatic retroactive macro recording on the web. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 81–90, New York, NY, USA, 2007. ACM. (p. 60).

[49] Andhy Koesnandar, Sebastian Elbaum, Gregg Rothermel, Lorin Hochstein, Christopher Scaffidi, and Kathryn T. Stolee. Using assertions to help end-user programmers create dependable web macros. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 124–134, New York, NY, USA, 2008. ACM. (pp. 60 and 62).

[50] Tessa Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI Workshop on Usable AI*, page 4, 2008. (p. 8).

[51] Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. A conversational interface to web automation. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 229–238, New York, NY, USA, 2010. ACM. (pp. 60 and 63).

[52] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM. (pp. 9, 60, and 62).

[53] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. Here's what i did: sharing and reusing web activity with actionshot. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 723–732, New York, NY, USA, 2010. ACM. (pp. 60, 63, and 65).

[54] Henry Lieberman. Autonomous interface agents. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, pages 67–74, New York, NY, USA, 1997. ACM. (pp. 4, 19, and 59).

[55] Darren Lunn, Simon Harper, and Sean Bechhofer. Identifying behavioral strategies of visually impaired users to improve access to web content. *ACM Trans. Access. Comput.*, 3(4):13:1–13:35, April 2011. (p. 3).

[56] Jalal Mahmud, Yevgen Borodin, I. V. Ramakrishnan, and C. R. Ramakrishnan. Automated construction of web accessibility models from transaction click-streams. In *Proceedings of the 18th international conference on World Wide Web*, WWW '09, pages 871–880, New York, NY, USA, 2009. ACM. (p. 61).

[57] Jalal Mahmud, Zan Sun, Saikat Mukherjee, and I.V. Ramakrishnan. Abstract web transactions on handhelds with less tears. In *Proceedings of the Workshop MobEA IV - Empowering the Mobile Web*, 2006. (p. 61).

[58] Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. Automating navigation sequences in ajax websites. In *Proceedings of the 9th International Conference on Web Engineering*, ICWE '09, pages 166–180, Berlin, Heidelberg, 2009. Springer-Verlag. (pp. 6, 8, 17, 60, 63, and 65).

[59] E. Murphy, R. Kuber, Philip Strain, G. McAllister, and W. Yu. Developing sounds for a multimodal interface: Conveying spatial information to visually impaired web users. pages 348–355, Montreal, Canada, 2007. Schulich School of Music, McGill University, Schulich School of Music, McGill University. (p. 3).

[60] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *CABIOS*, 4:11–17, 1988. (p. 41).

[61] Ermelinda Oro, Massimo Ruffolo, and Steffen Staab. SXPath: extending XPath towards spatial querying on web documents. *Proc. VLDB Endow.*, 4(2):129–140, November 2010. (p. 65).

[62] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World Wide Web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996. (p. 61).

[63] Bambang Parmanto, Reza Ferrydiansyah, Andi Saptono, Lijing Song, I Wayan Sugiantara, and Stephanie Hackett. Access: accessibility through simplification & summarization. In *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, W4A '05, pages 18–25, New York, NY, USA, 2005. ACM. (p. 3).

[64] Mike Perkowitz and Oren Etzioni. Adaptive web sites. *Commun. ACM*, 43(8):152–158, August 2000. (p. 59).

[65] Yury Puzis. Accessible web automation interface: a user study. In *Proceedings of the 14th international ACM SIGACCESS conference on Computers and accessibility*, ASSETS '12, pages 291–292, New York, NY, USA, 2012. ACM. (p. 59).

[66] Yury Puzis. An interface agent for non-visual, accessible web automation. In *Adjunct proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST Adjunct Proceedings '12, pages 55–58, New York, NY, USA, 2012. ACM. (p. 59).

[67] Yury Puzis, Eugene Borodin, Faisal Ahmed, Valentine Melnyk, and I. V. Ramakrishnan. Guidelines for an accessible web automation interface. In *The proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility*, ASSETS '11, pages 249–250, New York, NY, USA, 2011. ACM. (p. 59).

[68] Yury Puzis, Yevgen Borodin, Faisal Ahmed, and I. V. Ramakrishnan. An intuitive accessible web automation user interface. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, W4A '12, pages 41:1–41:4, New York, NY, USA, 2012. ACM. (pp. 21 and 59).

[69] Yury Puzis, Yevgen Borodin, Rami Puzis, and I.V. Ramakrishnan. Predictive web automation assistant for people with vision impairments. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 1031–1040, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee. (pp. 4, 19, and 37).

[70] Luz Rello. Dyswebxia: a model to improve accessibility of the textual web for dyslexic users. *SIGACCESS Access. Comput.*, (102):41–44, January 2012. (p. 3).

[71] Alex Safonov, Joseph A. Konstan, and John V. Carlis. WebMacros - a proxy-based system for automating user interactions with the web, 2010. (pp. 60 and 62).

[72] Zan Sun, Jalal Mahmud, I. V. Ramakrishnan, and Saikat Mukherjee. Model-directed web transactions under constrained modalities. *ACM Trans. Web*, 1(3), September 2007. (p. 61).

[73] Silvana Vanesa Aciar, Christian Serarols-Tarres, Marcelo Royo-Vela, and Josep Lluis De la Rosa i Esteva. Increasing effectiveness in e-commerce: recommendations applying intelligent agents. *International Journal of Business and Systems Research*, 1(1):81–97, 01 2007. (p. 61).

[74] M. S. Waterman and T. F. Smith. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981. (pp. 38 and 47).

[75] Wai Yu, Ravi Kuber, Emma Murphy, Philip Strain, and Graham McAllister. A novel multimodal interface for improving visually impaired people's web accessibility. *Virtual Real.*, 9(2):133–148, January 2006. (p. 3).

Appendices

Appendix A

Webpage Elements Representation

When the user visits a webpage element with Capti, or Automated Assistant, the element is voiced according to the following specifications:

BUTTON: LABEL [SUBTYPE] [‘button’] [‘pressed’]
CHECK_BOX: LABEL ‘checkbox’ [‘un’]‘checked’
RADIO_BUTTON: LABEL ‘radio button’ [‘un’]‘checked’
LIST_BOX: LABEL ‘listbox’ [‘nothing selected’ |‘blank’ |selected item]
LIST_BOX (multi-choice): LABEL ‘multiselect’ [‘nothing selected’ |‘blank’ |selected item |# ‘of’ # ‘items selected’]
TEXT_BOX: LABEL ‘textbox’ [‘blank’ |VALUE]
TEXT_BOX (multi-line): LABEL ‘textarea’ [‘blank’ |VALUE]
TEXT_BOX (password): LABEL ‘password’ [‘blank’|len(VALUE) ‘char[s]’]
TABLE: ‘table’ LABEL # ‘row[s]’ # ‘column[s]’
TEXT: LABEL [‘visited’] [‘link’] [‘heading level’ #]
IMAGE: LABEL [‘image’]
IMAGE (link): LABEL [‘visited’] [‘link’] ‘image’
ENUMERATION: ‘list’ # ‘item[s]’
SEPARATOR: ‘separator’
FRAME: ‘frame’
LANDMARK: LABEL [SUBTYPE]
Suffix: [‘disabled’][‘read only’][‘required’][‘crossed out’][‘suggested’ [SUGGESTED_VALUE]]

In this list LABEL stands for the textual content of the element: either plain text, or a descriptive label (e.g., ‘First Name’ for textbox). VALUE stands for text entered into, or selected in a form field.

Appendix B

Webpage Navigation

B.1 Navigation by Tokens

Navigation by tokens can be done on several levels of granularity: character, word, unit, paragraph, table cell tokens. When navigating at the unit level and above, narrate the element's string representation (see definition in Appendix A). When navigating at levels below unit, narrate only the navigable string.

Note: for any characters repeated over 3 times, read the number and the character or its description in plural, e.g., "10 spaces", "5 dashes" (only when plural form is known). If plural form is unknown say "10 symbols X" (unit and word navigation only).

Additional reading features:

1. Read current unit string representation from the beginning without moving the current position.
2. Move to the beginning/end of unit, and either read the first character, or say 'end' (in edit mode the browser cursor will move to the beginning/end of the line, not the textfield). Echo the first character of the string, or 'end', respectively.
3. Navigating left, right, or down from a TABLE element will move to the first cell of that table (if any).

When the transition to the new token is complete, start narrating the new current token. For levels above character, with each new word read position advances to the word's last character (using TTS bookmarks), until the end of the token is reached. System prompts are narrated, but position only moves on the tokens of the navigable string, ignoring the rest. When token is finished, the reading stops (unless in continuous mode, see Section B.3).

B.2 Navigation by Element Type

The user can navigate by element type (‘group’), e.g., buttons, links, jumping ‘over’ elements in between.

When user navigates by type (Table B.1), move the virtual cursor to the beginning of the previous/next predefined group of elements (or a single element). When the transition is complete, start voicing the group (element), until the end of the group (element) is reached.

Shortcut	Semantics	Shortcut	Semantics
A	LINK	V	*Link* that was visited before
B	BUTTON	N	*Element* that is not a link
E	TEXT_BOX	1…6	*Heading*, HTML5 elements h1…h6
X	CHECK_BOX	H	*Heading*, union of HTML5 elem. h1…h6
I	FRAME	U	*Update*, dynamically updated elements
T	TABLE	P	*Paragraph*, identified with heuristics
L	ENUMERATION	Tab	*Tabbable* element: non-disabled formfield, link, button
G	IMAGE	Ctrl+arrows	LANDMARK
R	RADIO_BUTTON	C	LIST_BOX

Table B.1: Navigation by Type: Element Types, Navigation Shortcuts

B.3 Continuous Navigation

When in continuous reading mode, start reading from the current position. When reached the end of the unit token transition to the next token and continue reading.

Instead of sending to the TTS the content of all the navigable nodes (may overload the TTS buffer) or sending element by element (which results in broken intonation and TTS start latency), content should be sent to the TTS in chunks of several nodes / units. This will result, on one hand, in merging parts of the same sentence that are split across several nodes, and consequently in correct TTS intonation. On the other hand nodes that do not belong to the same sentence, and have no punctuation, may merge as well, resulting in incorrect intonation. This should be corrected by adding a period between all nodes that do not belong to the same sentense. A node represents a continuation of a sentense from a preceeding, adjacent node iff both belong to the same heading (h1…h6) group, or the same paragraph group (identified with heuristics).

B.4 Invalidation of the Current Position

If the current position of the virtual cursor is invalidated, the general rule is: do nothing until some mechanism (user command, continuous reading in progress, focus change etc.) causes a change of current position. This will prevent automatic, and unexpected (by the user) position changes. Current position can become invalid for the following reasons:

- The webpage can set the focus to a hidden element.
- Current position can be deleted by a dynamic tree update (deleted subtree).
- Current position can become hidden due to an attribute update.