ACaMIA: Automating Configuration across Multiple Interactive Applications

Paper # 22 — 14 pages

ABSTRACT
Desktop setup and configuration is a common task in any computing environment, and its automation would save time and money. Existing automation tools are generally application-specific (i.e., macros). Most non-trivial configuration tasks, however, have dependencies across applications. To allow automation in an application-independent manner, past work circumvents the GUI by modifying the kernel to track causal dependencies across applications and then taking diffs of the affected files. Such kernel modifications, however, are unsuitable for the commercial operating systems typical in the desktop environment, and file diffs are fragile in the face of complex, application-specific and binary file formats.

This paper introduces ACaMIA, a tool that uses a new approach to automating configuration. Instead of circumventing the GUI, ACaMIA recognizes that the GUI is the right layer for this task. Modern GUIs are rich enough to tackle most configuration tasks directly, without fragile state introspection techniques. And unlike application file formats, they present a uniform interface that is relatively consistent across applications. By watching a few users perform a configuration task, ACaMIA generates a canonical automation script that enables future users to perform the same task. We implemented a prototype of ACaMIA and showed that it works effectively in automating the configuration tasks in our lab that most commonly required administrator intervention.

1 INTRODUCTION
Desktop configuration updates are basic ongoing tasks in enterprise and campus networks. New software installations, upgrades, security patches, and problem fixes all trigger configuration updates. Such updates typically involve user interaction via the graphical user interfaces of multiple applications. Installing a new software package, for example, can easily involve a Web browser interacting with a software download site, an email client (to acquire a license for the package), an installer, and the package itself. Automating this currently manual interaction promises to save organizations both user time and administrative cost [12]. But despite these substantial incentives, existing automation systems have significant limitations — they either work only within a single application (a typical example is Javascript macros, which work only within a Web browser [3]) or require changes to the operating system kernel to track dependencies across applications [15], which makes them unacceptable to most organizations, which are committed to using a stock, proprietary, off the shelf operating system.

This paper introduces ACaMIA, a new approach for automating configuration updates. The basic idea behind ACaMIA is to 1) record the graphical user interface actions that an initial, small set of users take when they perform the configuration update, 2) combine the resulting traces to extract a parameterized sequence of GUI actions that together accomplish the update, and 3) apply this sequence on demand to automatically perform the update to other machines. A key insight behind this approach is the identification of the graphical user interface as the appropriate interposition layer for recording and replaying configuration actions. While working at this layer imposes significant technical challenges (which may be why previous systems were designed to avoid dealing with the graphical user interface altogether [15]), the rewards for a viable solution are substantial — the system has access to a universal interface that enables it to work seamlessly across multiple applications, including applications it has never previously encountered.

ACaMIA generates a sequence of parameterized GUI actions that automate and canonicalize a particular configuration task. Specifically, given a set of recorded configuration traces that together include the full range of GUI actions required to perform the update on various machines, ACaMIA analyzes the set to classify the user actions as follows:

- **Auto Enter**: Actions that are the same in all traces. Such actions are placed in the parameterized action sequence intact.
- **User Enter**: Actions that only the human user knows. Examples include passwords and personal preferences. In the parameterized action sequence these actions prompt the user for the information required to complete the action. Note that since ACaMIA records passwords at the level of the graphical user interface (which almost invariably obscures passwords), it does not record the actual passwords.
- **Possible Auto Enter**: Actions with a few common values that occur repeatedly across the traces. A common example is printer names — organizations usually have a standard set of printers. In the parameterized action
sequence these actions let the user select one of the previously observed values or enter a new value.

- **Irrelevant**: Actions that are irrelevant to the configuration update at hand. A common example is interactions with entertainment software such as music or video players that take place while the user is performing the configuration task. These actions to not appear in the parameterized action sequence.

The key technical challenge for ACaMIA is combining multiple recorded GUI traces to obtain a single parameterized sequence of actions that will perform the update on other machines. This process is difficult for three reasons.

- First, UI objects have no global names and their state depends on the UI context. For example a text box has no global name and does not take input unless it is in an open window and the window has no open child window. Thus, ACaMIA creates its own naming system that includes the UI context, and hence identifies GUI objects and their state across different machines.

- Second, a trace of a successful configuration activity may contain erroneous segments. For example, a user may enter a wrong value, move forward to the next set of windows, realize her mistake, go back to the corresponding window, correct the error, then continue her configuration task. ACaMIA parses the traces to learn when a value is committed and removes erroneous assignments.

- Third, ACaMIA must map sequences of actions that occur in different orders in different traces. For example, one trace may have the user download a software package, then obtain a certificate, and a second trace may show the two tasks in reverse order. ACaMIA cannot address this problem by ignoring action order altogether, because in most configurations order matters, e.g., one cannot access a child window unless one opens its parent window. ACaMIA uses the UI context to map the same actions across traces. It learns valid action order by exploiting that actions are possible only when they become in-focus, e.g., one has to click the file menu before the “save” action is available to the user interface.

We implemented ACaMIA using the Microsoft Active Accessibility interface. Originally developed to enable accessibility aids for users with impaired vision, the accessibility interface has been built into all versions of the Windows platform since Windows 98 and is now widely supported on this platform. Moreover, there is a strong push to incorporate similar functionality in Linux and other open software projects.

We evaluated ACaMIA by using it to automate the five most common configuration updates in our department that typically require administrator intervention, such as IMAP configuration, printer configuration, etc. Our evaluation employs a population of 12 users, who perform the configuration updates both manually and using ACaMIA. The users perform the configurations on their personal machines. Our results show that ACaMIA is effective at automating configuration updates. Specifically:

- ACaMIA reduced the average time a user spent on these configuration problems by almost 3x.
- ACaMIA automated over 90% of the required actions, thus reducing the average number of user actions per configuration task from 25 to less than 2. Of these remaining non-automated actions, more than 90% of them are passwords or other personal information which cannot be generically automated.

## 2 Related Work

Configuration management is an important, unsolved area in systems research. The closest to our work is AutoBash [15], which helps recover from configuration errors by recording user actions to fix a problem on one computer and replaying them on another computer that is experiencing the same problem. ACaMIA is inspired by AutoBash but differs from it in two important ways. First, AutoBash requires kernel modifications to track causal dependencies across processes and system states. Kernel modifications however are unsuitable for the commercial operating systems typical in the desktop environment. Second, AutoBash automates configuration updates by taking diffs of the affected files and applying them to a different machine. File diffs however are fragile in the face of complex application-specific file formats. ACaMIA operates at the user interface which, unlike file formats, is relatively-consistent across applications. Furthermore, ACaMIA does not require kernel modifications.

Much prior work focuses on diagnosing configuration problems. PeerPressure [16] and its predecessor, Strider [17] diagnose configuration problems caused by erroneous values in the Windows registry. They trace registry entries affected by an application and use statistical methods to identify the registry entry that is likely to have caused the problem. Strider compares registry entries on a sick machine against their values on a healthy machine, whereas PeerPressure eliminates the need to explicitly identify a healthy machine by resorting to a large population of machines, under the assumption that most machines have correct states. Yuan et al. [19] diagnose runtime problems by mapping low-level system description (sequences of system calls) to known problems using a Support Vector Machines (SVM) classifier. Chronus [18] tries to find the point in time when a system’s configuration became erroneous. It does so by testing a predicate against past snapshots of system state, and solves the problem by rolling back to the most recent operational state. These systems identify the root cause of a configuration problem but do not automate complex or new configurations. In contrast, ACaMIA automates new and complex configurations that the user wants
to perform but does not know how to perform, or does not want to spend time or effort to figure them out.

NetPrints [6] diagnoses configuration problems in home networks. It collects examples of good and bad network configurations, builds a decision tree, and then determines the set of configuration changes that must happen to change a configuration from bad to good. This approach is successful for fixing configuration in the home network, where the set of possible configurations is relatively small. However, it becomes intractable with complex configuration tasks like those addressed by ACaMIA. For example, PeerPressure [16] applies a similar approach to NetPrints to diagnose Windows configuration problems, but given the complexity of the environment, it focuses on finding a single problematic registry entry.

Mirage [9] addresses configuration upgrade. It clusters machines according to their environment (i.e., OS, runtime libraries, executables, environment variables, and configuration files) and argues that upgrades should be performed in a staged approach by picking few machines in each cluster. Mirage is complementary to ACaMIA, and its clustering approach can be incorporated in the process of creating a generalized trace.

ACaMIA in some ways falls in the area of Programming by Example (PBE). PBE refers to a programming technique where the system records actions performed by a user and produces a generalized program that can be used later in similar scenarios [10]. Past works on PBE however require the intervention of a human programmer to record user actions in a scripting language and parameterize the script to work in various environments [8, 13, 14, 7]. These projects focus on developing high-level scripting languages that are easy to use by an average user [7, 13]. They are application-dependent, (e.g., Chickenfoot enables a user to customize her Web browser), or operate in special research environments (e.g., DocWizard provides walk through documentation for the Eclipse system [7]).

ACaMIA departs from prior work on PBE in three main ways. First, it addresses configuration updates across applications and in typical working environments (e.g., Windows). Second, instead of requiring a human to write automation scripts, it exploits Windows accessibility interface to automate recording and replay. Third, it develops new algorithms that combine traces of user actions to automate a configuration update and eliminate irrelevant actions and personal information.

Finally, there are a few commercial tools that allow a human expert to write a script that automates a specific configuration task (i.e., a macro). These include AutoIt [2], AutoHotKey [11], iMacros [3]. In contrast to these tools, ACaMIA does not require a human to write a macro for each configuration task; By watching a few users perform a configuration task, ACaMIA generates a canonical automation script that enables future users to perform the same task.

3 ACaMIA’S USAGE SCENARIO

Consider, for example, configuring a new email account—a task that occurs for every new student, professor, and staff member in our department. In an attempt to reduce the amount of time that administrators spend dealing with this task, the infrastructure group provides online documentation that walks the users through the configuration. As shown in Fig. 1, the task involves a relatively complex procedure and requires the user to interact with multiple programs (the Web browser, the installer, and the mail program). Experience has shown that many users are unable to complete the task on their own despite the detailed documentation, and eventually seek the administrator’s help.

ACaMIA can change this scenario. ACaMIA watches a few users as they set up their email account and combines their actions to generate a canonical form that enables future users to perform the same task. Specifically, a user who is willing to share her configuration solution with other users in the department, starts by activating ACaMIA’s recording button. She then goes about her configuration procedure following some online documentation like that in Fig. 1, or her own experience. Once she has successfully finished her configuration, she stops the recording and uploads the resulting trace to the configuration server by clicking the upload button in the ACaMIA window. ACaMIA will ask the user whether the configuration was successful, and will upload only configuration traces deemed successful by the user.

When an ACaMIA client uploads a trace, the configuration server asks the user for her name and password. The server accepts solutions only from a user with a valid password and labels each solution with the user’s name. The server also asks the user to label her configuration by picking from a list of known configuration tasks that are of in-

Figure 1—Documentation for configuring IMAP

\[\text{To configure IMAP:}\]

- Download and install the latest version of Thunderbird
- At the Account Wizard - New Account Setup, choose Email Account' click next.
  - Enter your full name as well as your full email address
  - Next, for Server Information, choose IMAP
    - Incoming Server: imap.FooLab.FooUniversity.edu
    - Check ‘Use Global Inbox’
    - Outgoing Server: outgoing.FooLab.FooUniversity.edu
    - Enter your IMAP email name for Incoming and Outgoing User Name
    - For Account Name, enter your full email address, and Click finish
  - Next, you need to configure IMAP
    - From the Thunderbird Menu, Click Tools, then Click Account Settings
      - From the Account Setting menu on the left, choose Server Settings.
      - Under the Security Settings - Use Secure Connection option, choose "SSL " and click "Use Secure Authentication"
      - While still in the Account Settings window, Choose the Outgoing Server from the menu on the left and click edit.
        - Change the Security Authentication to use "SSL"
      - Click OK. Click OK again to save your Account Settings information.
      - Double click the inbox
        - When prompted for a certificate, choose " Accept This Certificate Permanently."
      - You will now be prompted to enter your password.
      - You should now be able to securely send and receive email

• You should now be able to securely send and receive email
terest to the user’s organization, e.g., “Setting your IMAP Account”. If none of the existing labels applies, the user types a new label. The server analyzes the traces to produce a canonicalized automated configuration, which it continuously refines as more traces are uploaded.

Next, the server processes the traces associated with a configuration task to produce a canonicalized automated configuration. This process involves a few important steps.

- **Filter irrelevant information.** A common example is interactions with entertainment software such as music or video while the user is performing the configuration task. These events are irrelevant to the original configuration task and should not be included in the automated solution.

- **Filter user mistakes.** A user may provide a wrong value while performing a particular assignment, which she corrects later. For example, a user who is following the instructions in Fig. 1 may make a typographical error while entering the name of the IMAP server. ACaMIA needs to identify such mistakes and filter them out, otherwise many user inputs may look random and unamenable to automation – e.g., without filtering the wrong server name, ACaMIA may confuse the name of the server as private information though it is the same for the whole department, and hence can be easily automated.

- **Parameterize user inputs.** The server compares the traces to identify which user inputs are private and need to be provided by the user and which are public and can be gleaned from the traces. For the email configuration example, a user’s name and password cannot be entered automatically, while the IMAP server is likely the same for everyone in an enterprise network and hence can be automated or proposed as the default option.

- **Generate if-then automated configurations.** The states of the different machines may vary in ways that affect the set of actions that must be performed to update the configuration. Consider again the example in Fig. 1. Some users may already have Thunderbird installed while others do not. Thus, the ACaMIA server parses the traces for different execution paths, and generates if-then automated solutions.

Once an automated configuration is generated, it is added to a database that users can browse online. Next time a user needs to perform the configuration task, she can download the automated configuration and ask her ACaMIA client to play it. ACaMIA will run the automated configuration and stop only when it needs a private input from the user. For example, an ACaMIA automated solution of the configuration in Fig. 1 would stop only at the two highlighted bullets to allow the user to enter her name, email account, and password.

![Figure 2—Windows, Views, and Objects.](image)

A window may have different tabs, each tab is a view. This figure for example shows a particular view of the “Options” window in Thunderbird. The checkbox labeled “Display emotions and graphics” is an object in this view.

4 **DESIGN AND IMPLEMENTATION**

This section describes how ACaMIA performs its job. We start with a formal model that defines the UI objects that ACaMIA interacts with, how they are navigated, and what interaction capabilities are feasible. We then tie the model to the system implementation based on the UI accessibility interface. Next, we proceed to describe how ACaMIA parses the traces, filters irrelevant and erroneous events, classify user inputs into private and public, and finally automates a configuration task.

5 **SYSTEM MODEL**

Modern user interfaces are fairly complex. Having a formal model enables us to capture ACaMIA’s behavior in an accurate implementation-independent manner. This simplifies reasoning about the design and porting it to different implementations.

5.1 **UI and System State**

The system state is captured in a set of typed state variables $s \nu \in SV$. Each state variable has a value $\text{value}(sv) \in \text{String} \cup \text{Integer} \cup \text{Boolean} \cup \text{list(String \cup Integer \cup Boolean)}$.

Both human users and ACaMIA interact with the system state via the user interface (UI). The UI defines three types of entities: objects, views, and windows. Objects enable a user to view and edit the system state (e.g., a textbox or a checkbox) or to navigate between views (e.g., the Next button). Each view $v$ has a fixed set of UI objects $\text{objects}(v) = \{o_1, ..., o_k\}$. Views maintain the context of an object and also serve as navigation units, i.e., a user moves between views and cannot access an object unless she first accesses the view containing that object. Windows define a tree hierarchy whose root is the desktop. Each window $w$ has a fixed set of views $\text{views}(w) = \{v_1, ..., v_k\}$. We can also identify the window of a particular view $v$ as
w = \text{window}(v)$. The views in a window may refer to different tabs, or a sequence of dialog boxes that a user navigates using the Next and Back buttons. Fig. 2 shows an example of these three entities. The figure shows the “Options” window in Thunderbird. In particular, it shows the “Display/Formatting” view in the “Options” window. The checkbox labeled “Display emotions and graphics” is an object in this view.

We define a function that takes an object in a view and returns the associated state variable $sv = \text{variable}(v, o)$. Note that since objects are only defined within a context (e.g., a textbox is meaningless without a context view), $\text{variable}$ cannot be defined on an object alone without identifying its view. We require that each state variable is associated with a single UI object, i.e., that $\text{variable}(v_1, o_1) = \text{variable}(v_2, o_2)$ only if $(v_1, o_1) = (v_2, o_2)$ or $\text{variable}(v_1, o_1)$ is empty, in which case the object is not associated with any state variable. There is a navigation relation between views. So $v_2 = \text{goto}(v_1, o)$ means that one can move from view $v_1$ to $v_2$ by clicking on button $o$ in view $v_1$. Also, there is hierarchy $\subset$ on windows corresponding to nesting within the user interface. So $w_1 \subset w_2$ if $w_1$ is a child of $w_2$.

5.2 User Actions, Events, and Traces

Users interact with the user interface by performing actions $a \in A$. Examples of actions include setting the text in a textbox to a certain value and checking or unchecking a checkbox. More examples include pressing a button that causes another view such as a dialog box to appear and pressing the OK button in a view that causes the state changes in the view to be written back into the system state. Each action operates on an object and takes place in a view. The combination of an action, an object, a view is an event $e \in E$. Each user interaction session produces a trace $t \in T$, where each trace $t$ consists of a sequence of events $e_1, e_2, \ldots, e_k$. There are several kinds of events:

- **Assign State Events**: Events that correspond to potential assignments of system state (we say potential state assignment because in some cases the assignment may not actually take effect until a later event, such as the user pressing an OK button, causes the state change to be written back into the system state). Each assign state event has the form $\langle\text{assign}, v, o, x\rangle$, which assigns the value $x$ to the state variable $\text{variable}(v, o)$ of object $o$ in view $v$.

- **Augment State Events**: Events that correspond to potential operations that add new entries to system state. An example is a list box operation that adds a new item to a list of items in the system state. Each augment state event has the form $\langle\text{aug}, v, o, x\rangle$, which corresponds to an action that uses the value $x$ to augment the state variable $\text{variable}(v, o)$ of object $o$ in view $v$.

A key difference between assign state events and augment state events is that an assign state event is idempotent and eliminates the effect of all previous assign events. In contrast, the value of a system state variable updated with an augment state event depends on not just the latest event, but also on previous events that also augment the same system state variable.

- **Remove State Events**: This is the opposite of an augment event. An example is a list box operation that removes an item from the list of items in the system state. Each remove state event has the form $\langle\text{rem}, v, o, x\rangle$, which corresponds to an action that removes the item whose value is $x$ from the state variable $\text{variable}(v, o)$ of object $o$ in view $v$.

- **Navigate Events**: During the operation of the user interface only a (typically very small) subset of the set of views is available, i.e., open and enabled for user interaction. At any point during a user interaction session there is a set of available views $Av \subseteq V$. Events (such as opening the Control Panel or selecting one of a set of tabbed windows) that change this set of available views are called navigation events. Each navigation event takes place in the context of an already available view.

Each navigate event has the form $\langle\text{nav}, v, b, a, i\rangle$, where $v$ is the available view in which the navigation event occurs, $b$ is the name of the object whose activation causes the navigate event to occur (for example, a button that causes another view to appear when the user selects it), $a \subseteq V$ is a set of views that the navigation event makes available, and $i \subseteq V$ is a set of views that the navigation event makes unavailable.

- **Commit Events**: Events that cause pending state changes (from state events) to be written back into the system state. Commit events can commit a single object, all objects in a view, or all objects in all view in a window. Commit events of the form $\langle\text{com}, v, b, a\rangle$ means that button $b$ in view $v$ causes the state change of object $o$ in $v$ to be written in the system state. Commit events of the form $\langle\text{com}, v, b\rangle$ cause all of the state changes for all of the objects in view $v$ (these objects are $\text{objects}(v)$) to be written back to the system state. Commit events of the form $\langle\text{w-com}, v, b\rangle$ mean that button $b$ in view $v$ commits the pending states of all objects in all views in the window containing view $v$ (i.e., it commits all objects in all views in $w = \text{window}(v)$).

- **Commit and Navigate Events**: These events commit pending state changes then navigate the system to a new view. They have the form $\langle\text{com-nav}, v_1, b, v_2\rangle$ where button $b$ in view $v_1$ commits all states in view $v_1$, closes view $v_1$ making it unavailable and navigates to $v_2$. An alternative form is $\langle\text{w-com-nav}, v_1, b, v_2\rangle$, where button $b$ in view $v_1$ commits all pending states in all views of the window $w = \text{window}(v_1)$, closes window $w$ and all of its views and navigates to view $v_2$. 


• **Abort Events:** These events abort pending state changes. Similarly to commit events, we can define abort events over an object (i.e., \((\text{abort}, v, b, o)\), a whole view (i.e., \((\text{abort}, v, b)\)), or a whole window (i.e., \((\text{abort}, v, w)\)).

• **Abort and Navigate Events:** These events abort pending state changes then navigate the system to a new view. They have the form \((\text{abort}&\text{nav}, v_1, b, v_2)\) where button \(b\) in \(v_1\) aborts all states in view \(v_1\), closes view \(v_1\) making it unavailable and navigates to \(v_2\). An alternative form is \((w\text{-abort}&\text{nav}, v_1, b, v_2)\), where button \(b\) in view \(v_1\) aborts all pending states in all views of the window \(w = \text{window}(v_1)\), closes window \(w\) and all of its views and navigates to view \(v_2\).

### 5.3 Mapping Model to Accessibility Interface

We implement ACaMIA using the Microsoft Active Accessibility interface. ACaMIA obtains all information about the structure of the user interface from the accessibility interface and the OS’s window manager. In particular, ACaMIA obtains from the OS information about low-level inputs such as mouse-clicks and keyboard presses, as well as the window handle for where the event occurred. ACaMIA uses this information to query the accessibility interface for the current view of the window and its objects and their states (i.e., it queries the accessibility interface for the widget that has keyboard focus or where the mouse click has occurred).

**Naming Objects and Views:** The first challenge for ACaMIA is that UI objects, views, and windows have no global names that work across machines or even across reboots of the same machine. To address this challenge, ACaMIA defines its own naming system that carries the visual context. Specifically, ACaMIA refers to a UI object by concatenating its role and name as returned by the accessibility interface. For example, the checkbox in Fig. 4 has a role=checkbox and name="Display emotions as graphics". Clearly, this object name may not be unique and is meaningful only in the context of the view in Fig. 4. ACaMIA carries the view of an object as a context for any action that involves the object. ACaMIA gives each view a primary name and a secondary name. The primary name has to be the same for two views to be the same. It is created by taking a hash of the names of the permanent objects in the view. Said differently, we name a view using the collection of its objects. We exclude a few objects which may not always appear in a view, such as scroll bars (which may disappear if the user enlarges the window), tool bars, and list items. The secondary name is used only when the primary name matches multiple views. The secondary name is a concatenation of the default values in the view (like the title of the window) and the non-permanent objects. Finally, we refer to a window by the set of views that it contains. Note that do not need to have window names that work across machines. We use the window concept to define the relation between views. Thus, we only maintain the relation between views (e.g., a set of views is a child of another set of views).

**Recording Events:** As stated earlier, ACaMIA tracks mouse clicks and keyboard strokes, and queries the accessibility interface to retrieve the view and object that received the mouse click or the keyboard focus. ACaMIA uses this information to create an event trace. The accessibility interface does not classify an event into assign event, navigate event, commit event, etc. ACaMIA has to label each event by tracking how the associated view and object state have changed as a result. It uses the following rules:

• **Object state (but not number of items) changes and view does not change → Assign state event.** This event occurs when a user types in a textbox or checks/uncheks a checkbox, etc.

• **List object has an extra item in it and view does not change → Augment event.** Examples include adding an email account to an account list, or adding a new font to a font list, etc.

• **List object has one fewer item and view does not change → Remove event.** This is the opposite of the previous event.

• **Neither object nor view changes, and object type is not inert → commit event.** We have encountered cases where a button click does not make any visually observable change. These buttons are typically associated with a hidden state, like rebooting a process, or flushing a cache, etc. In this case, ACaMIA cannot read the value of the state variable associated with the button if any. In the absence of further information, ACaMIA tries to preserve the event in case it is necessary for the configuration task. Hence, it labels it as a commit event associated with a hidden state variable that ACaMIA cannot read.

• **View changes and object is the Cancel or Abort button → Abort and navigate event.** While navigation and assignment events are easy to identify from the visual context, commit and abort events are more subtle. Our current implementation uses a heuristic that is based on the name of a button (e.g., cancel) and common navigation methods to guess whether a particular button commits pending states. In particular, Cancel and Abort buttons are mapped to \((w\text{-abort}&\text{nav}, v_1, o, v_2)\), i.e., they abort all pending state for the whole window that contains the current view.

• **View changes to a view in the parent window, and object is neither Cancel nor Abort → Commit and navigate event.** Whenever a user closes a child view and goes back to its parent window, she either aborts her actions or commit them. If she went back to the parent window by pressing a Cancel or an Abort button, then the previous rule applies and all pending states in all views in the current window are aborted. Otherwise all pending
state in all views in the current window are committed.

- View changes to a view that is not in the parent window and object is not the Cancel button → Navigate event. This event happens when a user moves between tabbed views in the same window, follows the Next and Back buttons in wizards, or moves between different application windows. These cases however differ in whether the previous view is still available or not. For example, moving between tabs in the same window closes the current view and opens a new one, while moving between completely different applications keeps both the current and new views available.

Replaying Events: ACaMIA merges the traces to create a canonical automated trace that applies the configuration on new machines (see below). Once such a trace is available, replaying it on a new machine is relatively simple. At each step, ACaMIA queries the accessibility interface for all visible views. It uses the view primary name – and when needed the secondary name – to find the relevant view for the current event. It parses that view to find the relevant object, and applies the event to the object. After it applies an event, ACaMIA waits for a short period then queries the accessibility interface to check the visual outcome of the applied event. If the currently accessible views enable the next event (e.g., the checkbox needed to be checked is in an accessible view), ACaMIA proceeds to apply the next event. Otherwise, ACaMIA waits for a short period and checks again whether the view of the next event is available. It retries for a few times and aborts if unsuccessful at finding the next view. Depending on CPU load, there might be a short delay before a UI event takes place. To deal with this issue, ACaMIA exploits that the UI guides the user to which events are available. Specifically, only those events that involve available views (i.e., a view that is open and available for user interaction) can take place at any point in time. Thus, ACaMIA needs to wait until a view becomes available to perform an action on one of its objects. Information about which views are available can be obtained via the accessibility interface.

5.4 Filtering Irrelevant Activities

During the collection of traces that perform a given configuration update, users may perform irrelevant activities such as listening to music or randomly exploring the UI. These irrelevant activities cause spurious events to occur in the collected trace. ACaMIA aims to detect these irrelevant activities and eliminate any events associated with them.

At a high-level, ACaMIA takes a standard approach to address this problem, namely it assumes that any given irrelevant activity is uncommon and does not repeat across many traces. For each view, ACaMIA computes the percentage of traces that include that view at least once (i.e., they have events occurring in that view). ACaMIA attempts to exclude views that appear sufficiently rarely, as long as later views deemed essential do not have some direct dependency on these views (e.g., a user followed an unusual path to navigate to an essential dialog). ACaMIA also attempts to exclude views that can be shown not to include any commit events and those associated with OS processes otherwise identified as irrelevant to the task at hand. The navigational events that invoked these views are removed, along with any update and commit events that might’ve occurred within it.

5.5 Filtering Redundant or Erroneous Assignments

Users may make mistakes during a configuration activity, then go back and correct the mistake. For example, a user may enter a wrong value, move forward to the next set of windows, realize her mistake, go back to the corresponding window, correct the error, then continue her configuration task. ACaMIA parses the recorded traces to learn when an assignment, augment, or delete event, is committed and filters erroneous or redundant such events. Below we describe the detailed algorithm.

The algorithm processes the trace twice: once backwards to remove all assignment/augment/delete events except the last committed event, and once forward to remove all corresponding commit events except the commit event immediately following the last committed assignment/augment/delete event.

Backward Pass: The backward pass maintains two sets of view, object tuples: a pending set $P$ and a committed set $C$. Both sets are initialized to the empty set. The algorithm then performs the following actions for each event as the events are encountered during the backwards pass over the trace.

- Commit Events: For a commit event $e = \langle \text{commit}, v, o \rangle$, the algorithm checks to see if $\langle v, o \rangle \in C$. If not, it adds $\langle v, o \rangle$ to the set $P$ of pending objects. Otherwise, it does nothing.
  For a commit event $e = \langle \text{commit}, v, b \rangle$, the algorithm checks all $o \in \text{objects}(v)$. If $\langle v, o \rangle \notin C$, the algorithm adds $\langle v, o \rangle$ to $P$. Otherwise, it does nothing. Similarly, for all events of the form $\langle \text{commit}, v, b \rangle$, the algorithm checks all $o \in \text{objects}(v_i)$ for any $v_i$ that is a view in $w = \text{window}(v)$.

- Assign and Navigate Events: are treated similarly to commit events.

- Assign/Augment/Delete Events: Assign, delete, and augment events are all treated similarly. Thus, we limit our description to assign events. For an assign event $e = \langle \text{assign}, v, o := x \rangle$, the algorithm checks to see if $\langle v, o \rangle \in P$. If so, it adds $\langle v, o \rangle$ to $C$ and removes $\langle v, o \rangle$ from $P$. Otherwise, it removes $e$ from the enclosing trace. This is because either a subsequent event will overwrite the effect of $e$ or there is no commit event for $e$ following $e$ in the trace.

- Navigate Events: Navigation events have no effect on
effect on pending and committed assignments. Note that while navigation may cause a view \( o \) become unavailable it does not mean that the assignments have been aborted. An example occurs when a user open a child window, the resulting view will sit on top of the parent window making all parent views unavailable for user interaction. However pending assignments in the parent window are not lost.

- **Abort Events:** Abort events act in the opposite way of commit events, i.e., they make the algorithm remove the corresponding tuple \( \langle v, o \rangle \) from the pending set \( P \), causing these uncommitted assignments to be lost. For example, the event \( e = \langle \text{abort}, v, b \rangle \), causes the algorithm to remove all tuples that involve objects in view \( v \), i.e., \( \langle v, o \rangle \), from \( P \). The same applies for aborting a single object assignment or a whole window.

- **Abort and Navigate Events:** are treated similarly to abort events.

At the end of the backward pass, there is at most one assign/augment/delete event \( \langle v, o \rangle \) in the trace for each \( v, o \) pair. Moreover, there is also a corresponding commit event \( e \) after \( \langle v, o \rangle \) in the trace. There may also be other redundant commit events for \( \langle v, o \rangle \). The forward pass removes these redundant commit events.

**Forward Pass:** The forward pass maintains two sets of view, object tuples: a pending set \( P \) and a found set \( F \). Both are initialized to the empty set. The algorithm performs the following action for each commit event that it encounters during the forward pass over the trace:

- **Object Commit Events:** For each object commit event \( e = \langle \text{commit}, v, o \rangle \), the algorithm checks if \( \langle v, o \rangle \in P \), i.e., there is an assignment to the corresponding tuple that is pending to be committed. If so, it removes \( \langle v, o \rangle \) from \( P \) and adds \( \langle v, o \rangle \) to \( F \). Otherwise, it removes \( e \) from the trace — \( e \) is redundant because there is no pending assign state event for it to commit. Either a previous commit event committed the last committed assign event for \( \langle v, o \rangle \) or the last committed assign event for \( \langle v, o \rangle \) has yet to appear in the trace.

- **View and Window Commit Events:** Commit events that commit a whole view or a whole window are reduced to object commit events for all objects in the corresponding view or window. They are then processed as described above.

- **Commit and Navigate Events:** Commit and navigate events are decomposed into commit events, which are treated as described above, and navigate events which are ignored during the forward pass.

- **Assign/Augment/Delete Events:** For each assign/augment/delete event, the algorithm adds the affected tuple \( \langle v, o \rangle \) to \( P \).

Finally, the forward pass can be augmented to remove redundant abort events in a similar manner to how we removed redundant commit events.

### 5.6 Parameterize User Inputs

Now that ACaMIA has removed erroneous and unnecessary assign/augment/delete events, it can compare the traces to identify which user inputs are private and need to be provided by the human, and which are public and can be gleaned from recorded traces.

For each object in each view, ACaMIA parses all traces to find all unique values that were given to that object via assign/augment/delete events. It also finds the number of traces that have each value. It creates a table that maps \( \langle v, o \rangle \) to \( \text{list}(\text{tupleaction}, \text{value}, \text{occurrences}) \). For example, say that \( \langle v, o \rangle \) refers to a view and textbox object where the user types the name of her printer. The corresponding table entry may look like \( \langle \text{assign}, \text{printer}\_\text{floor}1, 8 \rangle, \langle \text{assign}, \text{printer}\_\text{floor}2, 2 \rangle \), which indicates that, in 8 traces, the user typed “printer_floor1” when asked for the name of her printer, and in 2 traces, the user typed “printer_floor2”. Another entry could refer to a specific checkbox and may have the value list(\( \langle \text{assign}, \text{checked}, 10 \rangle \)), which means that in all 10 traces the user checked this checkbox.

Next, ACaMIA parses the recorded trace to parametrize assign/augment/delete events, i.e., to decide whether it can automate them or needs to take user input. Without loss of generality, let us focus on assign events. For each such event, ACaMIA looks up the table entry for the associated tuple \( \langle v, o \rangle \) and labels the event with one of the following labels:

- **Auto Enter:** The is the case if the tuple \( \langle v, o \rangle \) is assigned the same value in all recorded traces. Any assign event to such tuple will be performed automatically by assigning the value from the traces.

- **User Enter:** The is the case if the tuple \( \langle v, o \rangle \) takes unique values across traces. Hence, ACaMIA has to ask for user input when performing the assign event.

- **Possible Auto Enter:** The is the case if the tuple \( \langle v, o \rangle \) shows a few common values that occur repeatedly across traces (e.g., a printer name). When ACaMIA needs to assign a value to such tuple, it lets the user select one of the previously observed values or enter a new value.

Note that events that involve passwords are always treated as user-enter. Specifically, since ACaMIA records passwords at the level of the graphical user interface (which almost invariably obscures passwords), it does not record the actual passwords. Furthermore, whenever it sees such obscured entry in a trace it always marks the corresponding assign event as user-enter.

### 5.7 Filtering Unnecessary Navigation Events

Next, ACaMIA removes unnecessary navigation events. These events may result from users making navigation mistakes, or they may be left from the previous section where
we removed the unnecessary assignment events but not the
events that navigated to/from the corresponding views. Re-
moving unnecessary navigation events is not a prerequisite
for correct configuration. However, it simplifies automation
and may save significant execution time.

Our approach to removing unnecessary navigation
events relies on two ideas. First, we can remove any se-
quence of only navigation events as long as we can nav-
igate immediately from the available views before the se-
quence to the next view after the sequence. Second, in-
stead of navigating to a particular view multiple times, each
time changing the state of a subset of its objects, we would
like to navigate to a view only once, make all object state
changes in that view, and leave the view. Thus, our second
optimization tries to delay all state changes to the last time
we enter a view, and removes all unnecessary navigation to
that view. Below we describe an algorithm that implements
these ideas.

The algorithm operates with the aid of four two-
dimensional tables. Each column of each table corresponds
to a time step in the trace. In these tables, only navigation
events cause transitions between time steps (i.e., transitions
between columns). Each row of each table contains infor-
mation for a single view. Each entry in each table therefore
contains information about a given view at a given point
during the trace.

- **Available Views**: The table $\text{available}_{i,j}$ : Boolean records whether view $v_i$ is available (for user interac-
tion) at time step $j$.
- **Event Lists**: The table $\text{events}_{i,j}$ : list(event) contains the list of events other than navigation events performed
in view $v_i$ at time step $j$.
- **Necessary Views**: The table $\text{necessary}_{i,j}$ : Boolean records whether a given view $v_i$ is necessary at step $j$.
There are two ways for a view $v_i$ to be necessary at step $j$: either some events occur in $v_i$ during time step
$j$, or $v_i$ presents a common view across traces and time
step $j$ is the first time step when $v_i$ is available. Com-
mon views are identified per application. In Firefox, we have
described how we associate a view with its application (e.g., a Firefox view). A view is a common view if it appears at least once in all traces that run the cor-
responding application. We deem such common views as
necessary and ensure that they stay in the automated
version.
- **Navigation Events**: The table $\text{navigations}_{i,j}$ : event contains a single row that refers to the navigation event
that caused the transition from column $j$ to column $j+1$
in the other tables.

5.7.1 Initialization step

Given a trace, the algorithm builds the above four ta-
bles by simulating the execution of the trace forwards in

time. The table construction algorithm maintains a current
set of available views $A$ and the current time step $t$. When it
processes a navigation event $e$ it sets column $t$ of available
based on $A$, sets navigations, to $e$, updates $A$ based on how
$e$ affects the views’ availability, then increments $t$. When it
processes an event $e$ that is not a navigation event, it adds $e$
to the list of events stored in $\text{event}_t$, where $e$ occurs in view
$v_i$. If $e$ is the first event in this list, it sets $\text{necessary}_{i,j}$
true.

5.7.2 Event re-location step

The algorithm next moves all non-navigational events
that occur in view $v_i$ at any time step to the last available
occurrence of $v_i$, and updates the event table. It also up-
dates the necessary table to reflect the move – i.e., some
earlier occurrences of a view may become unnecessary be-
cause their events were moved to later occurrences.

The goal of this step is to bring all of the events that
occur at view together with the hope of rendering earlier
occurrences of the view unnecessary and hence potentially
removable from the final trace.

5.7.3 Removing Redundant Navigation Events

Consider any contiguous collection of time steps
$t_1, ..., t_n$. It is possible to remove the navigation events for
time steps $t_2, ..., t_n$ if the following conditions are met:

- **Necessary View Condition**: None of the view instan-
ces in columns $t_2, ..., t_{n-1}$ is necessary, i.e., $\text{necessary}_{i,j}$ is false for all $i$ and $t_1 < j < t_n$. The ratio-
nale is that all of these view instances will be removed when the navigation events are removed, so they must
not be necessary.

- **Available View Condition**: The set of available views
at time step $t_1$ is a superset of the set of available views
at time step $t_n$, i.e., $\text{available}_{i, t_n}$ implies $\text{available}_{i, t_1}$ for
all $i$. The rationale is that the navigation events after step
$t_n$ will operate starting from the available views from
time step $t_n$. After removing the intermediate navigation
events, all of these available views will still be available
because they were open at time step $t_1$, the new time
step from which the successive navigation events will
start.

The algorithm that removes redundant navigation
events then simply processes the time steps in reverse or-
der to find maximal sequences of navigation events that it
can remove. The algorithm itself processes each time step
in turn, searching backwards from that time step to find a
maximal sequence of time steps that satisfy the two con-
ditions above. When it finds a non-empty maximal sub-
sequence $t_1, ..., t_n$, it removes columns $t_2, ..., t_n$ from all four
tables. This removal corresponds to removing the naviga-
tion events from $t_1, ..., t_{n-1}$ — in effect, column $t_1$ replaces
columns $t_1, ..., t_n$. As part of this removal, the algorithm ap-
ends any events from time sequence $t_n$ to the end of the
corresponding list of events in the time sequence \( t_i \). Specifically, it sets \( \text{events}_{t_i} \) to \( \text{events}_{t_{i+1}} \), \( \text{events}_{t_{i+2}} \), etc., for all \( i \), and update the necessary table to reflect the move. Note that there are no events in time steps \( t_2, ..., t_{n-1} \) (other than the removed navigation events).

When the algorithm terminates, it traverses the tables above to build a new trace. The removed navigation events do not appear in this trace and the corresponding navigation steps will therefore not take place when the trace is replayed.

5.8 Generating a Canonical Automated Trace

By now, ACaMIA has filtered irrelevant activities and unnecessary assignments. It has also parameterized events that involve state change (i.e., assign, augment, and delete events) and hence it knows which of them can be performed automatically and which requires user input. Furthermore, it has removed as many as it can of the unnecessary navigation events. Thus, at this stage, each trace has been transformed to a sequence of parameterized actions and, in principle, can be used to automate the required configuration task on a new machine.

The states of the different machines, however, may vary in ways that affect the set of events required to perform the configuration. Consider for example an update that requires downloading a particular software package using a Web browser. If our automated configuration uses FireFox but the machine has only IE, the configuration will fail unnecessarily. More generally, different machines may have different relevant states which require them to follow different paths to perform the configuration.

To address this issue, ACaMIA parses the traces for different execution paths, and generates an if-then automated solution. First, ACaMIA marks each parameterized trace by the set of used programs (i.e., the executables). When automating a configuration on a particular machine, ACaMIA starts by taking a System Restore snapshot in case it needs to restore the machine to its previous state. ACaMIA then tries to execute the shortest parameterized trace that involves only programs that the machine already has (unless the program is installed as part of the configuration). Second, if the automated execution fails to complete, i.e., at some point ACaMIA fails to find the view and object required for the next event, ACaMIA tries a different automated trace. Again, ACaMIA picks a trace that does not require any program that the machine does not have but also a trace that does not have the view where the previous automation failed. ACaMIA tries a few traces. If the automation fails again, ACaMIA probes the user for whether she wants to restore her old system state or continue the incomplete configuration manually.

6 LIMITATIONS

The current design of ACaMIA has rather limited support for tasks that require the user to navigate the file system. Specifically, ACaMIA uses the recorded traces to discover how to navigate among views. The underlying assumption is that there is some commonality in the views involved in a particular task across different traces, and hence we can learn a valid navigation path by observing only a few traces. This assumption does not apply to navigation through a user’s file system, however, because different users may have completely different directory structures and file names. For example, consider a task that involves installing a new software package. Different users may download the package, navigate their file browser to a random directory, and store the package there. A machine that later runs the automated configuration produced from this trace may have none of the directories used in the trace. Thus, there is no way to navigate on the file system of a new machine in a manner similar to any of the recorded traces. Currently, ACaMIA deals with this problem by requiring users to download all software packages required for configuration updates into a specific Download directory. Note that this limitation is only for file navigation done directly by the user, and does not include file storage performed directly by a program, which is done in fixed locations. None-the-less, it would be beneficial to extend the model and implementation to have more flexible support for the file system.

ACaMIA’s design is also limited to applications that have a reasonable support of the accessibility interface. Our experience indicates that all major Microsoft applications (Windows, MS Office, IE, Visual Studio, etc.) as well as most other popular applications (e.g., FireFox, Thunderbird, Matlab, Open Office, Adobe Acrobat) have sufficient accessibility support to enable a tool like ACaMIA.

The U.S. government has taken a strong stance on software accessibility, ensuring that all products which are sold to them provide at least a minimum level of accessibility support. For this reason and others, both commercial vendors and the open software community have made a commitment to support accessibility. Lastly, note that smaller vendors who do not directly support Accessibility are often building their applications, or at least their configuration interfaces, directly using widgets from Windows’s standard toolkit. Since the standard widget toolkit comes with basic Accessibility support built-in, this provides sufficient access to configure most of these types of applications as well.

A second limitation involves out-of-band channels. For example, consider the scenario where a user wants to install a software package but cannot complete the installa-

1Some of these applications would benefit from more robust implementation of certain features, but they all provide the main accessibility functionalities. For example, Acrobat does not name its UI objects, so we fall back on identifying a UI object by its role and a hash of its neighbors in the view.
tion without some information that she receives via email. The likelihood of automation success depends on the complexity of extracting the needed information from the email. For example, if all users receive the exact same email (e.g., there is one license for the whole site), ACaMIA will recognize this relevant state as auto-enter and copy it from the traces. If the email always has the same structure except for a user name and password, then, in principle, ACaMIA may be extended to take file diffs and handle this scenario but our current implementation does not handle it. As the email structure becomes more customized it will be harder to automate the interaction between the installer and the email program.

We also assume that a user can identify a successful configuration and uploads only successful configuration recordings. If a user uploads an unsuccessful configuration, which the server uses in generating an automated configuration, the resulting automation may fail. Note that this risk is similar to the risk a user takes whenever they seek a solution to their configuration problems either on-line or directly from others. As an automated system, however, we can take advantage of Microsoft’s built in System Restore state restoration mechanism and create a system snapshot before applying a ACaMIA solution, then restoring the system to that state if the user declares the configuration unsuccessful.

Finally, sharing configurations with other users raises security and privacy issues. ACaMIA’s current design targets a single organization or an enterprise network, where users know and trust each other. ACaMIA accepts configuration traces only from users with valid passwords and provides automation services to a user after validating her password. Also, ACaMIA can be easily augmented to allow the user to choose a subset of the recorded traces to produce the automated configuration. This enables a consumer of an automated configuration to choose only traces from people she trusts. Finally, since the UI obscures all passwords, ACaMIA naturally inherits this property.

7 EVALUATION

We evaluate ACaMIA on actual configuration tasks that have been documented by the system administrators in the computer science lab in our university. We compare its performance to a human user population that executes the same tasks using the online documentation.

(a) Tested Configurations. Fig. 3 describes the tested configurations. The description in the figure is an anonymized version of the online documentation provided by our lab infrastructure group to help CS students, professors and staff members in performing these configurations. As can be seen from the figure these are real-world configurations problems. Each of them involves multiple steps and requires interaction with multiple applications.

(b) User Population. Our evaluation employs 12 human users, all of which are graduate students in computer science. The users have relatively-diverse backgrounds which span: systems, artificial intelligence, programming languages, and computer networks. They are the natural audience for the online documentation in Fig. 3.

(c) Setup. We performed the experiments in two sessions. Before each session, we took a system restore point on each machine to ensure that we can restore the machine to its original state. Once the session ends we collect the resulting traces as well as some statistics, then restore the machine to its original state. In the first session, the users were asked to perform the configurations manually. They were directed to the online documentation for each task and given unlimited time to finish the configurations (but less than one day). The users performed these configurations on their own laptops. All used laptops run Windows XP but are otherwise customized by their owners. While the users performed the configuration manually, ACaMIA recorded their UI interactions. ACaMIA processed the recorded traces offline as described in §4. It generated a canonicalized automated configuration for each of the tested tasks. In the second session, the same users ran the automated configurations providing inputs only when ACaMIA probes them.

One limitation of our experimental setup is that the manual configuration session happens before the ACaMIA session. First, we need a few manual traces to generate an automated configuration, and hence it would not be feasible to reverse the session order. Second, we believe that the order has a very limited impact on the performance of ACaMIA since once a configuration is automated the human users have little to no interaction with the execution.

7.1 Configuration Successes vs. Failures

Our first result reports the number of users who failed – or succeeded – in completing each of the five configuration tasks. On the one hand, this allows the reader to gain insight to the relative difficulty of the tested configurations tasks. On the other hand, it shows how a tool like ACaMIA can reduce the need for administrator intervention.

Table 4 reports which user completed which task. The table shows that the first three tasks (IMAP configuration, double-sided printing, and joining the active directory) are harder from a user perspective than the last two. The IMAP configuration in particular, included a large number of steps and as a result users tended to make mistakes and go back to correct them. Some users lost track and could not finish the configuration. Enabling double-sided printing was fairly bimodal; about half the users finished the task quickly with no unnecessary actions, while others took a long time and could not complete the configuration successfully. A common problem has been that some users, though given the documentation, did not follow it because they felt that they know how to perform certain tasks. ACaMIA on the other hand, has successfully automated all five configura-
Enabling Double-sided Printing for Windows

- Depending on the printer model, go to Start > Settings > Printer and Faxes > select the printer > Properties > configuration. Check "Duplex" and "MBP".
- OR go to Device settings, and under "Duplex Unit" pick Installed.
- Once you enable the duplex option, the usual "flip on long edge" or "flip on short edge" option shows up in the printer properties.

Join Your Computer to the Active Directory Domain

- Install Kerberos as described in Installing Kerberos For Windows
- Install OpenAFS as described in Installing OpenAFS
- Download and run the FooLab Kerberos REG file. This will create entries for the FooLab Kerberos realm in the Windows Registry.

Joining the domain

- Open the My Computer properties window (Right click My Computer and select Properties from the menu)
- Select the Computer Name tab and click Change.
- Click Domain and type "ad.FooLab.FooUniversity.edu".
- Click more. Type "csail.mit.edu" as primary DNS suffix. Check box "Change Primary DNS suffix when Domain membership changes".
- Press OK. OK again. You will be asked for a Username and Password for Active directory. Type the username "adjoin" and the password "adjoin". Once you enter the password, you will get a dialog box saying "Welcome to ad.FooLab.FooUniversity.edu domain." Restart the machine.
- You should now be able to login using your FooLab Kerberos account and password using FooLab.FooUniveristy.EDU as domain name.
- Download and Install Windows XP support tools (Optional) for command line tools such as KLIST, KSETUP etc.

Firefox Certificate Problem

Issue
When navigating personal-certificate-protected FooLab websites, Firefox stops repeatedly, asking you to manually "choose a certificate to use as identification."

Cause
The Firefox developers changed the default certificate setting from "select automatically" to "ask every time" because some countries issue certificates to all of their citizens, and this would allow people in those countries to be tracked, by name and ID number, without their consent.

Both the impact (only your FooLab email address and full name are stored in your certificate) and probability (an attacker would have to specifically target and request FooLab certificates, a rather small population) of this information disclosure vulnerability are quite limited for FooLab users.

Resolution
To restore the previous Firefox behavior, open the Firefox preferences dialog (depending on your OS, this is Tools/Options, Edit/Preferences, or Firefox/Preferences) and select the "Advanced" pane, then the "Encryption" tab then change the setting back to "Select automatically."

Install VirusScan Detector

- Install VirusScan Enterprise from: http://web.FooUniveristy.edu/software/
- When asked if you should update now, do update now.

(d) Install VirusScan

(e) IMAP configuration; documentation is in Figure 1

Figure 3—Online Documentation for Tested Configuration Tasks. The figure shows the online documentation provided by the infrastructure group in the CS lab in our department. The text is anonymized and figures are removed for anonymization and space reasons. Note that the documentation for the fifth task (IMAP configuration) is already given in Figure 1.

<table>
<thead>
<tr>
<th>ACaMIA</th>
<th>User 1</th>
<th>User 2</th>
<th>User 3</th>
<th>User 4</th>
<th>User 5</th>
<th>User 6</th>
<th>User 7</th>
<th>User 8</th>
<th>User 9</th>
<th>User 10</th>
<th>User 11</th>
<th>User 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAP Configuration</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Double-Sided Printing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Active Directory</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Virus Scan</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Fix Firefox Certificate Problem</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 4—Configuration Successes and Failures. Tables shows that multiple users failed to perform some configuration tasks despite using online documentation. All users succeeded with ACaMIA.

7.2 ACaMIA’s Impact on Reducing User Inputs

The success of an automation tool is perhaps best measured in terms of how much it reduces the need for inputs from a human user. Thus, for each configuration task and for each user, we compute the number of human actions performed during the configuration. An action is any input
the user provides to a UI object. Examples include typing in a textbox, checking a checkbox or a radio button, clicking on a menu item, etc.

Fig. 5 shows a bar graph of the average number of human actions per task, both during manual configuration and while running ACaMIA's automated configuration. The average is taken over all 12 users, and the thin lines show the minimum and maximum across users. The figure shows that ACaMIA significantly reduces the need for human involvement in the tested configuration. Specifically, in the manual mode, on average, the human users had to perform between 8–40 actions per task. ACaMIA reduced this number to 0–7 actions.

ACaMIA gets this gain by requiring human intervention only when the task needs user-specific inputs, such as name, password, or account information. Note that IMAP configuration requires the user to enter her name, email account, and password. ACaMIA requires 7 user inputs because the email program asks the user to enter her name and her account three times. The current version of ACaMIA cannot tell that these are all the same, and hence the user need not be probed multiple times. However, even without this additional optimization, ACaMIA has significantly decreased the number of user actions by as much as $4 \times$. The only task where ACaMIA requires human intervention for a global input is active directory, which requires a password that is shared among all users. However, since most GUIs identify private information such as passwords, ACaMIA explicitly does not automate the entry of this private input. The other three tasks do not require any user private information, and ACaMIA has performed them automatically without any user input.

7.3 ACaMIA’s Impact on Reducing Configuration Time

Another benefit of ACaMIA is its ability to reduce the overall time required to complete a configuration, which adds to better user experience. Fig. 6 shows the average time consumed by each configuration task, both for the case of manual configuration and ACaMIA’s automated configuration. The average is taken over all 12 users. The thin bars show the minimum and maximum values. The figure reveals that ACaMIA is effective at reducing the overall configuration time, particularly for complex tasks that tend to take longer time. We note that ACaMIA could not reduce the time needed for installing the VirusScan because most of this time is spent on the download itself.

7.4 How Effective is ACaMIA at Filtering Irrelevant Actions?

ACaMIA’s design tries to filter out any unnecessary actions/events that do not help in performing the desired configuration task. Here we check that our attempt to remove such actions is successful. Fig. 7 shows the average number of actions involved in performing each of the configuration tasks. The figure also shows the maximum and minimum number of actions across all users. These are all actions, i.e., both automated and user performed. The figure shows that despite being completely automated, ACaMIA performs significantly fewer actions than the average human user. ACaMIA performs significantly fewer actions than the average user. ACaMIA.

Figure 5—ACaMIA’s impact on reducing user actions. The figure shows the average number of user inputs per task, both for manual configuration and a ACaMIA automated configuration. ACaMIA automated configuration probes the user only when it needs private information such as a password, a username, or an email account. ACaMIA.

Figure 6—Configuration time with and without ACaMIA. The figure shows the average amount of time per configuration task as well as the min and max values. It reveals that ACaMIA can improve user experience by speeding up configuration updates.

Figure 7—ACaMIA is effective at filtering unnecessary actions. The figure plots the sum of automated and user actions average across users, both for ACaMIA and manual configuration. It shows that despite being automated, ACaMIA performs a configuration task using significantly fewer actions than the average human user. ACaMIA.
<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Used by ACaMIA's Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE only</td>
<td>Used IE</td>
</tr>
<tr>
<td>Broken IE, Functional FireFox</td>
<td>Tried IE, Failed to complete, moved to FireFox</td>
</tr>
</tbody>
</table>

Figure 8—ACaMIA customizes an automated configuration depending on the state of the machine on which it executes the configuration.

7.5 ACaMIA’s Conditional Execution

As described earlier, the state of the different machines may vary in ways that affect the set of events required to perform a particular configuration. ACaMIA deals with this issue with an if-then execution style. In this section, we want to check that ACaMIA can customize the automated configuration according to the state of the machine. To experiment with this issue, we run ACaMIA’s automated installation of VirusScan on three types of machines: 1) a machine that has IE uninstalled and FireFox installed, 2) a machine that has only IE, and 3) a machine that has a broken version of IE and a functional version of FireFox. Note that installing VirusScan requires a Web browser. Thus, we would like ACaMIA to use the working browser on each machine.

Table 8 shows the results of running ACaMIA’s automated configuration on these three machine types. As expected, ACaMIA checked for the required programs and discovered that it needs to run an automation trace that involves IE on the IE-only machine, and FireFox on the FireFox-only machine. ACaMIA did not know that the IE version on the third machine is broken. Thus, it tried to automate the installation using IE. Since IE is broken, ACaMIA noticed that it could not find the next view in its trace events. It waited a bit, then switched to an automation option that does not have the failed IE-view. Since the machine has FireFox, it picked an automation option that uses FireFox to download VirusScan. This caused the configuration to succeed.

8 Conclusion

ACaMIA is a tool for automating desktop setup and configuration. It requires no kernel modifications and works across various applications. Our experimental results show that ACaMIA is effective in automating complex tasks that many users fail to complete even in the presence of online documentation. Further, ACaMIA achieves this performance without requiring the administrator to write any automation scripts. It simply watches a few users as they interact with the GUI to perform the desired task. ACaMIA can then generalize the users’ actions and play them back to the GUI to automate the desired task on a new machine. Thus, ACaMIA advances the-state-of-the-art in how organizations deal with desktop configuration updates. It can save an organization from spending an excessive amount of resources on administrative support and reduce user frustration.

REFERENCES


---

5We made Windows uninstall IE so that it is not in the program list.