Enabling Configuration-Independent Automation by Non-Expert Users

Nate Kushman
Massachusetts Institute of Technology

Dina Katabi
Massachusetts Institute of Technology

Abstract

The Internet has allowed collaboration on an unprecedented scale. Wikipedia, Luis Von Ahn's ESP game, and reCAPTCHA have proven that tasks typically performed by expensive in-house or outsourced teams can instead be delegated to the mass of Internet computer users. These success stories show the opportunity for crowd-sourcing other tasks, such as allowing computer users to help each other answer questions like "How do I make my computer do X?". The current approach to crowd-sourcing IT tasks, however, limits users to text descriptions of task solutions, which is both ineffective and frustrating. We propose instead, to allow the mass of Internet users to help each other answer *how-to* computer questions by actually performing the task rather than documenting its solution.

This paper presents KarDo, a system that takes as input traces of low-level user actions that perform a task on individual computers, and produces an automated solution to the task that works on a wide variety of computer configurations. Our core contributions are machine learning and static analysis algorithms that infer state and action dependencies without requiring any modifications to the operating system or applications.

1 Introduction

Computer systems are becoming increasingly complex. As a result, users regularly encounter tasks that they do not know how to perform such as configuring their home router, removing a virus, or creating an email account. Many users do not have technical support, and hence their first, and often only, resort is a web search. Such searches, however, often lead to a disparate set of user forums written in ambiguous language. They rarely make clear which user configurations are covered by a particular solution; descriptions of different problems overlap; and many documents contain conjectured solutions that may not work. The net result is that users spend hours manually working through large collections of documents to try solutions that often fail to help them perform their task.

What a typical user really wants is a system that automatically performs the task for him, taking into account his machine configuration and global preferences, and asking the user only for information that cannot be automatically pulled from his computer. Today, however, automation requires experts to program scripts. This process is slow and expensive and hence unlikely to scale to the majority of tasks that users perform. For instance, a recent automation project at Microsoft succeeded in scripting only about 150 of the hundreds of thousands of knowledge-base articles in a period of 6 months [10].

This paper introduces KarDo, a system that enables the mass of Internet users to automate computer tasks. KarDo aims to build a database of automated solutions for computer tasks. The key characteristic of KarDo is that a user contributes to this database simply by performing the task. For lay users this means interacting with the graphical user interface, which manifests itself as a stream of windowing events (i.e., keypresses and mouse clicks). KarDo records the windowing events as the user performs the task. It then merges multiple such traces to produce a canonical solution for the task which encodes the various steps necessary to perform the task on different configurations and for different users. A user who comes across a task he does not know how to perform searches the KarDo database for a matching solution. The user can either use the solution as a tutorial that walks him through how to perform the task step by step, or ask KarDo to automatically perform the task for him.

The key challenge in automating computer tasks based on windowing events is that events recorded on one machine may not work on another machine with a different configuration. To address this problem, a system needs to understand the dependencies between the system state and the windowing events. While the system could track these dependencies explicitly by modifying

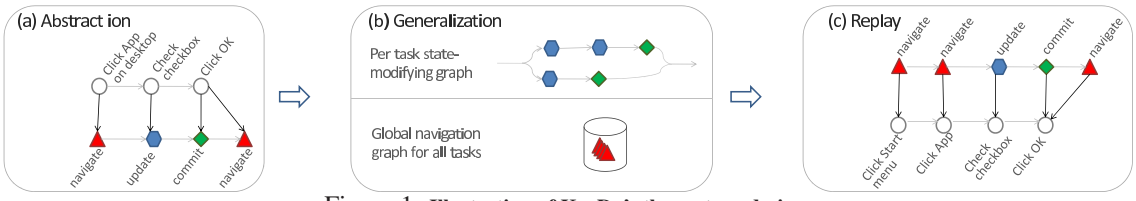


Figure 1: **Illustration of KarDo's three-stage design.**

the OS and potentially applications [18], such an approach presents a high deployment barrier and is hard to use for tasks that involve multiple machines (e.g., configuring a wireless router). KarDo therefore adopts an approach that implicitly infers system state dependencies, and does not require modifying the OS or applications. In particular, KarDo builds a model that maps windowing events to abstract actions that capture impact on system state: UPDATE and COMMIT actions, which actually modify system state, and NAVIGATE actions, which simply open or close windows but do not modify system state. KarDo performs this mapping automatically using machine learning. It then runs a set of static analysis algorithms on these sequences of abstract actions to produce a canonical solution which can perform the task on various different configurations. The system operates in 3 stages, described below and shown in Fig. 1.

(a) Abstraction. KarDo first captures the context around each windowing event (e.g, the associated application, window, widget *etc.*) using the accessibility interface, which was originally developed for visually impaired users and is supported by modern operating systems [8, 5]. KarDo then extracts from the context a per event feature vector, which it uses in a machine learning algorithm to map the event to the corresponding abstract action. Fig. 1(a) illustrates this operation.

(b) Generalization. KarDo then performs static analysis on the abstract actions in each recorded trace to eliminate irrelevant actions that do not affect the final system state. Once it has the relevant actions for each task, it proceeds to generalize them to deal with diverse configurations. Since navigation actions do not update state, KarDo can learn the many diverse ways to navigate the GUI from *totally unrelated tasks*, and therefore builds a global navigation graph across all tasks. In contrast, for state-modifying actions (i.e., UPDATES and COMMITS), KarDo uses differences across recordings of the *same task* to learn the different sequences of state-modifying actions that perform the task on various configurations, and represents this knowledge as a per task directed graph parameterized by configuration. Fig. 1(b) illustrates the generalization stage.

(c) Replay. In order to perform the task in a specific environment, KarDo walks down the graph of state-modifying actions trying to find a branch where all the actions involve applications (i.e. Thunderbird, Firefox, etc.) that exist on the machine. Once it finds such a branch, it proceeds to execute the actions along it. It moves from one state modifying action to the next by leveraging the global navigation graph to find a path from one of the active desktop widgets to the widget corresponding to the next state-modifying action. Fig. 1(c) illustrates the replay stage.

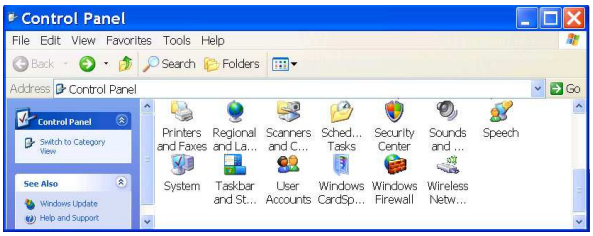
We built a prototype of KarDo as a thin client connected to a cloud-based service. We evaluate KarDo on 57 computer tasks drawn from the Microsoft Help website [9] and the eHow [4] websites which together include more than 1000 actions and include tasks like configuring a firewall, web proxy, and email. We generate a pool of 20 diversely configured virtual machines which we separate into 10 training VMs and 10 test VMs. For each task, two users performed the task on two randomly chosen VMs from the training set. We then attempt to perform the task on the 10 test VMs. Our results show that a baseline that tries both user traces on each test VM, and picks whichever works better, succeeds in only 18% of the cases. In contrast, KarDo succeeds on 84% of the 500+ VM-task pairs. Thus, KarDo can automate computer tasks across a wide variety of configurations without modifying the OS or applications.

We also performed a user study on 5 different computer tasks, to evaluate how well KarDo performs compared to humans for the same set of tasks. Even with detailed instructions from our lab website the students failed to correctly complete the task in 20% of the cases. In contrast, when given traces from all 12 users, KarDo produced a correct canonical solution which played back successfully on a variety of different machines.

2 Challenges

A system that aims to automate computer tasks based on user executions and without instrumenting the OS or applications, needs to attend to multiple subtleties.

(a) Generalizing Navigation. Consider the task of configuring a machine for access through remote desktop. On Microsoft Windows, the first step is to enable remote desktop on the local machine through the "System" dialog box which is accessed through the Control Panel. Automatically navigating to this dialog box can be difficult however because the Control Panel can be configured in three different ways. Novice users typically retain the de-



(a) Classic View

(b) Category View

Figure 2: **Diverse Configurations.** To enable remote desktop one must go to the "System" dialog box. Depending on the configuration of the Control Panel, one can either directly click the "System" icon (a) or must first navigate to "Performance and Maintenance" (b) then click the "System" icon.

fault view which uses a category based naming scheme, as in Fig. 2(a). Most advanced users however switch to the "Classic View" which always shows all available controls, as in Fig. 2(b). And, efficiency oriented users often go as far as configuring the control panel so it appears as an additional menu off of the start menu. All three paths however lead to the same "System" dialog box where one can turn on remote desktop. The challenge is to produce a canonical GUI solution that performs the task on machines with any of these configurations even when the recorded traces for this task show only one of the possible configurations.

(b) Filtering Mistakes and Irrelevant Actions. KarDo needs a mechanism to detect mistakes and eliminate irrelevant actions that are not necessary for the task. For example, while performing a task, the user may accidentally open some program that turns out to not be relevant for the task. If this mistake is included in the final solution, however, it will require the playback machine to have this irrelevant program installed in order for KarDo to automatically perform the task. It is important to remove mistakes like this to prevent the need for the user to rerecord a second "clean" trace, thus allowing users to generate usable recordings as part of their everyday work.

(c) Parameterizing Replay. After enabling remote desktop on his local machine, the user needs to configure the router to allow through the incoming remote desktop connections and direct them to the right machine. KarDo can easily automate a task like this, since it is done through a web-browser interface to the router, which provides the same accessibility information as all other GUI applications. The challenge arises, however, because one user may have a static IP address while another has a dynamic IP address, or worse, one user might have a DLink router, while another has a Netgear. Different steps are required to perform this task if the user has a static IP address vs. a dynamic IP address. Similarly, different routers present different web-based configuration interfaces, so users with different routers need to perform different GUI actions to perform this task. KarDo needs to retain each of these paths in the final canonical solution, and parametrize them such that the appropriate path can be chosen during playback. The challenge is to distinguish these configuration based differences from mistakes and irrelevant actions so that the former can be retained while the later are removed.

(d) User-Specific Entries. Some tasks require a user to enter his name, password, or other user-specific entries. KarDo can easily avoid recording passwords by recognizing that the GUI naturally obfuscates them, providing a simple heuristic to identify them. However, KarDo also needs to recognize all other entries that are user specific and distinguish them from entries that differ across traces because they are mistakes or configuration-based differences. It is critical to distinguish user specific entries from mistakes and configuration differences because KarDo should ask the user to input something like his username, while it should automatically discover which path to follow for different router manufacturers.

3 KarDo Overview

KarDo is a system that enables end users to automate computer tasks without programming, and does not require modifications to the OS or applications. It has two components, a client that runs on the user machine to do recording and playback, and a server that contains a database of solutions.

When a user performs a task that he thinks might be useful to others, he asks the KarDo client to record his windowing events while he performs the task. If the user cannot, or does not want to perform the task on his machine, he can perform the task remotely on a virtual machine running on the KarDo server, while KarDo records his windowing events. In either case, when the user is done, the client uploads the resulting windowing event trace to the KarDo server. The server asks the user for a task name and description. It uses this information to search its database for similar tasks and asks the user if his task matches any of those. This ensures that all traces for the same task are matched together.

When a user encounters a task he does not know how to perform, he searches the KarDo database for a solution. KarDo's search algorithm has access to not only the information that a normal text search would have, such as the task's name and description, the steps of the task,

and the text of each relevant widget, but also system level information like which programs are installed, and which GUI actions he has taken recently. As a result, we believe that task search with KarDo can be much more effective than standard text searching is today. However, effective search represents a research paper on its own, and so we leave the search algorithm details to future work.

The user can either use the solution as a tutorial that will walk him through how to perform the task step by step, or allow the solution to automatically perform the task for him. It is important to recognize however that KarDo's solutions are intended to be best-effort. Even a highly evolved system will not be able to automate correctly all of the time. Thus, KarDo takes a Microsoft Virtual Shadow Service snapshot before automatically performing any task, and immediately rolls back if the user does not confirm that the task was successfully performed (as discussed in §8, however, we leave the security aspects of this problem to future work).

The next three sections detail the three steps for transforming a set of traces recorded on one set of machines into a solution which allows automated replay on any other machine. §4 covers how to record the windowing events and map them to abstract actions that highlight how each action affects the system state. §5 then describes how to merge together multiple such sequences of abstract actions to create a generalized solution for any configuration. Finally, §6 discusses how replay utilizes the generalized solution and the state of the playback machine to determine the exact set of playback steps appropriate for that machine.

4 Windowing Events to Abstract Actions

The first phase of generating a canonical solution from a set of traces is to transform a windowing event trace into a sequence of abstract actions, since the generalization phase, discussed in §5 works over abstract actions. Performing this abstraction requires first converting the trace to a sequence of raw GUI actions by associating GUI context information with each windowing event, and then mapping raw GUI actions to abstract actions using a machine learning classifier.

4.1 Capturing GUI Context

A low-level windowing event contains only the specific key pressed, or the mouse button click along with the screen location. Effectively mapping these low-level events to abstract actions requires additional information about the GUI context in which that event took place such as which GUI widget is at the screen location where the mouse was clicked. KarDo gathers this information using the Microsoft Active Accessibility (MSAA) interface [8]. 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 [8]. Apple's OS X already provides a similar accessibility framework [7], and the Linux community is working to standardize a single accessibility interface as well [5]. The accessibility interface provides information about all of the visible GUI widgets, including their type (button, list box, etc.), their text name, and their current value, among other characteristics. It also provides a naming hierarchy of each widget which we use to uniquely name the widget. KarDo uses this context information to transform each windowing event to a raw GUI action performed on a particular widget. An example of such a raw GUI action is a left click on the OK button in the Advanced tab in the "Internet E-mail Setting" window.

4.2 Abstract Model

KarDo uses an abstract model for GUI actions. This model captures the impact that each action has on the underlying system state. We do not claim that our model captures all possible applications and tasks, however, it does capture common tasks (e.g., installation, configuration changes, network configurations, e-mail, web tasks) performed on typical Windows applications (e.g., MS Office, IE, Thunderbird, FireFox) as shown from the 57 evaluation tasks in Table 3. As discussed in §12, it also can be extended if important non-compliant tasks or applications arise.

In the abstract model all actions are performed on *widgets*. A widget could be a text box, a button, etc. There are three types of abstract actions in KarDo's model:

UPDATE **Actions:** These actions create a pending change to the system state. Examples of UPDATE actions include editing the state of an existing widget, such as typing into a text box or checking a check-box, and adding or removing entries in the system state, e.g., an operation which adds or removes an item from a list-box.

COMMIT/ABORT **Actions:** These actions cause pending changes made by UPDATE actions to be written back into the system state. An example of a COMMIT action is pressing the OK button, which commits all changes to all widgets in the corresponding window. An ABORT action is the opposite: it aborts any pending state changes in the corresponding window, e.g., pressing a Cancel button.

NAVIGATE **Actions:** These change the set of currently visible widgets. NAVIGATE actions include opening a dialog box, moving from one tab to another, or going to the next step of a wizard by pressing the Next button.

Note that a single raw GUI action may be converted into multiple abstract actions. For example, pressing the

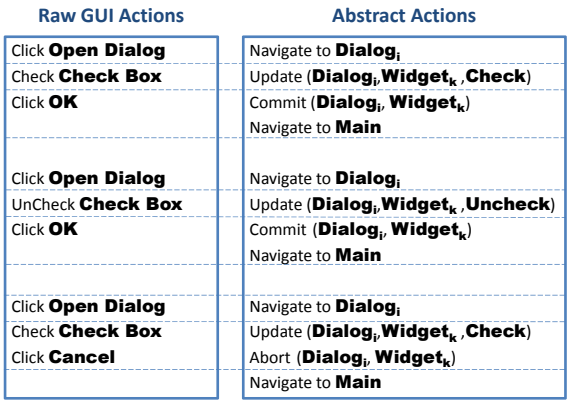


Figure 3: **A simplified illustration mapping raw GUI action to the corresponding abstract actions.**

OK button both commits the pending states in the corresponding window and navigates to a new view.

Fig. 3 illustrates a simple sequence of raw GUI actions and the corresponding abstract actions. Here a user clicks to open a dialog box, clicks to check a check box, and then clicks OK. He then realizes that he made a mistake and opens the dialog again to uncheck the check box. Finally, he opens the dialog one last time, rechecks the box, but reconsiders his change and hits the Cancel button. The corresponding sequence of abstract actions shows that the user navigated thrice to the dialog box, updated the check box, committed or aborted the UPDATE, and navigated again to the main window. However, the abstract model allows us to reason that the first UPDATE and the corresponding NAVIGATE and COMMIT actions are overwritten by the later UPDATE and hence are redundant and can be eliminated. Similarly, since the last UPDATE and associated ABORT do not update the state, they too can be eliminated. In §5.1, we describe KarDo's static analysis algorithm for filtering out such mistakes.

4.3 Mapping to Abstract Actions

KarDo has to label the raw GUI actions returned by the accessibility interface as UPDATE, COMMIT, and/or NAVIGATE. It does not attempt to explicitly classify ABORT actions because KarDo's algorithms implicitly treat the lack of a COMMIT action as an ABORT action as explained in §5.1. Further, a given action can have multiple different abstract action labels, or not have any label at all. KarDo performs the labeling as follows.

To label an action as a NAVIGATE action, KarDo uses the simple metric of observing whether new widgets become available before the next raw action. Specifically, KarDo's recordings contain information about not only the current window, but all other windows on the screen. Thus, if an action either changes the available set of widgets in the current window, or opens another window, then KarDo labels that action as a NAVIGATE action.[1]

Widget/ Window Features	Widget name (typically the text on the widget) Widget role (i.e., button, edit box, etc.) Does the widget contain a password? Is the widget updatable (i.e., check box, etc.)? Is the widget in a menu with checked menu items? Does the window contain an updatable widget?
Response To Action Features	Did the action cause a window to close? Did the action cause a window to open? Did the action generate an HTTP POST? Did the action cause the view to change? Did the action cause the view state to change?
Action Features	Action type (right mouse click, keypress, etc.) Keys pressed (the resulting string) Does the keypress contain an "enter"? Does the keypress contain alpha numeric keys? Is this the last use of the widget?

Table 1: **SVM Classifier Features.** This table shows the list of features used by the SVM classifier to determine which actions are UPDATE and COMMIT actions. All features are encoded as binary features with multi-element features (such as widget name) encoded as a set of binary features with one feature for each possible value.

Labeling an action as a COMMIT or UPDATE action is not as straightforward. There are cases where this labeling is fairly simple; for example, typing in a text box or checking a check box is clearly an UPDATE action. But to handle the more complex cases, KarDo approaches this problem the same way a user would, by taking advantage of the visual features on the screen. For example, a typical feature of a COMMIT action, is that it is associated with a user clicking a button whose text comes from a small vocabulary of words like {OK, Finish, Yes}.

KarDo does this labeling using a machine learning (ML) classifier. Specifically, an ML classifier for a given class takes as input a set of data points, each of which is associated with a vector of features and produces as output a label for each data point indicating whether or not it belongs to that class. It does this labeling by learning a set of weights which indicate which features, and which combinations of features, are likely to produce a positive data point, and which are likely to produce a negative data point. KarDo uses a supervised classifier, which does this learning based on a small set of training data.

KarDo uses two separate classifiers, one for COMMITS and one for UPDATES. These classifiers take as input a data point for each user action (i.e., each mouse click or keypress), and label them as UPDATES and COMMITS respectively. [2] Table 1 shows the features used by KarDo's classifiers to determine the labels. Features such as widget name, and widget role cannot be used directly by the classifiers however, because classifiers only work with numerical features. Thus, KarDo handles features

[1] KarDo will also label a window close as a NAVIGATE action in cases like a modal dialog box, where the user cannot interact with the underlying window again until the dialog box is closed.

[2] Note that since a given action is fed to both classifiers it can be classified as both an UPDATE and a COMMIT to account for actions like clicking the "Clear Internet Cache" button which both update the state and immediately commit that update.

like these, which are character strings, using the same technique as the Natural Language Processing community. Specifically, it adds a new binary feature for each observed string, i.e., is the the widget name "OK", is the widget name "Close", etc. This creates a relatively large number of features for each action which can cause a problem called overfitting, where the classifier works well only on the training data set, and it does not generalize to new data. To handle this large number of features, KarDo uses a type of classifier called a Support Vector Machine (SVM) which is robust to large numbers of features because it uses a technique called margin maximization. KarDo trains the SVM classifier using a set of training data from one set of traces, while all testing is done using a distinct set of traces.

5 Generalization

Generalization starts with multiple abstract action traces which perform the same task on different configurations and transforms them into a single canonical solution that performs the task on all configurations. KarDo performs this step by separating how it handles NAVIGATE actions from how it handles state modifying actions, i.e. UPDATES and COMMITS. Specifically, it first prunes out all NAVIGATE actions from each trace (and all unlabeled actions), leaving only the state modifying actions. It then follows a three step process to generate a canonical solution: (1) it runs a static analysis algorithm on each pruned trace that removes all the mistakes and irrelevant UPDATES; (2) these simplified traces are merged together to create a single canonical trace which is parameterized by user-specific environment; and (3) the NAVIGATE actions from all traces for all tasks are utilized to create a global navigation graph which is used to do navigation during playback. The rest of this section describes these three steps in detail.

5.1 Filtering Mistakes

The first step of generalization is to filter out mistakes from each trace. To understand the goal of filtering out mistakes, consider the example in Fig. 3, where the user opens the dialog box multiple times, changing the value of a given widget each time. In this example, the first check box UPDATE is overwritten by the second, while the third is never committed. Thus both of these UPDATES are unnecessary, and they should be removed along with the opening and closing of the dialog box associated with them. Their removal is important for two reasons. First, if a user chooses to read the text version of a solution, or to have KarDo walk him through the task, then such mistakes will be confusing to the user. Second, if not removed, mistakes like this can be confused as

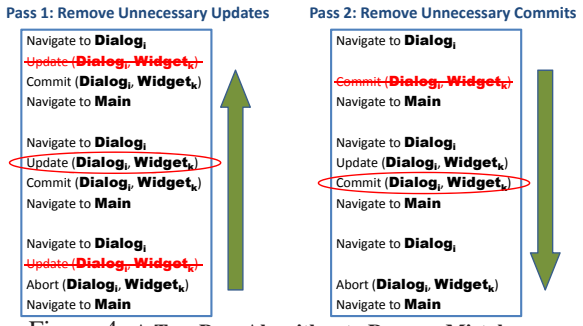


Figure 4: **A Two-Pass Algorithm to Remove Mistakes.**

user-specific or environment-specific actions and hence limit our ability to generalize.

The naive approach to identifying mistakes would compare multiple GUI traces from users who performed the same task, and consider differing actions as mistakes. Unfortunately, such an approach will also eliminate necessary actions which differ due to differences in users' personal information (e.g., printer name) or their working environment (e.g., different wireless routers).

In contrast, the key idea in KarDo is to recognize that the difference between unnecessary actions and environment specific actions is that unnecessary actions do not affect the final system state, and GUIs are merely a way of accessing this system state. So KarDo tracks the state represented by each widget and keeps only actions that affect the final state of the system. It does this using the following two-pass static analysis algorithm that resembles the algorithms used in various log recovery systems to determine the final set of committed UPDATES.

Pass 1 - Filtering Out Unnecessary UPDATES**:** The first pass removes all UPDATES on a particular widget except the last UPDATE which actually gets committed. Specifically, consider again our example from Fig. 3 where a user opens a given dialog box, and modifies a widget three times. We can see that KarDo needs to recognize that the second UPDATE overwrote the first UPDATE, rendering the first unnecessary. However, it cannot blindly take the last UPDATE, because the final UPDATE was aborted. Thus KarDo needs to keep the final *committed* UPDATE for each widget. It does this by walking backwards through the trace maintaining both a list of outstanding COMMITS, and a list of widgets for which it's already seen a committed UPDATE. As it walks backwards, it removes both UPDATES without outstanding COMMITS and UPDATES for which it's already seen a committed UPDATE on that same widget.

Pass 2: Filtering Out Unnecessary COMMITS**:** The second pass removes COMMITS with no associated UPDATES. It does this by walking *forwards* through the trace maintaining a set of pending UPDATES. When it reaches an UPDATE, it adds the affected widget to the

pending set. When it reaches a COMMIT, if there are any widget(s) associated with this COMMIT in the pending set, it removes them from the pending set, otherwise it removes the COMMIT from the trace.

One may fear that there are cases in which having the system go through an intermediate state is necessary even if that state is eventually overwritten. For example, if the task involves disabling a webserver, updating some configuration that can only be modified when the webserver is disabled and then re-enabling the webserver, it would be incorrect to remove the disabling and re-enabling of the webserver. While in theory such problems could arise, we find that in practice they do not arise. This is because actions like enabling and disabling a webserver typically look to KarDo like independent UPDATES which do not reverse each other, since one may require clicking the "disable" button while the other requires clicking the "enable" button. This causes the mistake removal algorithm to be somewhat conservative, which is the appropriate bias since it's worse to remove a required action than to leave a couple of unnecessary actions.

5.2 Parametrization

The second step of generalization is to parameterize the traces. Specifically, now that we have removed mistakes and navigation actions, the remaining differences between traces of the same task are either user specific actions (e.g. user name), or machine configuration differences (static IP vs. dynamic IP) which change the set of necessary UPDATE or COMMIT actions. To integrate these differences into a canonical trace that works on all configurations KarDo parametrizes the traces as follows:

(a) Parametrize UPDATES**.** The values associated with some UPDATE actions, such as usernames and passwords, are inherently user specific and cannot be automated. KarDo identifies these cases by recognizing when two different traces of the same task update the same widget with different values. To handle these kinds of UPDATES, KarDo parses all traces of a task to find all unique values that were given to each widget via UPDATE actions that were subsequently committed. Based on these values the associated UPDATE actions are marked as either *AutoEnter* if the associated widget is assigned the same value in all traces of that task, or *UserEnter* if the associated widget is assigned a different value in each trace. On play back, AutoEnter UPDATES are performed automatically, while KarDo will stop play back and ask the user for UserEnter actions. Note that if the widget is assigned to a few different values, many of which occur in multiple traces (e.g., a printer name), KarDo will assign it *PossibleAutoEnter*, and on play back let the user select among values previously entered by multiple different users or enter a new value.

(b) Parameterized Paths. All of the remaining differences between traces now stem from configuration differences in the underlying machine, which necessitate a different set of UPDATES or COMMITS in order to perform the same task. To handle this type of difference, KarDo recognizes that when a user's actions in two different traces differ because of the underlying machine configuration, the same action will generate two different resulting views. For example, consider the task of setting up remote desktop. Different traces may have used different routers, which require different sets of actions to configure the router. Since the routers are configured via a web browser, opening a web browser and navigating to the default IP address for router setup, http://192.168.1.1, will take the user to a different view depending on which router the user has. KarDo takes advantage of this to recognize that if the DLink screen appears, then it must follow the actions from the trace for the DLink router, and similarly for the other router brands.

Thus, KarDo builds a per-task state-modifying graph and automatically generates a separate execution branch with the branch point parameterized by how the GUI reacts, e.g., which router configuration screen appears. This ensures that even when differences in the underlying system create the need for different sets of UPDATES and COMMITS, KarDo can still automatically execute the solution without needing help from the user. If the traces actually perform different actions even though the underlying system reacts exactly the same way, then these are typically mistakes, which would be removed by our filtering algorithm above. If differences still exist after filtering, this typically represents two ways of performing the same step in the task, i.e. downloading a file using IE vs. Firefox. Thus KarDo retains both possible paths in the canonical solution and if both are available on a given playback machine, then KarDo will choose the path that is the most common among the different traces.

5.3 Building a Global Navigation Graph

Real world machines expose high configuration diversity. This diversity stems from basic system level configuration like which programs a user puts on their desktop and which they put in their Start Menu, to per application configuration like whether a user enables a particular tool bar in Microsoft Word, or whether they configure their default view in Outlook to be e-mail view or calendar view. All of these configuration differences affect how one can reach a particular widget to perform a necessary UPDATE or COMMIT. KarDo handles this diversity with only a few traces for each task by leveraging that multiple tasks may touch the same widget, and building a single general navigation graph using traces for *all tasks*.

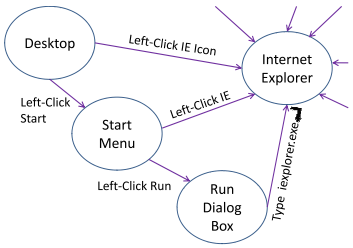


Figure 5: **Illustration of the Navigation Graph.** A simplified illustration showing a few ways to reach the IE window. The actual graph is per widget and includes many more edges.

Building such a general navigation graph is relatively straightforward. KarDo marks each NAVIGATE action as an enabler to all of the widgets that it makes available. KarDo then adds a link to the navigation graph from the widget this NAVIGATE action interacted with (e.g., the icon or button that is clicked), to the widgets it made available, and associates this NAVIGATE action with that edge. Fig. 5 presents a simplified illustration of a portion of the navigation graph. It shows that one can run IE from the desktop, the Start menu, or the Run dialog box.

6 Replay

The replay process takes a solution constructed using the process described in the preceding sections, and produces the low-level window events to perform a task on a particular machine. At each step, this process utilizes the full navigation graph, the per-task state-modifying dependency graph, and the current GUI context.

During replay, KarDo walks down the task's state-modifying dependency graph. As described in §5.2, this graph is parameterized by GUI context. Thus, KarDo utilizes the current GUI context and the installed applications to determine the path to follow at any branch point.

At each step, KarDo needs to ensure that the next state-modifying action is enabled. To enable a given UPDATE/COMMIT action, KarDo finds the shortest directed path in the navigation graph between the widget required for the UPDATE/COMMIT action, and any widget that is currently available on the screen. KarDo finds this path by working backwards in the navigation graph. Specifically, it first checks to see if the necessary widget is already available. If not, it looks in the navigation graph for all incoming edges to the necessary widget, and checks to see if any of the widgets associated with those edges are available. If not, it checks the incoming edges to those widgets, etc. It continues this process until either it finds a widget which is already available on the screen, or there are no more incoming edges to parse.

Once KarDo's navigation algorithm finds a relevant widget in the navigation graph which is currently available on the screen, it performs the associated action. If the expected next widget in the graph appears, KarDo follows the path through the navigation graph until the widget associated with the necessary UPDATE/COMMIT action becomes available. If at any point, the expected widget that the edge leads to does not appear, however, KarDo marks that navigation edge as unusable, and again performs the above search process. [3] It continues this process until either it succeeds in making the necessary UPDATE/COMMIT widget appear on the screen, or it has exhausted all possibilities and has no paths left in the navigation graph between widgets currently on the screen and the next necessary UPDATE/COMMIT widget.

Finally, each abstract action, whether NAVIGATE or state-modifying, is mapped to a low-level windowing event by utilizing the accessibility interface similar to the way it is used during recording.

7 Solution Validation

When a user uploads a solution for a task, KarDo allows the user to provide a solution-check. To do so, the user performs the steps necessary to confirm the task has been completed correctly and highlights the GUI widget that indicates success. For example, to check an IPv6 configuration, the user can go to ipv6.google.com and highlight the Google search button. As with standard tasks, KarDo will map the trace to abstract actions, clean it from irrelevant actions, etc. Such solution-checks allow KarDo to confirm that its canonical solution for a task works on all configurations by playing the solution followed by its solution-check on a set of VMs with diverse configurations, and checking that in each VM the highlighted GUI widget has the same state as in the solution-check.

8 Security

Ensuring that users cannot insert malicious actions into KarDo's solutions is an important topic that represents a research paper on its own. We do not attempt to tackle that problem in this paper. To handle non-malicious mistakes, however, KarDo takes a Microsoft Virtual Shadow Service snapshot before automatically performing a task and rolls back if the user is unhappy with the results.

9 Implementation

The KarDo implementation has three components: a client for doing the recording and the playback, a server to act as the solution repository, and a virtual machine infrastructure for remote recording and solution testing.

9.1 Client

Our current KarDo client is built on Microsoft Windows as a browser plugin. The user interface runs in the

[3] It caches the searched subgraphs to speed up any later searches.

browser and is built using standard HTML and Javascript which communicate with the plugin to provide all KarDo functionality. The plugin is written natively in C++. As discussed in §4.1, the plugin uses the OS Accessibility API to do the recording and playback.

The main implementation challenge in the client is to ensure that the GUI context of each mouse click and keypress can be recorded before the GUI changes as a result of the user action, i.e. before the window closes as a result of clicking the "OK" button. KarDo achieves the timely recording of the GUI context by utilizing the Windows Hooks API, which allows registration of a callback function to be called immediately before keypress and/or mouse click messages are passed to the application. The challenge is that such a callback function needs to be extremely fast, otherwise the UI feels sluggish to the user [17]. Calls to MSAA to get the GUI context are very slow, however, for two reasons: (1) they use the Microsoft Component Object Model (COM)[4] interface to marshal and unmarshal arguments for each function call, and (2) MSAA requires a separate call for each attribute of each widget on the screen (e.g., a widget name or role) often resulting in thousands of COM function calls per window.

We use two main techniques to maintain acceptable recording performance. First, we implement the callback function in a shared library so that it can run in-process with the application receiving the click/keypress. This significantly improves performance since it avoids the overhead of COM IPC for each function call. Second, instead of recording the GUI context of every window on the screen with every user input, we record only the full context of the window receiving the user input, and for all other windows we record only high level information such as the window handle, and window title. As we show in §10.4, this significantly improves performance when the user has many other windows open.

9.2 Solution Repository Server

The solution server provides a central location for upload, download and storage of all solutions. In our current implementation, all solution merging also happens on the server. We implement the solution server on Linux using a standard Apache/Tomcat server backed by a Postgres database. All solutions are stored on disk, with all meta-data stored in the database. When the client finishes recording a trace, KarDo immediately asks the user if he would like to upload the trace. Upon confirmation, the client uploads the trace to the server, and the server searches its existing database for solutions with similar sets of steps, and asks the user to confirm if his trace matches any of these. The server also provides a web interface listing all solutions. When a user finds a task they would like automatically performed, they click the *Play* button which calls into the client browser plugin to download that solution from the server and start playback.

9.3 Virtual Machine Infrastructure

The VM infrastructure is used for two purposes: 1) to enable users to record a solution for a task which they either cannot or do not want to perform on their own machine; and 2) to perform solution validation as discussed in §7.[5] KarDo's VM infrastructure is build on top of Kernel-based Virtual Machine (KVM)[6]. Its design is based on Golden Master (GM) VM images, which are generic machine images that have been configured to expose a certain dimension of configuration diversity, or make available a certain set of tasks. For example, some GMs are configured with static IP addresses, while others have dynamic IP addresses, and some have Outlook as the default mail client, while others have Thunderbird. The infrastructure can then quickly bring up a running snapshot of any GM by taking advantage of KVM's copy-on-write disks and its memory snapshotting support.

10 Evaluation

We evaluate KarDo on 57 computer tasks which together include more than 1000 actions and are drawn from the Microsoft Help website [9] and the eHow [4] website. We chose these tasks by randomly pulling articles from the websites and then eliminating those which did not describe an actual task (i.e. "What does Microsoft Exchange do?"), those which described hardware changes (i.e. "How to add more RAM"), and those which required software to which we did not already have a license. We focused on common programs, e.g., Outlook, IE, and diversified the tasks to address Web, Email, Networking, etc. The full list of tasks is shown in Table 3 and includes tasks like configuring IPv6, defragmenting a hard drive, and setting up remote desktop.

10.1 Handling Configuration Diversity

Our goal with KarDo is to handle the wide diversity of ways in which users configure their machines. Measuring KarDo's performance on a small number of actual user machines is not representative of the wide diversity of configurations, however, since many users leave the default option for most configuration settings. To capture this wide diversity, we generate a pool of 20 virtual machines whose configurations differ along the following axes: differences of installed applications (e.g., Firefox

[4] a binary interface used for inter-process communication

[5] We also used it to produce the evaluation results in §10.1.

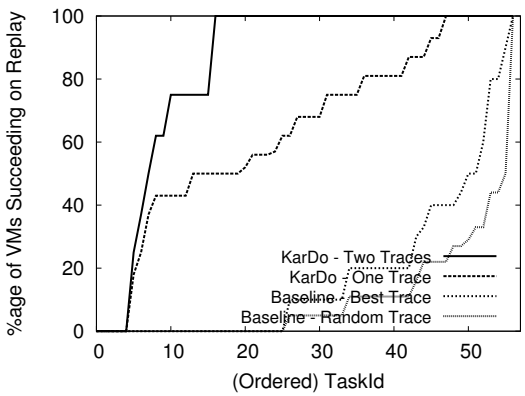


Figure 6: **Success Rate on Diverse Configurations:** For each task on the x-axis the figure plots on the y-axis the percentage of test VMs that succeeded in performing the task using a specific automation scheme. For each scheme, the area under the curve refers to the success rate taken over all task-VM pairs. KarDo-Two-Traces has a success rate of 84%, whereas KarDo-One-Trace has a success rate of 64%. In contrast, Best-Trace, which tries both of the two traces and picks whichever works better, has a success rate of only 18%, and Random-Trace, which randomly chooses between the two traces, has a success rate of only 11%.

vs. IE, Thunderbird vs. Outlook), differences of per-application configuration (e.g., different enabled tool and menu bars), user-specific OS configuration (e.g., different views of the control panel, different icons on the desktop), and different desktop states (e.g., different windows or applications already opened). We apply each configuration option to a random subset of the VMs. This results in a set of machines with more configuration diversity than normal, but which represent the kind of diversity of configurations we would like to handle.

We separate this pool of VMs into 10 training and 10 test. We recruited a set of 6 different users to help us record traces, including 2 non-expert users and 4 computer science experts. For each of the 57 evaluation tasks, two of the six users perform the task on two randomly chosen VMs from the training set. We then try to replay each task on the 10 test VMs. We compare four schemes:

- **KarDo - Two Traces:** We generate a canonical solution by merging together the two traces for each task, and we generate a navigation graph using all of the traces from all tasks. We then use the KarDo replay algorithm to playback the resulting solutions on the test VMs.
- **KarDo - One Trace:** We randomly pick one of the two traces and use it to generate a canonical solution for that task. The navigation graph is generated from that trace plus all traces for all other tasks (but not the other trace for that same task).
- **BaseLine - Best Trace:** For each VM, we try directly playing both of the two recorded traces for each task. If either trace succeeds then we report success for that VM-task combination. This shows how well a baseline system would perform with two traces per task.
- **BaseLine - Random Trace:** We randomly pick one of the two traces and directly playback all of the GUI actions in the original trace on the test VMs. This represents how well a baseline system would perform with only one trace per task.

Fig. 6 plots the success rate of these four schemes. It shows that the Best-Trace approach succeeds on average on only 18% of the VMs while the Random-Trace succeeds on just 11% of the test VMs. In contrast, KarDo succeeds on 84% of the 500+ VM-task pairs when given two traces, and on 64% when given only one trace. Thus, KarDo enables non-programmers to automate computer tasks across diverse configurations.

10.2 Understanding Baseline Errors

The Best-Trace and Random-Trace schemes are very susceptible to configuration differences. Even a single configuration difference can cause the Random-Trace scheme to fail. The two traces considered by the Best-Trace approach make it more robust to configuration differences, but it still only works if the test VM looks very similar to one of the VMs on which the recordings were performed. Consider a case where one recording opened Outlook from the desktop, and then accessed a menu item to change some configuration, and the other recording opened it from the Start Menu, and then used the tool bar to change that configuration. Even in this simple case where the two recordings see a large amount of diversity between them, the Best-Trace algorithm cannot handle a case where the tool bars are turned off, but Outlook is not on the desktop, or a case where menus are turned off, but Outlook is not in the Start Menu. More generally, even if the test VM is a hybrid of the two VMs on which the traces were recorded, the Best-Trace approach will fail. This is because a hybrid configuration requires pulling different parts from each of the traces which cannot be done without KarDo's technique of merging the traces together. Thus, the Best-Trace approach requires an excessive number of examples to successfully play back on diverse machines. Finally, we note that there are a number of tasks where the Best-Trace fails on all VMs. This occurs when all test VMs are widely different from the two VMs where the recordings were performed.

10.3 Understanding KarDo Errors

While KarDo successfully plays back in the vast majority of the cases, it still fails to playback successfully on 16% of the VM-task pairs. There are three main causes of these errors: classifier mistakes, incorrect navigation steps, and missing navigation steps. Fig. 7 shows the breakdown of these errors. Specifically it shows that eliminating classification errors results in a 91% success

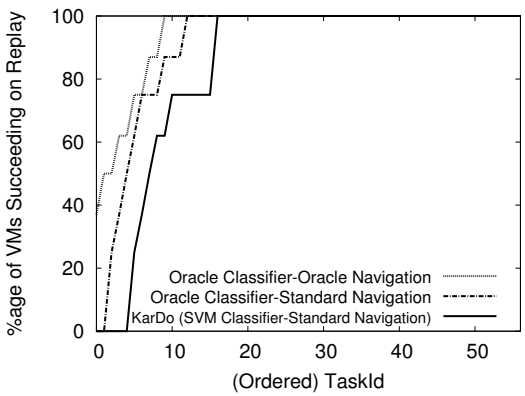


Figure 7: **Cause of KarDo Errors:** This figure shows the breakdown of the KarDo playback errors by showing the playback success when various parts of the KarDo algorithms are replaced by oracle versions. Recall that the success rate is the area under the curve. Based on the figure, replacing KarDo's classifiers with oracle classifiers increases the playback success rate from 84% to 91%. Additionally, eliminating all mistakes in the navigation database by using an oracle for navigation increases the playback success rate from 91% to 95%. The remaining 5% failure cases result from missing navigation steps that did not appear in any of the input traces.

rate while eliminating incorrect navigation steps results in a 95% success rate. We observe that the remaining 5% of the errors result mostly from missing navigation steps. The following discusses each of these in detail.

(a) ML Classification Errors: To evaluate our ML classifier, we manually labeled each of the actions performed by the users for the 57 tasks as a COMMIT action, an UPDATE action, both or neither. We then split this labeled data into half training and half test data. As described in §4.3 we run two separate classifiers on the data, one for UPDATE actions, and one for COMMIT actions. Since KarDo's generalization algorithm (from §5) retains only COMMITS and UPDATES as necessary actions, false negative misclassifications will cause KarDo to skip one of these necessary UPDATES or COMMITS during playback. False positives on the other hand will cause unnecessary actions to be retained, requiring KarDo to attempt to playback irrelevant actions which may be unavailable on a test VM. We calculate the false positive rate for each of the two classifiers as the percentage of actions in the COMMIT/UPDATE class that should not be in it, and the false negative rate as the percentage of actions not in the COMMIT/UPDATE class but should be in it.

The resulting performance of the KarDo classifiers is shown in Table 2. As we can see, the ML classifiers perform quite well even though classification mistakes account for almost half of the playback failures. Specifically, the COMMIT classifier has a false positive rate of only 2% and a false negative rate of only 3%. The COMMIT classifier performs so well because COMMITS follow very predictable patterns, i.e., they almost always occur when a button is pressed, and very frequently cause the associated window to close. The UPDATE classifier performs slightly worse with a 6% false positive rate and a 5% false negative rate. The higher false positive rate for UPDATES is caused by actions using widgets like combo boxes and edit boxes which are typically used for UPDATES, but are sometimes used just for navigational purposes. Occasionally when an action uses one of these widgets only for navigation (i.e., it's not an UPDATE), KarDo will misclassify the action as an UPDATE action. The higher false negative rate stems from actions which are both UPDATES and COMMITS. These actions tend to look much more like COMMITS than UPDATES and as a result the COMMIT classifier typically correctly classifies them, but the UPDATE classifier occasionally misclassifies them, not realizing they are also UPDATES. One such example is clicking the button to defragment your hard drive, which looks very much like a COMMIT action as it is a button click, and closes the associated window, but does not look very much like a typical UPDATE action since button clicks usually do not update any system state. In fact, if we test the UPDATE classifier after removing actions that are both COMMITS and UPDATES from the training and test sets the false negative rate drops to 2% without increasing the false positive rate at all.

	False Positive Rate	False Negative Rate
COMMITS	2%	3%
UPDATES	6%	5%

Table 2: **Performance of the COMMIT and UPDATE Classifiers.**

Note that a misclassification does not necessarily cause an error in the resulting canonical trace. In particular, only misclassifications that result in the eventual discard of a necessary action produce erroneous task solutions. For example, one may misclassify an action that is both COMMIT and UPDATE as only COMMIT. Still, as long as the mistake removal algorithm keeps this action as necessary, the resulting solution will still perform the UPDATE.

To evaluate the effect of classification mistakes on the final playback performance, we ran an "Oracle Classifier" version of KarDo where instead of using the output from the ML classifier to determine whether an action is an UPDATE or a COMMIT, we directly use the hand generated labels so that all classifications are correct. As shown in Fig. 7 this increases the playback success rate by an additional 7%. More training data would help eliminate these mistakes.

(b) Incorrect Navigation Steps: The next cause of playback problems comes from limitations in the way we currently generate the navigation graph. As discussed in §4.3, KarDo assumes that navigation depends only on the final action that made a widget visible. In a few cases, however, navigation depends on other earlier actions in the trace. A simple example of this is the "Run" dialog box which allows a user to type in the name of a program and then click "OK" to run it. In this case, the naviga-

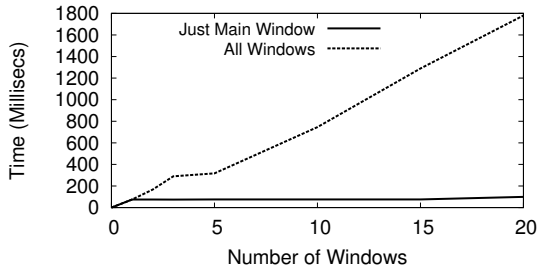


Figure 8: **Real Time Window System Context Recording:** The figure shows that KarDo's optimized recording, which limits recording full context information to only the main window, has a response time less than 100ms regardless of the number of windows. This is significantly below the 200ms threshold at which users perceive the UI to be sluggish. In contrast, recording the full context of all windows has a response time that scales with the number of windows, eventually becoming very slow.

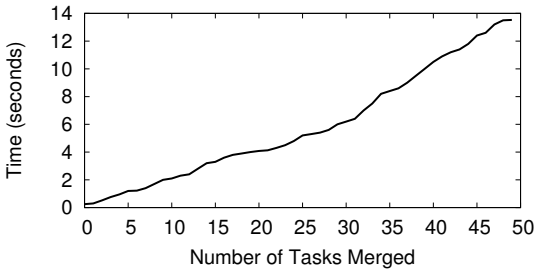


Figure 9: **Performance of Solution Merging:** The figure graphs the time that KarDo takes to merge a given number of traces, showing that KarDo can scale to quickly merge a large number of traces for a given task.

tion depends not only on clicking "OK", but also on the program name filled into the edit box.

To test the effect of incorrect navigation steps on the final playback success, we hand labeled all such dependent navigation actions. We then ran a "Oracle Navigation" version of KarDo where each navigation step had the full set of required actions associated with it. As shown in Fig. 7 this increases the playback success by an additional 4%. These mistakes can be eliminated by the additional classifier discussed in §12.

(c) Missing Navigation Steps: The final cause of playback problems stems from KarDo's fairly limited view of the GUI navigation landscape, due to the relatively small number of input traces in our experiments. Specifically, since many of the traces KarDo uses to generate its solutions are performed by users that already know how to perform a task, these traces rarely include navigation information related to incorrect navigations. This can cause playback to fail in the small fraction of cases where KarDo navigates in a way that is not appropriate for a given configuration and thus results in an error dialog box or some other GUI widget/window which was not seen in any trace. In this case, to ensure that it does not cause any problems, KarDo will immediately abort playback and roll back the user's machine to its original state. These type of errors account for most of the remaining 5% of playback errors shown in Fig. 7, and can be solved by more traces.

10.4 Feasibility Micro-Benchmarks

We want to ensure that KarDo's design performs well enough to be feasible in practice. To test this, we ran three performance tests on a standard 2.4 GHz Intel Core2 Duo desktop machine.

First, as discussed §9.1, context recording has to be fast so that it does not cause the user to perceive the UI as sluggish. Fig. 8 shows that even with many windows on the screen, KarDo can grab the relevant windowing system context in well less than 100ms, and the overhead is relatively constant regardless of the number of windows. Since users only start to notice delay when it is greater than 200ms [17], this additional delay should be acceptable to users. In contrast a scheme which records the context of all windows reaches an unacceptable delay of more than 1 second with even just 15 windows open.

Next, we check the performance of solution merging. Fig. 9 shows that merging up to 50 traces takes only 15 seconds, and it takes less than a second to merge 5 traces. This result shows that KarDo can easily scale to merging a large number of traces for each task.

Finally, KarDo's playback is relatively fast. For the 57 tasks in Table 3, playing a KarDo solution takes on average 52 seconds with a standard deviation of 9 seconds. The maximum replay time was 125 seconds, which was mostly spent waiting for the virus scanner to finish.

10.5 Working with Users

We evaluate KarDo's ability to improve on the status quo of using text instructions to perform computer tasks. We asked 12 CS students to perform 5 computer tasks within 1 hour, based on instructions from our lab website. We also used KarDo to automate each task by merging the students' traces into a single canonical solution.

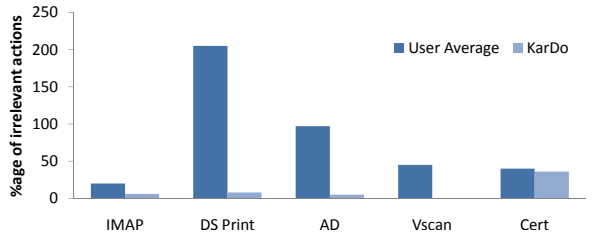
We find three important results. First, as shown in Fig. 10(a), even with detailed instructions, the students fail to correctly complete the tasks in 20% of the cases. In contrast, KarDo always succeeded in generating a solution that automated the task on all 12 user machines.

Second, as shown in Fig. 10(b), even when the students did complete the tasks they performed on average 84% more GUI actions than necessary, and sometimes more than three times the necessary number of actions. KarDo's automation removes most of these irrelevant actions, performing only 11% more actions than necessary.

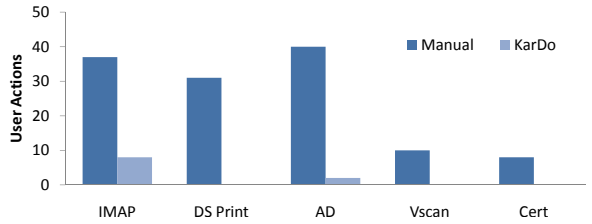
Third, as shown in Fig. 10(c), KarDo reduced the per-task required number of times the user had to interact with the machine from 25 to 2 times, on average. This reduction is because KarDo requires manual entry only for user-specific inputs, and automates everything else.

	IMAP Configuration	Double-Sided Printing	Active Directory	Virus Scan	Fix FireFox Certificate
KarDo	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
User 1	Yes	Yes	Yes	Yes	Yes
User 2	Yes	No	Yes	Yes	Yes
User 3	Yes	Yes	Yes	Yes	Yes
User 4	Yes	No	No	Yes	Yes
User 5	No	No	Yes	Yes	Yes
User 6	No	Yes	No	Yes	Yes
User 7	Yes	Yes	Yes	Yes	Yes
User 8	Yes	No	Yes	Yes	Yes
User 9	Yes	No	Yes	Yes	Yes
User 10	No	No	Yes	Yes	Yes
User 11	Yes	Yes	Yes	Yes	Yes
User 12	Yes	Yes	Yes	Yes	Yes

(a) Task successes and failures.



(b) Percentage irrelevant actions performed by users and KarDo



(c) User manual inputs with and without KarDo.

Figure 10: **Working with Users:** The figures shows that (a) KarDo performs the task correctly, even when many users fail, (b) KarDo filters most irrelevant actions, and (c) with KarDo users need to manually perform very few steps, typically only those which require user-specific information.

These results show that KarDo can help users reduce the time and effort spent on IT tasks.

11 Related Work

While there are many tools to help automate computer tasks, most either do not support recording and must be scripted by programmers (e.g., AutoIt [2] and AutoHotKey [1]), or allow recording only by relying on application specific APIs and thus cannot be used to automate generic computer tasks (e.g., macros, DocWizards [14]). Apple's Automator [3], Sikuli [13] and AutoBash [18] are the only exceptions as far as we know. However, neither Automator nor Sikuli can automatically produce a canonical GUI solution that works on different machine configurations. AutoBash covers only tasks which are entirely contained on the local machine, which is increasingly infrequent with today's networked computer systems. Additionally, it requires modifying the kernel to track dependencies across applications and then taking diffs of the affected files. Such kernel modifications are a deployment barrier, and file diffs are ineffective on binary file formats.

Some tools support recording and check pointing, such as DejaView [16], but they do not actually playback a task, instead only returning to a checkpointed state.

Lastly, there are tools that leverage shared information across a large user population [21, 20, 15, 19, 12, 11]. Strider [21] and PeerPressure [20] diagnose configuration problems by comparing entries in Windows registry on the affected machine against their values on a healthy machine or their default values in the population. FTN addresses the privacy problem in sharing configuration state by resorting to social networks [15]. [19] and [12] track kernel calls similar to AutoBash to determine problem signatures and their solutions. NetPrints [11] collects examples of good and bad network configurations, builds a decision tree, and determines the set of configuration changes needed to change a configuration from bad to good. All of these tools compare potentially problematic state information against a healthy state to address computer problems and failures. KarDo focuses on a complementary issue where the existing machine state maybe perfectly functional but the user wants to perform a new task. KarDo addresses such how-to tasks by working at the GUI level, which allows it to handle any general task the user can perform.

12 Addressing KarDo's Limitations

While our system represents a first step towards providing a system for automating a task by doing it, our current implementation has multiple limitations we expect to explore in future work. First, our model of labeling all actions as COMMITS, UPDATES and NAVIGATE actions is not exhaustive. Specifically, it does not cover tasks which simply show something on the screen. For example, a task like "Find my IP Address" will look to KarDo like it does nothing, and so all actions will be removed. This can be addressed by extending the model. Second, as discussed in §10.1, it does not handle tasks containing complex navigation actions. For example if navigation requires typing the name of a program in an edit box and then clicking "Run" then KarDo will only click the "Run" button. This can be solved using an additional classifier to detect these dependent navigation actions. Finally, KarDo requires unnecessary manual steps when entering the same user specific information across many tasks. For example, a user will have to manually enter his Google username every time he wants to run any task that accesses Google services.To handle this, we'd

like to build a profile for each user which will remember previous inputs by a user and reuse them across tasks.

13 Concluding Remarks

This paper presents a system for enabling automation of computer tasks, by recording traces of low-level user actions, and then generalizing these traces for playback on other machine configurations through the use of machine learning and static analysis. We show that automated tasks produced by our system work on 84% of configurations, while baseline automation techniques work on only 18% of configurations.

This paper has focused on use of our system for building an on-line repository of automated IT tasks which would include both local configuration and setup as well as remote tasks such as configuring a wireless router. We note, however, that our system is useful for many other applications as well, including replacing IT knowledge-bases, automated software testing, and even use by expert users as an easy way to automate repetitive tasks.

Acknowledgments

We'd like to thank Steve Bauer and Neil Van Dyke for their help implementing an early version of the system, and Micah Brodsky and Martin Rinard for help with the mistake removal algorithm. Also, we greatly appreciate Hariharan Rahul's help editing an early draft of this paper, and Sam Perli and Nabeel Ahmed's help generating early results. Lastly we'd like to thank Regina Barzilay, S.R.K. Branavan, James Cowling, Evan Jones, Ramesh Chandra, Jue Wang, Carlo Curino, Lewis Girod and our shepherd Michael Isard for their feedback on the paper. This work was supported by NSF grant IIS-0835652.

References

[1] AutoHotkey. http://www.autohotkey.com/.
[2] AutoIt, a freeware Windows automation language. http://www.autoitscript.com/.
[3] Automator. http://developer.apple.com/macosx/automator.-html.
[4] eHow. http://www.ehow.com.
[5] IAcessibility2. http://www.linuxfoundation.org/en/Accessibility/-IAccessible2.
[6] KVM. http://www.linux-kvm.org.
[7] Mac OS X Accessibility Framework . http://developer.apple-.com/documentation/Accessibility/Conceptual/AccessibilityMac-OSX/AccessibilityMacOSX.pdf.
[8] Microsoft Active Accessibility. http://en.wikipedia.org/wiki/-Microsoft_Active_Accessibility.
[9] Microsoft Help. http://windows.microsoft.com/en-us/windows/help.
[10] Security Garden Blog. http://securitygarden.blogspot.com/2009/-04/microsoft-fix-it-gadget.html.
[11] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, 2009.
[12] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. *USENIX*, 2008.
[13] T.-H. Chang, T. Yeh, and R. C. Miller. Gui testing using computer vision. In *CHI*, 2010.
[14] L. D. B. et. al. DocWizards: A System For Authoring Follow-me Documentation Wizards. In *UIST*, 2005.

E-mail	
Sending/ Receiving	Turn off E-mail Read Receipts (54, 27) Automatically forward e-mail to another address (35, 30)
Viewing	Restore the unread mail folder (16, 8) Highlight all messages sent only to me (31, 24) Change an e-mail filtering rule (18, 19) Add an e-mail filter rule (46, 26) Make the recipient column visible in the Inbox (27, 11) Order e-mail message by sender (19, 73) Create an Outlook Search Folder (12, 12) Turn on threaded message viewing in Outlook (16, 9) Mark all messages as read (44, 48) Automatically empty deleted items folder (22, 24)
Junkmail	Empty junk e-mail folder (9, 9) Turn off Junk e-mail filtering (22, 14)
Security	Consider people e-mailed to be safe senders (19, 25) Send an e-mail with a receipt request (20, 12)
Contacts/ Calendar	File Outlook contacts by last name (25, 13) Set Outlook to start in Calendar mode (15, 23)
RSS Feeds	Add a new RSS feed (14, 15) Change the Name of an RSS feed (12, 23)
Other	Turn off Outlook Desktop Alerts (24, 35) Reduce the size of a .pst file (26, 39) Turn off notification sound (22, 66) Switch calendar view to 24-hour clock (20, 14)
Office Applications	
Excel	Delete a worksheet in Excel (8, 8) Turn on AutoSave in Excel (33, 111)
Word	Disable add-ins in Word (25, 23)
Web	
Browser	Install Firefox (23, 21)
Proxy	Manually Configure IE SSL Proxy (61, 83) Set Default Http Proxy (7, 7)
Networking	
Security and Privacy	Enable firewall exceptions (9, 9) Enable Windows firewall (6, 6) Disable Windows firewall notifications (8, 9) Disable Windows firewall (6, 9)
IPv6	Disable IPv6 to IPv4 tunnel (8, 7) Show the current IPv4 routing table (10, 17) Show the current IPv6 routing table (13, 10)
DNS	Use OpenDNS (44, 38) Stop caching DNS replies (6, 9) Use Google's Public DNS servers (32, 32) Use DNS server from DHCP (22, 22)
Routing	Configure system to pick routes based on link speed (22, 17) Set routing interface metric (18, 19)
System	
Utilities	Analyze hard drive for errors (7, 13) Defragment hard drive (10, 13) Enable Automatic Updates (7, 6) Set Up Remote Desktop (12, 10)
User Interface Settings	Hide the Outlook icon in the System tray (21, 18) Change to Classic UI (15, 13) Delete an Item from the Task Bar (13, 9) Change desktop background color (35, 26) Enable Accessibility Options (20, 20) Auto-Hide the Taskbar (52, 41) Change date to Long Format (33, 19) Set Visual Effects for Performance (13, 13)
Other	Set Outlook as default E-mail program (26, 15) Enable Password on Screen Saver and Resume (22, 29)

Table 3: 57 tasks used to evaluate KarDo. Each task is listed with the number of actions performed in each of the two traces.

[15] Q. Huang, H. Wang, and N. Borisov. Privacy-Preserving Friends TroubleShooting Network. In *NDSS*, 2005.
[16] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. Dejaview: A personal virtual computer recorder. In *SOSP*, 2007.
[17] Olsen. *Developing User Interfaces*. Morgan Kaufmann, 1998.
[18] Y. Su, M.A., and J. F. Autobash: improving configuration management with operating system causality analysis. *SOSP*, 2007.
[19] Y.-Y. Su and J. Flinn. Automatically generating predicates and solutions for configuration troubleshooting. *USENIX*, 2009.
[20] H. J. Wang, J.P., Y.C., R.Z., and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.
[21] Y.-M. Wang and et. al. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.