Blaze – Automating User Interaction in Graphical User Interfaces

Gabriel Barata
Tiago Guerreiro
Daniel Gonçalves

gabriel.barata@ist.utl.pt
tjvg@vimmi.inesc-id.pt
daniel.goncalves@inesc-id.pt

ABSTRACT
Recurrence, computer users face repetitive tasks that could be automated. Although several applications can automate well-known repetitive tasks, most of those tasks are unique and arise from everyday usage, making them unpredictable. In a situation where users face a repetitive task, they have few choices: if they are experienced users, they’ll probably be able to build a script to automate the task; if not, they will need to complete it themselves or try to find a specific application to do it, spending as much time learning how to use it as they would spend finishing it by hand.

We present Blaze, a system capable of automating user’s repetitive tasks in graphical user interfaces, operating system-wide. Blaze is an application launcher that continuously monitors and analyzes the users’ activities, without disturbing them. Blaze then identifies relationships and repetitions in user actions, infers patterns in those repetitions, and offers to accomplish the remaining of the task by the user. To detect repetitions, Blaze resorts to a fast string search algorithm, based on common prefix search in a suffix-tree. It suggests a set of actions to complete the recurrent tasks, in a non-intrusive way. These actions can be generalized and stored in scripts, for future use in new situations, different from the ones in which they were initially detected. User studies show that, even for relatively simple tasks, Blaze can help users to perform 3 times faster.

Author Keywords
Implicit programming by example, End-user programming, User context awareness, Task automation.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces.

INTRODUCTION
Nowadays operating system interfaces are full-blown and very appealing, providing a lot of interactive aids for most common user problems. However, progress has begotten interfaces that have lost the expressive power once granted by command line interfaces. While requiring users to memorize sets of valid commands, those interfaces gave them a more efficient control over their files, by using mechanisms such as wildcards and through the ability of a command to affect several files at once. Also, by providing scripting capabilities, they more easily supported repetitive user task automation.

Imagine that you went on vacations, and you took over one hundred photos with your digital camera. After you download them to your computer, you notice that all photos have names, given by the digital camera, such as “SDC10050.JPG”, “SDC10051.JPG”, etc. Wanting to have more meaningful names, such as “Summer Vacation 01.jpg”, “Summer Vacation 02.jpg”, and so on, it is clear you must rename them. Depending on your skill level, regarding the operating system you use, you may have different options to face the problem. If you are inexperienced, your only options are to rename the photos one by one, or trying to download a renaming tool from the Internet, and spend as much time learning how to work with it as you would spend renaming the photos yourself. However, if you are proficient with your operating system and familiar with programming languages, you could just do a script yourself (such as a batch file on Microsoft Windows) to accomplish the task, by iterating through all JPG files in that folder and renaming them. Therefore, you need some programming skills and knowledge to perform this task on an acceptable amount of time.

Leaving file system operations behind, there are several other interesting situations where the user repeats the same actions, over and over again. For instance, when an user inserts a table on a text processor, always repeating the same button pressing sequence, or even when he inserts a few sequential text tokens in separated lines, like “Note #1:”, “Note #2:”, “Note #3:”, and so on, where, clearly, there is a numerical pattern that could easily be automated. To solve the former case, one can use a macro recorder, which does not require much knowledge about the operating system itself, but still may have a steep learning curve. To solve the latter case, an advanced scripting language could be used, such as AutoIt v31. Nonetheless, this solution requires a lot of knowledge on Basic like programming languages and about how the operating system communicates with processes. This is, thus, beyond the reach of most common users. Most common users do not have any advanced knowledge about programming.

1 http://www.autoitscript.com/autoit3/ (Last visited in: June 1st, 2009)
languages and do not understand how the operative system really works. They only know how to use it.

In order to bridge these gaps, we developed an application that is able to automate repetitive tasks, on Microsoft Windows. This application, named Blaze⁵, takes the form of an application launcher, implementing and enhancing most of the features from its counterparts. The typical application launcher interface is well suited for the job because it provides an easy and fast way to launch a specific function, based on a portion of the function’s name. Furthermore, Blaze takes this to a whole new level by being able to tolerate typos, allowing the user to mistype commands. This way, not only the users are exempt from having to navigate through dozens of menu entries, but also, they only need to remember part of a function’s name.

Besides automating well known recurrent tasks, such as launching applications, performing calculations and web searches, common application launchers are useless when faced with the unique tasks that emerge from everyday usage. Blaze addresses this issue by being able to monitor user’s activity and infer repetitive tasks, without requiring user intervention. As user actions are detected, Blaze establishes relationships between them, describing a specific behavior, and detects repetitive patterns in these behaviors, in order to infer which recurrent tasks the user is performing. Whenever a new recurrent task is detected, Blaze non-intrusively notifies the user and exposes a suggestion which can be accepted or just ignored.

Blaze is always aware of the user context and uses that information to personalize most features. Examples of contextual information detected by Blaze might be the address of the webpage or folder currently being browsed by the user, folder items or text currently selected by the user and even what wireless networks is he/she connected to. Moreover, Blaze also features advanced scripting capabilities, which have two main functions: a) allow the user to save detected repetitions, which can later be reproduced; b) allow the user to easily extend Blaze’s functionalities, without having to compile any code, just by using a regular text editor.

In the following section we will describe other prototypical approaches, regarding repetitive user task automation, and analyze the most relevant application launchers currently available. By identifying their shortcomings, we are able to build solid foundations on which we rely to develop Blaze. We will briefly describe Blaze’s architecture and give a more detailed explanation of how it detects repetitive patterns and creates generalizations. This will lead us to a description of the user studies and their results, of a system performance evaluation, and a brief discussion of the major conclusions and possible future work.

RELATED WORK
Despite all the evolution that operative systems’ graphical user interfaces have gone through, it is clear that a lot of expressive power has been lost, formerly granted by command line interfaces. This loss created a gap on nowadays interfaces, which have been filled by application launchers. This lack of expressive power is so notable that recent commercial operative systems include their own basic application launching mechanisms, such as the Mac OS’s Spotlight and the Windows Vista start menu search.

Currently, there are several application launchers, available for almost all operating systems, and most of them share a set of common features. Such features include on-the-fly application launching, performing calculations, web searching, and file-system browsing, command and path auto-completion and browsing recently accessed files.

However, a few application launchers stand out from the crowd. The spotlight points to those that, somehow, offer advanced automation features. These are mainly attained through macro editing features, operating system integration and manipulation capabilities, and interactive command learning. For example, activAid⁴ and Keybreeze⁴ are noteworthy for offering the user means to automate basic repetitive tasks, such as performing cronojobs and recording macros. Although these features might very useful for experienced users, regular PC users will not be able to use them to their advantage. Furthermore, the mere act of recording a macro puts the user on an explicit programmer position.

There are, also, other remarkable application launchers, such as Dash Command⁴ and Enso⁵, due to their ability to integrate with other applications and with the operating system itself. These applications are also slightly context aware, being able to operate over currently selected text and folders. Operating over selected text is one of Enso’s most useful features; however, this relies on the operating system’s accessibility features, limiting its use by not providing an application-independent way to exchange information between applications [2]. Launchy⁷ stands out by being simple, fast, extensible, easy to use, and visually appealing. This makes it the application launcher of choice for thousands of users. It provides no automation features, however.

Although these application launchers may speed up user activity, they lack of significant automation tools to face the unique tasks that arise from everyday usage. There are, however, a few prototypical systems that address this issue.

---

⁵ Available online at http://blaze-wins.sourceforge.net/

³ http://www.heise.de/ct/activaid/ (Last visited in: June 1st, 2009)
⁴ http://www.keybreeze.com/ (Last visited in: June 1st, 2009)
⁵ http://www.trydash.com/ (Last visited in: June 1st, 2009)
⁶ http://www.humanized.com/ (Last visited in: June 1st, 2009)
⁷ http://www.launchy.net/ (Last visited in: June 1st, 2009)
SMARTedit [6,7] consists of a text editor, which can automate repetitive tasks, resorting to machine learning mechanisms based on version space search [10]. Lau introduces the Version Space Algebra [5], which allow the composition of a more complex version space by composing together simpler version spaces. SMARTedit starts by having a hierarchical version space, containing all possible programs that it is able to learn. As the users start recording their actions, some functions are thrown out of the version space, because they are not consistent with the observed actions. This way, as long as the users are recording their actions, the version space gets increasingly more specific, allowing SMARTedit to pick the best choice, once the recording is finished. The most likely action is chosen by capturing the current application state, treating it as an input and executing each function, in the version space, on this input, thus producing a set of output states. If the result is a single output state, then it is presented to the user with 100% probability of being what the user intended to do. If there is more than one result, the one with highest probability is chosen.

It is interesting to point out that, to build a hierarchical version space and to composite more complex version spaces from simpler ones, SMARTedit relies on domain knowledge, taking into account the importance of each domain concept. SMARTedit’s other major drawback is that it is based on the macro concept, which implies the user taking part in the programming procedure.

There are, however, a few systems that can solve this problem. Remarkable examples are Eager [1] and APE [13], [14]. Eager is an assistant for the HyperCard environment, which is able to identify the first iterations of a cycle, until a certain condition is met. On the other hand, APE consists of an assistant for a programming environment that is able to extend Eager’s capabilities by detecting repetitions even if these do not occur consecutively in the user actions’ history. Both these systems differ from SMARTedit mainly because they do not require the user to start any action recording. They constantly monitor the user, without interfering with his task. Both embrace the Implicit Programming by Example [15] concept, which consist in learning what to automate and when to make a suggestion, without being a nuisance to the user. Therefore, it relies on two major concepts: avoiding the macro metaphor, because the user does not need to be explicitly involved in the programming procedure, and embracing the Assistant metaphor, by having an agent that offers assistance whenever a repetition is detected.

Accordingly, APE is composed by three agents: the Observer, the Apprentice and the Assistant. The Observer monitors user’s activity and notify the other agents whenever a new action is detected. The Apprentice, in turn, detects repetitive tasks and recognizes the situations in which these tasks occurred. For detecting repetitive tasks, APE uses the KMR algorithm [4]. For detecting the situations in which the repetitions occur, two algorithms are used: a decision tree based algorithm, named C4.5 [12] and a Candidate-Elimination based one, named IDHYX [13]. Lastly, the Assistant is responsible for matching current user actions with the situational patterns composed by the Apprentice, and check if there is any associated repetition. If so, a suggestion is discreetly made to the user.

Also implementing the Macro metaphor, Creo [3] automates repetitive actions on Internet Explorer, resorting to a data detector, named Miro, which matches web pages’ content to high-level user goals. Therefore, Creo is able to record a sequence of operations in the user interface, which, thus, is generalized accordingly to particular kinds of data, detected by Miro. This sequence can later be replayed even if the new situation is not exactly alike the one in which the recording was performed.

Regarding repetitive user task automation, our goal is similar to APE’s: resort to Implicit Programming by Example to learn what tasks should be automated and when should a suggestion be made, without disturbing the user. However, unlike APE, we will extend this automation, not to a single application, but to a whole operating system. This leads us to a new obstacle: we cannot rely on domain knowledge, as there are too many different applications which can be used on an operating system. On the other hand, regarding application launcher features, Blaze implements most of its counterparts’, resorting to user context awareness to enhance them to a superior level.

A NEW SYSTEM IS BORN
To fill the gap left by the above mentioned shortcomings, we created a new system, named Blaze. Blaze’s development was based on 6 main requirements:

1. Expressive yet simple user interface. Therefore, we chose a typical application launcher’s interface, as it fulfils user’s needs with only a few keystrokes.
2. By picking an application launcher as the interface, we have to enhance its typical features to a new level.
3. The system must be easily extensible.
4. Blaze should detect repetitions in user’s activities and offer to auto-complete tasks in the user’s place.
5. Repetition pattern detection and suggestion must not distract the user.
6. Suggested automations should be able to adapt to new situations and be saved for later reproduction.

Refining the Raw Material
The typical application launcher interface was our choice as it offers the expressive power of the former command line interfaces, yet it is more visually appealing and integrates very well with the operating system and modern graphical user interfaces.

Blaze offers most of its counterparts’ features, such as on-the-fly application launching, web and file system browsing, command and path auto-completion, numerical
calculations, web search, command line integration, and so on. Additionally, it offers more advanced features, like quick text insertion in other applications, and is able to perform operations over user’s selected text. It is also capable of indexing the contents of files, such as text within text files and ID3 tags from music files. Major enhancements were made in three major areas: text prediction, user context awareness and extensibility.

Text prediction is the key to an application launcher’s success. It is responsible to match user’s typed text to commands supported by the system. Current application launchers have a major drawback: they do not allow users to make typos. If a user types “fierfox” instead of “firefox”, a regular application launcher will not be able to indentify that the user might be willing to launch Mozilla Firefox.

Blaze overcomes this issue by implementing a fast text prediction algorithm that relies on the Levenshtein distance metric [8] to tolerate user typos, as show in Figure 1.

![Figure 1. Blaze tolerating typos.](image)

Blaze offers better user context awareness by being able to retrieve valuable information about what the user is doing in a precise moment, such as the address of the webpage or folder currently being browsed by the user, currently selected files or text and even the SSID of the wireless network to which the user is connected to. For instance, the user may select the headline of an interesting article from a website, open Blaze and type:

```
email "john.doe@provider.com" "!this" "!url"
```

The default email client will opens, with a composed mail for “john.doe@provider.com”, with the article’s headline as the subject and the webpage’s address as the body.

The ![this](image) and ![url](image) context commands are examples of contextual commands, giving access to contextual information, which can be combined with other Blaze commands. The ![this](image) command represents the currently selected text while the ![url](image) one represents the address of the webpage currently being browsed. It is interesting to point out that the ![this](image) command uses the clipboard to capture data, which is application independent and, thus, assures compatibility with many applications. Several other contextual commands exist.

In terms of extensibility, Blaze provides two ways to extend its features: through dynamic-link libraries, written in C#, or through a script, written in IronPython. The former allows the user to add new sets of commands and new indexing rules. The latter adds one new command per script, which can make use of the powerful .NET framework without the user having to compile any code.

IronPython scripts can also be automatically generated by Blaze, which will perform some repetitive task it detected.

**Adding New Stuff**

The innovation behind Blaze resides in the possibility of monitoring all user actions, OS-wide. To attain this, we resort to several windows API and .NET functions, as well as mouse and keyboard system hooks.

![Figure 2. Blaze Assistant.](image)

Like APE, Blaze includes three agents: an Observer, an Apprentice and an Assistant. The Observer collects the user’s context information and converts it into domain objects, named User Actions. User actions may be combined among them to compose higher level actions. The Apprentice, in turn, analyzes the user actions’ history, captured by the Observer, and detects repetitive patterns. To achieve this, a string search algorithm is used. Whenever a repetition is detected, the Assistant generates sequences of actions, named automations, capable of completing the task for the user. These automations are suggested to the user, non-intrusively, by lighting up an icon in the system tray. The user can accept the suggestion by double tapping the CapsLock hotkey or just ignore it. By accepting it, a form shows up displaying one or several narratives, as shown in Figure 2, where the user can parameterize the execution and even modify the suggestion.

**UNDER THE HOOD**

Building a system capable of detecting repetitions and automating tasks, for the entire operating system, is quite different from doing so for a single application. While in the latter case we may have at our disposal domain information about every action the user can make, in the former case we not always know what the user might be doing. For example, Blaze might capture that the user is pressing the left mouse button on Notepad++, but it does not know if different tab or a piece of text are being selected, or some setting is being changed. There are too many different applications supported by current operating systems, which are coded by different people, using different standards, to keep track of every action the user can make on each one of them.
The Observer

The aforementioned issue raises the major problem the Observer has to face: to convert sparse data about user activities, from different sources, into reliable data that can, subsequently, be processed by the other agents.

The Observer monitors three sources: 1) mouse and keyboard system wide hooks, which raise events every time the user uses those peripherals; 2) the file system watcher, which raises events every time the user performs actions over specific folders or files; and 3) the context awareness module, that complements other sources’ information with additional data about the application in which the events occurred. However, these different sources provide data with different granularity. For instance, if the user types “hello”, five key press actions would be generated: press “h”, “e”, “l”, “l” and “o”. On the other hand, when a file is renamed, only one rename action is generated. We cannot afford to have such a relevant action, like renaming a file, to be represented by a single action, while such a common one, like typing text, is represented by five actions. To bridge this gap, we classify actions in two categories: low-level actions and high-level actions. Low-level actions can be combined amongst themselves to generate high-level ones. For example, one key press action can be combined with another key press to generate a type text action. Therefore, if the user types “hello”, those five key press actions will be merged into a single “type hello” action. Nonetheless, high-level actions can also be combined to composite another high-level ones. For instance, two mouse click actions, which result from one mouse down and one mouse up action, can be combined into a single double mouse click action.

Every time the user performs an action, a new low-level user action is generated by the Observer and added to the actions’ history. Upon this addition, a test is made to check if the new action and the last action on the list can be combined. If so, the two are merged, composing a new high-level action. Then, the same procedure is carried out to check if the new created high-level action can be combined with its previous action on the list. This is repeated until there is no further merges between two user actions are possible.

Whenever a new action is created, it is assigned an id and two actions may have the same id if, and only if, they are similar. Two actions are said to be similar if, and only if, it is possible to create a generalization between them. A generalization consists of an expression and a set of functions. These describe the relationship that can be established between two user actions and are used to parameterize future user actions. Therefore, it is easy to note that to each id is associated a set of generalizations.

As a new action is created, the Observer looks up for a similar older action and creates a generalization. If there are no previous generalizations bound to the older action’s id, then the new generalization is validated and the new action gets the elder’s id. Otherwise, the new generalizations and the ones already bound to that id have to be merged. If the merging between those generalizations is successful, the new action gets the id from the old one. If not, then a new id is assigned to the new action.

The next example clarifies the generalization mechanics:

**Observed Actions:**

```
[ID: 13] Type "Hello 1"
[ID: 13] Type "Hello 2"
```

**Generalizations:**

```
[ID: 13] Hello $1, $1(n+1) = $1(n) + 1, $1(n) = 2;
[ID: 14] repeat;
```

In this case, the first and third actions have been generalized, as well as the second and the fourth ones. In the former case, the resulting generalization contains the expression “Hello $1” and the function “$1(n+1) = $1(n) + 1, $1(n) = 2” where “n” is the current iteration number. This generalization expresses a sequential numeric relationship between the two actions, and its expression and function allow Blaze to deduce the next iterations. The reader can easily notice, with this example, that generalizations are the key for inferring user’s future iterations. In the latter case, the resulting generalization only features the “repeat” expression, because “repeating” is the only relation that can be established between two key presses of the same key.

To simplify, only one generalization was shown per id, in the last example. However, in the real system, several generalizations may be created. Generalizations are created, orderly, following a predictive criterion. For example, after two file renaming actions, a generalization that implies that there is a sequential numeric relationship between both old names and a sequential numeric relationship between both new names, will have a higher probability of being right than one that implies that both old names just contain a common set of text tokens. This latter case would have a higher probability of being a misunderstanding.

Generalizations are built based on the Difference algorithm\(^8\) [11], regular expression validation and string and arithmetic operations. These operations are performed over specific data to each kind of user action. For example, in the previous example, the generalization of actions 1 and 3 results from applying the Diff algorithm and a regular expression validation for integers, on the text field of both text type actions. Then, a set of arithmetic and string operations are applied to generate the expression and the functions set.

---

\(^8\) http://www.math.tel.de/Diff/ (Last visited in: June 5th, 2009)
The Apprentice

The Apprentice identifies repeated sequences of actions in the user actions’ history, maintained by the Observer.

As our system is not intrusive, it cannot interrupt the user’s work to ask for approval upon identifying positive or negative examples. Therefore, we needed a different approach to identify which examples are positive examples of a user task, worthy of being automated. As mentioned by Mahmud [9], a good way to compensate for lack of negative examples is to have repeated occurrences of the positive ones. This is, indeed, what we really need. Actions that the user repeats several times have an enormous potential to integrate a repetitive task, which should be automated. Hence, we need a fast way to detect repetitive sequences in the user actions historic.

Assuming that all actions have an id and the actions’ history has a finite size (20 actions in the current version) we know that we will not have more than 20 different ids. Therefore, action ids can be seen as letters from an alphabet, which allow us to treat this problem as a string search one.

The search problem described above can be seen as the Longest Repeated Substring (LRS) problem. We want to find the longest non overlapping substring of ids that occurs in the user’s actions’ history. Thereby, the best solution to solve this problem is to build a suffix-tree [16] and find all of its deepest internal nodes. The edge leading to one of these nodes represents a common prefix of suffixes of the input string. Therefore, we keep track of all common prefixes of suffixes, picking the longest as the longest non overlapping repeated substring. This approach allows us to detect more repetitions, such as constant repetitions and variable and conditional loops, and offer a broader range of options to the user. Although the KMR would be suitable for the task, the suffix-tree approach is more efficient, as it solves the problem in linear time and space.

Imagine that a, b, c and d are distinct user action ids and that the actions’ history contains the sequence “abcbad”, as in the above example. To find the longest repeated non overlapping substring, a depth-first search must be performed, going as deep as the deepest internal node in each path. The edge leading to one of these nodes represents the deepest common prefix of suffixes on a specific path, and each deepest prefix is suitable to be the longest repeated non overlapping substring. Ergo, it is trivial to find out that both “ab” and “b” are common prefixes of suffixes, as the former is depicted by the edge leading to node 7, which is the deepest internal one on his path, and the latter is described by the edge leading to internal node 8, which is also the deepest on his path. Accordingly, being both “ab” and “b” suitable, “ab” is chosen as the best choice, as it is the longest. This algorithm, thereafter, proves to be ideal to find the longest repeated non overlapping subsequence of user action ids. Furthermore, it allows Blaze to detect repetitions even if these do not occur consecutively in the actions’ history.

The Apprentice keeps a list of repetitions sorted by descending order of length and date, which along with the information provided by the Observer, is used to form consistent narratives describing repetitive task automations.

The Assistant

Now that we already have an agent that knows what to automate, we need another one that knows when to suggest. The suggestion must be non intrusive, as the users should not be interrupted of whatever they are doing. The Assistant highlights Blaze’s icon on the system tray and shows an icon on the top-right corner of the user interface. To see Blaze’s suggestions, the user can double tap the CapsLock key or click that icon. Otherwise they can just be ignored.

By pressing the icon or double tapping the hotkey, a new form shows up, displaying comprehensive narratives. Each describes an automation aimed to complete a specific task, identified by the Apprentice and subsequently processed by the Assistant. Narratives are generated based on the most probable generalization for each action id in the repetition.

Imagine that the following repetition was detected:

Observed Actions:

[ID: 3] Type "Hello 1" on window "new 2 - Notepad++" (notepad++.exe)
[ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)
[ID: 3] Type "Hello 2" on window "new 2 - Notepad++" (notepad++.exe)
[ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)
[ID: 3] Type "Hello 3" on window "new 2 - Notepad++" (notepad++.exe)
[ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)

Generalizations:

[ID: 3] Hello §1, §1(n+1) = §1(n) + 1, §1(n) = 3; Hello §1, §1(n+1) = §1(n) + 1, §1(n) = '3';

Figure 3. Suffix-tree representing “abcbad”.

Figure 3 depicts an example of a suffix-tree representing the string “abcbad”. Each edge is labeled with a prefix string and every path from the root node to a leaf corresponds to a possible suffix of that string. Note that squares represent leaf nodes, and circles internal nodes.

http://www.allisons.org/ll/AlgDS/Tree/Suffix/ (Last visited in: June 16th, 2009)
Repetitions:
[3, 17]

Upon being detected, this repetition is processed by the Assistant. A repetition is said to be processed when the Assistant picks the most probable generalization for each action and predicts the next user iterations. As generalizations are already sorted, when bound to a specific id, the Assistant only needs to pick the first one from the array. Therefore, in the above example, the Assistant picks the most probable generalization for action 3 and action 17. Despite of action 17 only having one generalization, action 3 has two distinct ones. The first one, the most probable, treats the changing portion of the string as an integer, while the second one treats it as a single character. The Assistant uses the generalizations to generate the next values, displaying them in the narrative (Figure 4).

A big challenge is to decide how many iterations should be performed for a given repetition. There is no trivial way to decide without resorting to domain knowledge. For most common cases, like the one in the example above, we let the user specify the number of iterations. The narrative changes to reflect updates to that value.

In more complex cases, such as file system operations, we resort to user context information to indentify which folder items have been manipulated and which still need to be in order to fulfill the repetitive task. For instance, imagine you want to organize a folder, “C:\IM”, where you store received files received through your instant messaging client, where many files share a common token, “mary”. You create a new subfolder, named “Mary Jane”, and start moving a generic sample of, at least, 3 files to the new folder. By generic sample we comprise a sample of files that only have in common a specific set of characteristics. In this case, the three files should only have in common the token “mary” in their names. Also, there should be, at least, two different file extensions.

Observed Actions:

[ID: 15] File 'Evil Mary.jpg' moved from folder 'C:\IM' to folder C:\IM\Mary Jane.
[ID: 15] File 'exam scan mary.tif' moved from folder 'C:\IM' to folder 'C:\IM\Mary Jane'.

Best Generalization:
[ID: 15] $\vec{S}_1(n) = C:\IM\*(.*) containing "mary".

This generalization denotes that all files in the folder “C:\IM”, containing the token “mary” in their names, are suited to be moved to the folder “C:\IM\Mary Jane”, regardless of their extension. Therefore, the Assistant only needs to query the context module to gather all file names, contained in “C\IM”, and check which of them contain those tokens. The ones meeting this condition will be added to a list of suited folder items. With this list determined, there is no need to ask the user how many iterations should be performed. An iteration should be performed for each item in the suited folder items list. Figure 5 shows the narrative generated by Blaze Assistant for this example.

It is also interesting to point out that the Assistant allows the user to modify each suggestion, in case that the displayed narrative did not really describe what he was intending to do. As every suggestion is displayed along with a “modify” button, when the user presses it, a new form shows up, which allows the user to browse for alternative actions and to choose which should be performed. These alternative actions result from processing the other generalizations, which were not picked as best probable in the first round.

Figure 5. Narrative example of a file-system automation.

Putting it All Together
Although we have three agents, the Observer is the only one continuously listening for user activity. Every time the user performs an action, an event is raised, which triggers the Observer to generate the corresponding low-level user action object. This agent then tries to merge the new action with the ones already logged, assigning to it an id and the respective generalizations. Thence, the Observer notifies
the Apprentice that an action was performed by the user. The Apprentice, then, uses the Observer’s actions’ history to build a suffix-tree and walk through it, to identify the longest common substrings of action ids. After that, the Apprentice notifies the Assistant which, by its turn, processes the identified repetitions and builds the respective suggestions.

In short, all operations, including the ones regarding periodic action validation, are only performed when an event is raised, which allow us to save a lot of processor time and, therefore, have a fast algorithm.

EVALUATION
In order to validate our approach, we carried out both user and performance tests.

User Tests
We performed user tests in order to evaluate the performance gains, in terms of saved time, provided by Blaze. The tests were carried out in the same laptop machine by 20 test subjects. User had to fill a characterization survey, perform 10 tests representing typical tasks and fill a satisfaction survey.

From the characterization survey, we were able to find out that 60% were man, 85% had ages between 18 and 25, all of them used a computer daily, 90% used Microsoft Windows and only 50% were familiarized with application launchers. From these, only 50% were currently using an application launcher, QuickSilver\[^{10}\] being the most used.

Procedure
The user evaluation was composed by a total of 10 tests, in which the first 8 represented unique recurrent tasks that might arise from daily usage, and the other 2 represented advanced automation tasks that can be performed with Blaze. For all tests, the amount of time and the number of errors taken to accomplish the task were noted down. As errors we considered each time the user reject a suggestion made by Blaze or each suggestion accepted that did not complete the task.

For comparison proposes, for the first 7 tests, the time taken to perform the task without Blaze was also taken into account. The test batch consisted of:

Test 1 – Move all files, which had in common a set of text tokens, to a specific folder.
Test 2 – Rename a set of jpg files according to a specific numeric pattern.
Test 3 – Delete all text files, in a folder, which had in common a set of text token.
Test 4 – Apply bold to all lines beginning with a specific text token in several sheets on Microsoft Excel book. The user should use the redo command in order to repeat the automation for the remaining sheets.
Test 5 – Write in Microsoft Word twenty text tokens respecting a sequential numeric order.
Test 6 – Continue the last test typing additional twenty tokens. The user should use the continue command in order to continue the last automation for more twenty iterations.
Test 7 – Perform a few arithmetic calculations.
Test 8 – Create an email for a person, using contextual commands to retrieve data from the webpage currently being browsed by the user. No time improvements are expected.
Test 9 – Record an adaptive macro using Blaze, move the affected window and reproduce the macro. The macro should perform well even though the window was moved.
Test 10 – Repeat test 4 and modify the automation so that italic is applied instead of bold.

Results
The results achieved are depicted in Chart 1 and Table 1. As we had 20 test subjects, we assumed the sample to have a normal distribution and, therefore, applied Student’s t-test to assess if the time spent to complete the tasks with and without Blaze was significantly different ($\mu_{\text{Blaze}} \neq \mu_{\text{other}}$, 95% CI). The null hypothesis was rejected for all tasks and, indeed, the times spent to perform the tasks with Blaze are likely to be smaller than without Blaze. Dividing both averages, for a specific test, results in the speed multiplier that describes the speed gains.

![Chart 1 – Time improvement resultant from the tests.](image)

Blaze provides the user with a significant speed boost when performing recurrent tasks. Test 8 was not considered for time improvement measurement as it had a different goal. It was aimed to determine how hard it was for the user to remember the contextual commands `this` and `!url`. Users had no problems remembering those, as all users employed them while performing the test. Also, Tests 4 and 6 tested how hard it was for users to remember and use the automation commands `redo` and `continue`. We could

\[^{10}\] http://quicksilver.en.softonic.com/ (Last visited in: September 21\[a\], 2009)
conclude that users had no problems using them. Test 6 demonstrates that the continue command allowed users to perform the task 5.15 times faster. It is interesting to point out that there are tasks that are more recurrent than others which could be accelerated with Blaze with even more significant gains. For instance, imagine that in test 2, instead of 50 photos there were 250 to rename. The user would probably give up. Or if in test 6, instead of 20 more cases we had asked for 300, it would not be feasible without Blaze.

Test 4 was also aimed to test how well Blaze performed automating recurrent tasks that involved the mouse peripheral. This test was somehow problematic, as 25% of the users could not accomplish it. The lower success rate (for all other tests, 100% of users managed to finish them) is due to the fact that Blaze requires the user to be precise in order to precisely infer future mouse clicks. Therefore, users that lack of good aim would induce Blaze to infer sequential mouse clicks with wrong offsets. Besides the low success rate, for the remaining 15 users that accomplished the test with success, they were able to perform 1.45 times faster using Blaze than with any other tool.

On average, Blaze allowed the users to perform 2.98 times faster, which is a very impressive result. Also, it provides a much greater speed boost when facing tasks that are very repetitive, like the ones depicted by tests 2 and 6.

<table>
<thead>
<tr>
<th></th>
<th>Tw</th>
<th>Tw/o</th>
<th>#Err</th>
<th></th>
<th></th>
<th>T2</th>
<th>T3</th>
<th>Tw/o</th>
<th>#Err</th>
<th></th>
<th>T4</th>
<th>Tw</th>
<th>Tw/o</th>
<th>#Err</th>
<th></th>
<th>T5</th>
<th>Tw/o</th>
<th>#Err</th>
<th></th>
<th>T6</th>
<th>Tw/o</th>
<th>#Err</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>Tw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Hit Count</th>
<th>CPU Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blaze - Calling and Dismissal</td>
<td>26</td>
<td>0.596</td>
</tr>
<tr>
<td>Blaze - Index Load &amp; Unload</td>
<td>28</td>
<td>0.409</td>
</tr>
<tr>
<td>Blaze - Predicting Text</td>
<td>127</td>
<td>0.751</td>
</tr>
<tr>
<td>Blaze Indexer - Build Index</td>
<td>29</td>
<td>0.572</td>
</tr>
<tr>
<td>Observer - Monitoring Activity</td>
<td>3729</td>
<td>1.377</td>
</tr>
<tr>
<td>Apprentice - Build Suffix-Tree</td>
<td>2633</td>
<td>0.342</td>
</tr>
<tr>
<td>Assistant - Generate Suggestions</td>
<td>2633</td>
<td>0.355</td>
</tr>
</tbody>
</table>

Table 2. Performance test results.

The heaviest operations that run in the background are, without any doubt, the ones regarding indexation. The reader can notice that building the index got almost 0.6% of CPU usage for only two method calls. That is why indexing is not done continuously and we confined these methods to a standalone executable, Blaze Indexer, which is periodically called by Blaze’s main process. Furthermore, while a new index is being built, Blaze keeps a backup of the old one, making it always available to the user and not affecting Blaze’s performance.

Regarding memory usage, Blaze’s average is around 37 megabytes, although it might vary along with the amount of indexed items. While 37 MB might look too much, when compared to other simple application launchers, like Launcheer, which consume around 6 MB, we have to take into account that Blaze is constantly monitoring the user and needs to store a lot of information. Therefore, when compared to other applications, like Mozilla Firefox, which consumes around 105 MB with only seven tabs opened, we consider Blaze’s 37 MB average to be a very reasonable one. Note that these memory measurements are related to each process’s private working set.
CONCLUSION
Users of current operating systems are demanding more expressive power. This power was formerly granted by the old command line interface and has been lost with the evolution that graphical user interfaces have been through. To fill this gap, a new kind of application arose: the application launcher. Currently available application launchers may simplify the user’s life, but are no match for the unique recurrent tasks that emerge from daily usage.

A few approaches were developed, which resort to Implicit Programming by Example, to learn what tasks should be automated and when to make a suggestion to the user. Although these approaches are based on systems that only automate user’s tasks in a single application, we developed a new system, named Blaze, which is able to automate user’s experience over an entire operating system.

Our solution consists of an application launcher, which enhances its counterparts’ features, and offers additional advanced capabilities regarding user context awareness. Thereafter, Blaze is able to monitor user’s activity, throughout the operating system, and relationships between the logged actions. These relationships are named generalizations. Therefore, a string search algorithm, based on suffix-trees, is used to identify the longest repeating non-overlapping sub-sequences of actions, in the actions’ history. These subsequences, named repetitions, combined with the generalizations bound to each action, allow Blaze to infer future user actions and formulate suggestions.

It is interesting to remark that the users are never interrupted from what they are doing, during the whole process of learning what to automate, and thereby, they do not play an explicit role in the programming process. Furthermore, our tests show that Blaze is able to monitor user activities, using a reasonable amount of computational resources, and is able to enhance user performance, while performing recurrent tasks, by 175%. This means user can perform 2.98 times faster using Blaze, even for simple cases, which is an impressive result.

In the future, we will extend Blaze’s capabilities by making it aware of more contextual information and able to detect more repetitive patterns. A few enhancements can be done in order to allow Blaze to detect nested loops and to be sensitive to a few rare cases that are not correctly detected.

ACKNOWLEDGMENTS
We thank all the users that participated in our studies and the community, which is already contributing to improve Blaze.

REFERENCES
15. Ruvini, Jean-David. The Challenges of Implicit Programming by Example. IUI ’04, Madeira, Portugal, 2004