InspectorWidget: a System to Analyze Users Behaviors in Their Applications

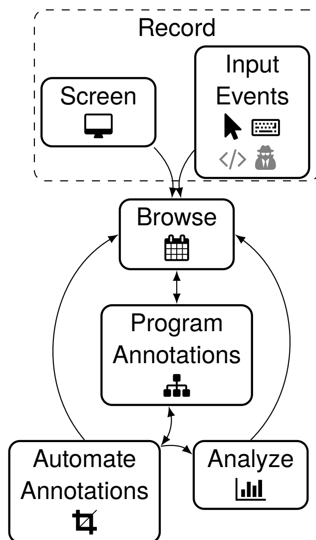

Figure 1: The InspectorWidget workflow: the experimenter records users' activity, then browses the recordings and programs annotations. InspectorWidget then automates annotations that the experimenter can analyze using the provided visualization tool. Data can be exported for processing in a different statistical analysis tool.

Christian Frisson
numediart Institute
University of Mons
Boulevard Dolez 31
7000 Mons, Belgium
christian.frisson@umons.ac.be

Sylvain Malacria
Inria
40, avenue Halley
59650 Villeneuve d'Ascq, France
sylvain.malacria@inria.fr

Gilles Bailly
LTCI, CNRS, Télécom-ParisTech
University Paris-Saclay
46 rue Barrault
75 013, Paris, France
gilles.bailly@telecom-paristech.fr

Thierry Dutoit
numediart Institute
University of Mons
Boulevard Dolez 31
7000 Mons, Belgium
thierry.dutoit@umons.ac.be

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Copyright is held by the author/owner(s).
CHI'16 Extended Abstracts, May 7–12, 2016, San Jose, CA, USA.
ACM 978-1-4503-4082-3/16/05.
DOI: 10.1145/2851581.2892388

Abstract
We propose InspectorWidget, an opensource application to track and analyze users' behaviors in interactive software. The key contributions of our application are: 1) it works with closed applications that do not provide source code nor scripting capabilities; 2) it covers the whole pipeline of software analysis from logging input events to visual statistics through browsing and programmable annotation; 3) it allows post-recording logging; and 4) it does not require programming skills. To achieve this, InspectorWidget combines low-level event logging (e.g. mouse and keyboard events) and high-level screen features (e.g. interface widgets) captured though computer vision techniques. InspectorWidget benefits end users, usability experts and HCI researchers.

Author Keywords
Logging; automatic annotation; computer vision

ACM Classification Keywords
H.5.2. [Information Interfaces and Presentation (e.g. HCI)]: User Interfaces—*Benchmarking*

Introduction
Interface designers, HCI researchers or usability experts often need to collect information regarding usage of applications in order to interpret quantitative and behavioral aspects from users or to provide user interface guidelines.

Unfortunately, most existing applications are *closed*: source code or scripting support are not always available to collect and analyze users' behaviors in real world scenarios.

Different methods can alleviate this problem such as replicating the application with specific interaction "hooks" [15]. However, implementation is time consuming and the replicated application is often a light version of the real one biasing users' behaviors. Dedicated activity plugins or ad-hoc accessibility monitors can also capture user events at different levels of granularity [5] but several applications do not support these mechanisms. Moreover, these methods are application dependent, i.e. require a specific programming language and/or implement one plugin for each application which is not ideal for complex task involving several applications. Conversely, screen recording while users carry their task [7] provides a lot of information, but requires to carefully watch and manually annotate the video, which is tedious and not precise enough for studying low-level interactive phenomena (e.g., duration of scrolling actions).

We propose the *InspectorWidget*, an opensource application to track and analyze users' behaviors in any *closed* application by combining computer vision based analysis of video-recording with low-level interaction collection.

InspectorWidget has four key features. 1) It works with all applications, in particular closed applications where the source code is not available, or applications that do not offer plugins support. 2) It supports logs and event browsing offering data analysis at different levels of granularity, from low level (mouse, keyboard events) to high level (screen recording). 3) It supports additional post-logging and analysis because it collects *all* possible events *a priori* (screen record, mouse and key events) and that the experimenter defines rules *a posteriori* depending of what the focus of the study is. Thus, the experimenter can run other studies with different focus using the same collected data, weeks or months later, simply by defining new rules and without having to collect data again. 4) InspectorWidget relies on visual programming [14] and thus minimizes the need of of programming skills, making it possible for experimenters with sociological or psychological backgrounds to use it.

To achieve this, InspectorWidget relies on a five-step workflow (figure 1): first, it **records** users' screen content and low-level user interactions (mouse and key events) while they carry their task. Second, the experimenter **browses** this recording and **programs annotations** on it by combining explicit active zones definition (e.g, ''this is the scroller'', ''there is the scrollbar'', etc.) and specifying rules (e.g. ''the scrollbar is clicked''). InspectorWidget then **automates annotations**, allowing to combine the programmed ones with computer-vision and low-level events analysis. Finally, the experimenter **analyzes** the annotations either by using the visualization provided in InspectorWidget, or by exporting data to process it in her favorite statistical analysis tools.

Design Goals: Observing Applications' Usage

To better understand users' need and practices, we conducted an informative study with 4 experts. We interviewed two HCI researchers, one usability consultant and one software developer to cover different class of users. Based on these interviews, we present five design goals and discuss related works according to these design goals (tables 1 and 2 summarize these goals and compare existing systems).

Collect diverse data. Understanding subtle interactions generally requires to study both users' actions (e.g. mouse, key events) and the semantics of the interface (e.g. menu, buttons). It is thus necessary to collect diverse data such as:

- keylogging or macro recording, for low-level interactions such as mouse or keyboard events;

Name		Monitor diverse applications	Collect diverse data	Automate diverse annotations	Browse collected data	Allow iterative monitoring	Minimize programming requirements	Platforms	Distribution
AppMonitor	[5]							Windows	Free to download (gray)
Delta	[12]				Browse collected data				
Morae 3.3	[3]	Monitor diverse applications (gray)	Screencast, Keyboard, Mouse, Document Object Model (gray), Accessibility (gray)	Keyboard, Mouse, Document Object Model (gray), Accessibility (gray)	Browse collected data	Allow iterative monitoring		Windows	With commercial support
Patina	[13]		Mouse		Browse collected data				
Prefab	[9]		Screencast					Windows	Opensource, Free to download
Chronicle/Screencast	[10]	Monitor diverse applications	Screencast		Browse collected data			OSX, Windows	Free to download
Sikuli	[8]			Computer vision		Allow iterative monitoring	Minimize programming	Linux, OSX, Windows	Opensource, Free to download
Waken	[6]				Browse collected data				
InspectorWidget	[]	Monitor diverse applications	Screencast, Keyboard, Mouse, Document Object Model (gray), Accessibility (gray)	Computer vision, Keyboard, Mouse, Document Object Model (gray), Accessibility (gray)	Browse collected data	Allow iterative monitoring	Minimize programming	Linux (gray), OSX, Windows	Opensource, Free to download

Table 1: Comparison between InspectorWidget and related works. Gray items indicate partial implementations or plans for future work.

Data types:
- Screencast
- Keyboard
- Mouse
- Document Object Model
- Accessibility

Methods:
- Computer vision

Design goals:
- Monitor diverse applications
- ... Collect diverse data
- ... Automate diverse annotations
- Browse collected data
- Allow iterative monitoring
- Minimize programming

Platforms:
- Linux
- OSX
- Windows

Distribution:
- Opensource
- Free to download
- With commercial support

Table 2: Legend for comparison table entries

- widgets and states changes logging, for applications implementing accessibility Application Programming Interfaces (APIs) that are specific to operating systems; or applications providing their own information (e.g. Document Object Model (DOM)).
- screencasting, to get a video record of the interface.

AppMonitor [5] combines the two first methods to monitor Microsoft Word and Adobe Reader. However, it requires two different implementations as the "accessibility monitor" differs between these two applications. The Autodesk Screencast based on the Chronicle project [10] is a powerful tool combining the three methods but it is limited to Autodesk applications only. Screen capture software such as ScreenFlow record both video and key events, events can be overlaid on the video but not exported as raw data.

Automate diverse annotations. Combining screen recording with computer vision can be used to reverse engineer the GUI of closed applications. This method has been proven useful for automating GUIs and their testing [8], for better understanding the GUI [6], or for regenerating GUIs [9].

Browse collected data. Browsing collected data is an important part of data analysis. With Chronicle [10], users can visualize all event histories on an interactive timeline and play back the actual workflow. Delta [12] allows to compare workflows through several views, among which a cluster of similar workflows and a union graph of shared commands. Finally, Patina [13] overlays a heatmap of mouse locations directly over the recorded GUI.

Capitalize through a posteriori annotations. During data analysis, experimenters can think of different study that is unfortunately impossible to conduct and would require to re-collect data. Typically, a usability expert could investigate how users display their slides when editing them in Powerpoint, and then realize that she was interested in how users include external materials in their slides. Efficient application usage observation should capitalize on collected data. In that respect, a tool recording all data such as the screen and the entire DOM description of the interface, where the experimenter specifies the focus of the study *a posteriori*, is more powerful and flexible than a dedicated logging plugin (e.g. to study a scrollbar position) or a tool where the experimenter specifies *a priori* what she wants to collect.

Example scenario
Suzanne is a HCI researcher interested in color selection and wants to know which color tool (colorwheel, spectrum, etc.) users prefer when selecting colors. She installs Inspector-Widget on the computers of 2 designers using respectively Photoshop and PowerPoint and starts the record mode. At the end of the day, she retrieves the recordings.

Suzanne then uses InspectorWidget to record herself selecting the different tools on her own computer in both Photoshop and Powerpoint and uses that recording to program the annotations she wants to perform with InspectorWidget. To achieve this, she first draws a bounding box around each tab of each color picker windows when selected. She then uses visual programming to program InspectorWidget that automatically annotates users' recordings to find the exact time a specific color tool was used.

Suzanne now wants to analyze the annotated data in R. She exports the annotated data from InspectorWidget and imports it into R.

Minimize programming requirements. Experimenters, analysts or usability experts are often not computer scientists. They are thus not always familiar with programming language. Even computer scientists are not familiar with all programming languages. A software observing usage application should minimize the needs of programming skills to favor immediate usability and wide adoption.

InspectorWidget

In this section, we describe the concept and implementation of our application.

Concept

Analyzing user's behaviors with closed applications using InspectorWidget follows a five-step workflow: 1) record activity, 2) browse and 3) program annotations, 4) automate annotations and 5) analyze the results (figure 1).

Record activity. First, the experimenter records the screen and logs events to track users activity. Screen recording and interaction events logging are usually separated into two tools, making the observation process cumbersome and the collected data harder to synchronize. We solve this problem by proposing a cross-platform desktop tool that both records the content displayed on screen(s) and logs low-level interaction events (mouse and keyboard).

Browse and Program annotations. The experimenter then *programs* annotations. In practice, she browses a recording to find a time frame where the system was in a specific state she is interested in (figure 2). She then draws bounding boxes on widgets of the recording for specifying a center of interest, and creates *rules* on this object with a dedicated syntax (figure 3). Using our example scenario, the experimenter wants to know which color selection tool users used. She first browses the recording until the color picker window of Powerpoint is displayed; she then draws bounding boxes around all of the five color tool buttons (**A*** on figure 2) for each selection state (ColorWheel, sliders, spectrum, etc.). Because she is only interested in when users were interacting with a color tool, she wants InspectorWidget to annotate only when the mouse pointer was located over the color tool window. In that purpose, she also annotates the color dropper **(B)**, the window buttons **(C)** and the window title **(D)**. These graphical widgets are always displayed in the color tool window **(E)** and can be tracked using visual programming (figure 3) to retrieve the size and location of the window even if it has been moved or resized.

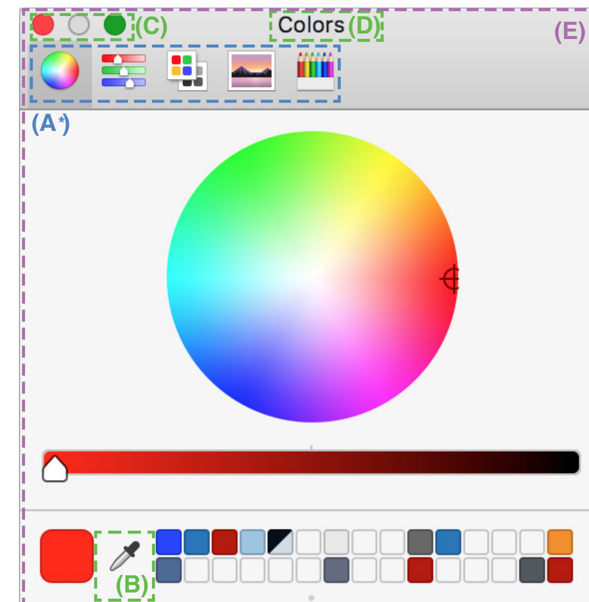

Figure 2: Programming annotations (concept): the experimenter defines an active region for the five color tools selection (A*), and then defines active regions for the color dropper (B), the window buttons (C) and the window title (D) that are used to retrieve the window (E) size and location on screen.

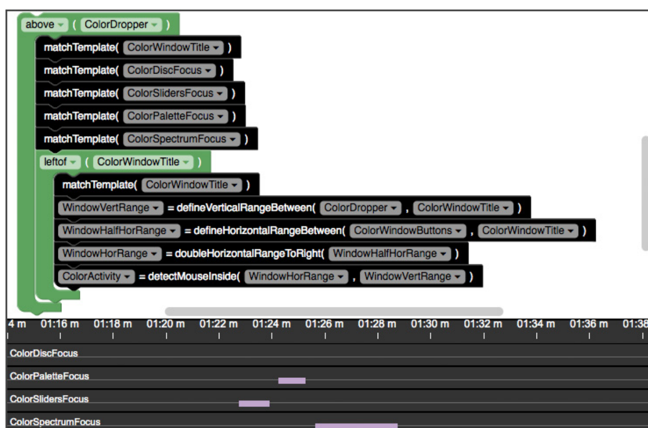

Figure 3: Programming annotations (implementation): on the top the experimenter creates rules using a visual programming language to program the automated annotations, on the bottom the annotated timeline reveals from 1:22 that the user toggled colors tools from sliders to palette then spectrum but not disc.

Note that browsing the video to locate the time where the interface is a in specific state (e.g. the color picker is displayed) might be a difficult task because the experimenter does not know exactly when the interface was in this state, or because it is possible that the interface was never in this state. For this reason, the experimenter sets the interface in the correct states on her own computer and then records it. She can also program the annotations and use these as an input for InspectorWidget to annotate all users' recordings.

Automate annotations. Once all annotations have been programmed, the experimenter clicks on "automatic annotation" to instruct InspectorWidget to start the automatic annotation process, using computer vision and the recorded low-level and programmed annotations as inputs (see below for implementation details).

Analyze. The experimenter can directly visualize simple statistics before exporting results. InspectorWidget produces interactive plots on several extracted and nested metrics, such as the time the mouse pointer was over a widget. Data can also be exported as csv files to be processed in the experimenter's favorite statistical analysis tool.

Implementation

InspectorWidget is composed of a cross-platform **desktop application** for *recording*, a **dedicated server** for *automating annotations*, and a **web-based application** for *browsing* the recordings, *programming annotations* and *analyzing* the results. InspectorWidget is released as an opensource project under a GPLv3 license, available on github[1].

Recording (standalone application). Our tool extends application obs-studio [2] with a plugin for logging input events (mouse and keyboard) using the libuiohook [1] library. It supports recording multiple screens and cameras, all mouse and key events, and audio.

Browse recordings, program annotations. Browsing recordings and programming annotations is achieved through a dedicated web-based application, based on the amalia javascript video player [11] for browsing and enriching the video with annotation, and based on Google Blockly (inspired by Scratch [14]) for programming the annotations.

Automate annotations. The automatic annotation of the recording is performed by a dedicated server, using OpenCV for template matching and tesseract [4] for text/number detection.

Analyze. The aforementioned web-based application also provides simple data visualization by displaying charts rendered using the d3.js javascript library.

[1] http://github.com/InspectorWidget/InspectorWidget

Discussion and future work

We see several directions to improve these works including technical considerations, novel features and user studies.

Technical considerations. A limitation of our approach is that it relies on computer vision which can require time to process large screen videos. We plan to speed up this computational phase with GPU/Parallel processing since we can process the video frames in a non-linear sequence and after the recording. We also plan to pre-process the video in order to remove sequences without users' actions. This will improve both human processing (less video to watch) and computer vision processing. Finally, we plan to delegate some video-based analysis (eg. position of a pop-up menu) to accessibility tools when these are available on the target application / operating system. Indeed, extracting DOM data can provide information about the hierarchical structure of the widgets and accelerate the computer vision process.

Features. Extracting DOM information will also provide an additional level of granularity between low-level events (e.g. mouse and key events) and high-level events (video screen) by considering the attributes of graphical widgets. We also plan to extend our software from desktop workstation to mobile devices.

Evaluation (of annotated data). As every computing system relying on computer vision, InspectorWidget is likely to result in errors, either by missing parts of the videos that should be annotated, or by incorrectly annotating parts that should not. We thus plan to run an evaluation with various use cases in order to quantify the accuracy and precision of InspectorWidget, assessing its efficacy in terms of usability.

Evaluation (through annotation tasks). We also plan to conduct a user study with HCI researchers and usability experts to evaluate the qualitative and quantitative benefits of InspectorWidget over manual annotation, assessing the efficiency of our system and the satisfaction of its users. One of the primary motivations for creating InspectorWidget was to support longitudinal "in the wild" studies concerning our own research projects. Reciprocally, the analysis of such upcoming studies will provide insight on the strengths and shortcomings of InspectorWidget.

Acknowledgments

The authors would like to thank Nicolas Riche for bridging their collaboration. This work has been partly funded by the Walloon Region of Belgium through GREENTIC grant 1317970 for project SONIXTRIP. The authors would like to thank the SONIXTRIP project partners for their feedback while developing InspectorWidget: Dame Blanche post-production studios, Océ Software Laboratories and IRiSib Laras lab.

REFERENCES

1. libuiohook. (2016). **https://github.com/kwhat/libuiohook**
2. OBS studio. (2016). **https://github.com/jp9000/obs-studio**
3. TechSmith Morae 3.3. (2016). **https://www.techsmith.com/morae.html**
4. Tesseract. (2016). **https://github.com/tesseract-ocr/tesseract**
5. Jason Alexander, Andy Cockburn, and Richard Lobb. 2008. AppMonitor: A tool for recording user actions in unmodified Windows applications. *Behavior Research Methods* 40, 2 (2008), 413–421. DOI: **http://dx.doi.org/10.3758/BRM.40.2.413**

6. Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2012. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 83–92. DOI: **http://dx.doi.org/10.1145/2380116.2380129**
7. Suresh K. Bhavnani and Bonnie E. John. 2000. The Strategic Use of Complex Computer Systems. *Hum.-Comput. Interact.* 15, 2 (Sept. 2000), 107–137. DOI:**http://dx.doi.org/10.1207/S15327051HCI1523_3**
8. Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1535–1544. DOI: **http://dx.doi.org/10.1145/1753326.1753555**
9. Morgan Dixon, Alexander Nied, and James Fogarty. 2014. Prefab Layers and Prefab Annotations: Extensible Pixel-based Interpretation of Graphical Interfaces. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 221–230. DOI: **http://dx.doi.org/10.1145/2642918.2647412**
10. Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, New York, NY, USA, 143–152. DOI: **http://dx.doi.org/10.1145/1866029.1866054**
11. Nicolas Hervé, Pierre Letessier, Mathieu Derval, and Hakim Nabi. 2015. Amalia.Js: An Open-Source Metadata Driven HTML5 Multimedia Player. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference (MM '15)*. ACM, New York, NY, USA, 709–712. DOI: **http://dx.doi.org/10.1145/2733373.2807406**
12. Nicholas Kong, Tovi Grossman, Björn Hartmann, Maneesh Agrawala, and George Fitzmaurice. 2012. Delta: A Tool for Representing and Comparing Workflows. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 1027–1036. DOI: **http://dx.doi.org/10.1145/2207676.2208549**
13. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2013. Patina: Dynamic Heatmaps for Visualizing Application Usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3227–3236. DOI: **http://dx.doi.org/10.1145/2470654.2466442**
14. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. DOI: **http://dx.doi.org/10.1145/1592761.1592779**
15. Joey Scarr, Andy Cockburn, Carl Gutwin, Andrea Bunt, and Jared E. Cechanowicz. 2014. The Usability of CommandMaps in Realistic Tasks. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2241–2250. DOI: **http://dx.doi.org/10.1145/2556288.2556976**