IJDC | *Conference Paper*

Automation is Documentation - Functional Documentation of Human-Machine Interaction for Future Software Reuse

Jurek Oberhauser
OpenSLX GmbH

Rafael Gieschke
University of Freiburg

Klaus Rechert
University of Applied Sciences Kehl

Abstract

Preserving software and providing access to obsolete software is necessary and will become even more important for work with any kind of born-digital artifacts. While usability and availability of emulation in digital curation and preservation workflow has improved significantly, productive (re)use of preserved obsolete software is a growing concern, due to a lack of (future) operational knowledge. In this article we describe solutions to automate and document software usage in a way, such that the result is not only instructive but also productive.

Submitted date 20 June 2022 ~ *Accepted* 13 July 2022

Correspondence should be addressed to Jurek Oberhauser, Guntramstrasse 8, 79106 Freiburg im Breisgau, Germany. Email: jurek@openslx.com

This paper was presented at International Digital Curation Conference IDCC22, online, 13-16 June, 2022

The *International Journal of Digital Curation* is an international journal committed to scholarly excellence and dedicated to the advancement of digital curation across a wide range of sectors. The IJDC is published by the University of Edinburgh on behalf of the Digital Curation Centre. ISSN: 1746-8256. URL: http://www.ijdc.net/

http://dx.doi.org/10.2218/ijdc.v17i1.836

DOI: 10.2218/ijdc.v17i1.836.

Introduction

Emulation is an essential tool for accessing born-digital artifacts but is considered difficult to use due to the technical tasks of installing, preparing and maintaining emulation software. Furthermore, integrating emulation into existing archiving and preservation workflows is necessary for productive use. These technical and integration issues of access to emulated software setups have been addressed in the past and have fostered the use of emulation in digital preservation (Rosenthal, 2015). Still (long-term) usability of old software remains a pressing issue because the software running within the emulators must be operated by (future) users in order to make use of digital artifacts. Interacting with software is essential to make use of the functional and performative nature of emulation. With time, we will likely lose a significant portion of operational knowledge. We can already feel the difficulties of operating once popular but now obsolete operating systems such as early DOS, Microsoft Windows, Classic Mac OS (Apple Macintosh), or Unix systems. Even more difficult to operate are specialized and less popular software products designed for specific administrative processes, business processes, engineering or scientific research which require additional contextual or domain specific knowledge. As part of the Emulation-as-a-Service Infrastructure (EaaSI)[1] program of work, we are developing strategies to support usage of preserved software. In particular, we create tools and workflows for automating common user tasks, such as rendering a digital artifact with a defined software application, but also tasks such as installing or configuring software. Furthermore, we are developing a workflow allowing users to create executable documentation, i.e., capturing and describing software interactions such that they become reusable and re-executable by future users.

Re-using Software

Software products usually ship with documentation, additionally there may exist operational documentation in the form of printed books, online tutorials, blogs, or community forums. From the long-term perspective, these documentations are a necessary precondition and useful but for many access scenarios in the future not solely sufficient. User manuals are written for contemporary users, omitting implicit knowledge on (mostly) basic operation knowledge. For instance, to operate an unknown obsolete computer system, one must start with basic concepts, e.g., locating and starting a program. Every teenager owning a Commodore C64 in the late 80s knew the string LOAD "*",8,1 to load a program from floppy and run it. Similarly, users of DOS or Windows 3.11 systems knew how to find and run a program or how to find and open a file. Even though today's software systems still use similar concepts, operating obsolete systems without "historic" knowledge can be challenging, if not frustrating. To capture such basic knowledge, automation and, thus, (functional) documentation are a further required ingredient to make emulated software setups useful.

There are initiatives to capture user knowledge using screen captures and/or videotaping the user interacting with the computer and/or software (e.g., Lowood, 2011). This is in particular an option for artistic, performative software or to some degree for computer games. The result, however, is quite similar to the aforementioned manuals or books. Since it is difficult to capture which input event caused a specific outcome, it remains to the future user to experimentally mimic the captured information and reproduce the captured workflow.

Additionally, automation does not only matter for documentation purposes. Flexible and adaptable automation scripts are required to cope with a huge number of software products, operating systems, and file formats. Any preserved software setup, i.e., installed and configured software, will eventually require maintenance. Contemporary emulators will eventually become

[1] https://www.softwarepreservationnetwork.org/emulation-as-a-service-infrastructure/

obsolete and have to be replaced. While a new emulator still emulates the same computer platform, some details of the hardware configuration may differ. In most cases and especially with more current systems, the operating system is able to detect these variances and adapts in an automated way. However, even a simple Windows boot cycle necessary to trigger the automated hardware detection and driver installation quickly becomes a huge task if the number of preinstalled software setups is in the hundreds or thousands (the EaaSI emulation network has already configured about a thousand ready-made software setups available) and eventually this process has to be manually initiated.

A quite similar issue exists when creating a ready-to-use software library. Installation and configuration of software is not only a time-consuming manual task but also requires operational knowledge of computer systems and software. Ideally, memory institutions are able to tap into a shared pool of ready-made, installed, and configured software setups to avoid redundant manual tasks, and especially, to compensate for the lack of operational knowledge. However, due to licensing restrictions, it is often difficult or not possible to share (installed) software. In these cases, publishing and sharing a re-executable documentation of the installation may be sufficient, leaving the task of acquiring software (and licenses) as a future task. For certain usage scenarios, simply installing software is not sufficient. For reproducing scientific or digital art software setups, specific installation choices and settings matter as well as additional post-installation configuration steps of the software and/or the underlying system. In these cases, it is even more important to not only carefully document configuration and settings but also to assist the future user to perform necessary tasks to avoid misinterpretations, omissions, or usage errors.

Finally, preserved software may not just be used in an (user) interactive way but could also serve as a tool in a migration or content extraction workflow. Hence, automating even very basic tasks like printing a document (as PostScript or PDF file) or using a specific software to simply save an object in a different file format are sufficient to turn obsolete software into a powerful preservation tool.

In this article, we explore two different options to automate tasks within emulated systems and integrate these tasks into preservation workflows.

Cooperative Automation

Automating (and documenting) interactive software usage could either be done by recording the user inputs or through a descriptive approach, creating an executable script of actions to be carried out. When these automations are replayed, the system in which they are executed never behaves hundred percent identically. Various factors may influence the performance of the system (e.g., CPU speed, performance of storage read/write operations) but also other sorts of non-determinism (e.g., random window placement, random system events, etc.) may occur and make a deterministic playback of automated tasks difficult. Previous attempts to automate recordings suffered from these non-deterministic behaviours (Stobbe et al., 2014). In order to improve a deterministic replay, one option is to make use of automation tools or macro recorders that are running within the system to be recorded. These tools usually cooperate with the (guest) operating system, for instance, to identify windows, buttons, or similar user interface elements but also to verify that desired actions have been executed and yield the expected outcome. These tools, however, have a few significant shortcomings. Firstly, these tools rely on specific operating system features and run inside an (emulated) guest. Therefore, it is required to identify a suitable tool for each software platform of interest and additionally integrate (install) the tool in each environment. Secondly, these tools are specifically implemented not only using platform and software specific APIs but also store automation scripts using a proprietary syntax such that it is difficult to re-use such recordings directly or extract knowledge for re-use.

Since currently available technologies are unlikely to yield deterministic playback of automatically recorded actions, we have been looking for use-cases and domains as well as for suitable automation software that can justify the drawbacks using a cooperative approach.

Deterministic behaviour matters most if a large number of artifacts is to be processed without human intervention or supervision. Hence, an automation script or recording should be text-based, comprehensible and editable without specific software requirements. This way the automation can be flexible enough to either help or inform users to cope with similar operations, e.g., applying the same recorded concept on a different digital object or in a slightly different system. Furthermore, it will become possible to apply adaptations for a specific software or object, e.g., by the workflow engine, before execution. A simple example could be printing the same (rendered) document as PDF with a newer version of a word processor or printing (exporting) a number of different documents using a specific software in an emulated system.

Example Workflows using Cooperative Automation

As an implementation example, we have integrated the system specific automation tool AutoHotKey[2] with the Emulation-as-a-Service framework. We use the latest AutoHotKey version (AutoHotKey 1.0.48.05) that is compatible with Windows 95. This allows us to execute the software in different systems covering Windows 95 to the most current Windows 11. AutoHotKey has already been proposed for curation and preservation tasks (Weidner and Alemneh, 2013), our focus, however, is to abstract and generalize automated tasks as much as possible, firstly, to reduce the number of scripts (and thus documentation and maintenance) and, secondly, to allow highly automated processing of digital artifacts. To achieve this, we have implemented automation tasks as templates[3], such that these are reusable with different software used and independent of the Windows version used. The templates are maintained externally and are instantiated on-demand depending on the chosen workflow, software to be used and user provided digital objects to be processed. For evaluation purposes we have implemented three related example automation tasks, to test the flexibility of this approach:

- file creation: Create a file of each file format that is supported by a specified software;
- file format information extraction: Store all supported output file formats of a specified software in a text file;
- file format migration: "Migrate" one or multiple input files to another file format by using the specified software.

The first example, file creation, takes a path to the software to be run (location of the executable within a configured software environment, e.g., C:\WINDOWS\system32\mspaint.exe) together with a sample input file as workflow runtime arguments. The automation task will then generate all possible file formats that the software can produce. As a result of a task, either a snapshot of the running software environment is created for user inspection or further re-use or, alternatively, the generated data is extracted. This and similar tasks may prove useful to automate the technical description of software products by analysing the output created (e.g., using Siegfried or DROID). The second example, file format information extraction, behaves similarly, however, instead of creating all possible files, it only writes all types into a text file, which can then be further processed by other workflows (e.g., creating metadata). Figure 1 displays the correlation between the Microsoft Paint "Save As" drop-down menu with the result of the file creation and file format information extraction automation tasks: Microsoft Paint can save files in 16 different file formats. Correspondingly, the result of file creation is 16 files. Each filename consists of the original file name (in this case "example"), the file format (e.g., 24-bit Bitmap), a number that displays the order of creation, and finally the file extension. The result of file format information extraction is a text file that contains all entries that are displayed in the "Save As" menu.

[2] AutoHotKey Website, https://www.autohotkey.com/
[3] https://gitlab.com/emulation-as-a-service/experiments/autohotkey/

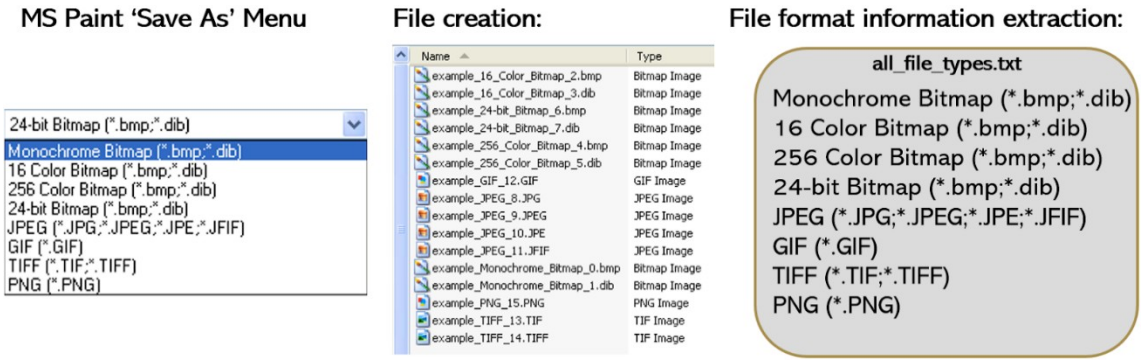

Figure 1. Correlation between the Microsoft Paint "Save As" drop-down menu with the result of the "File Creation" and "File Type Information" automation tasks.

The third example task, file migration, uses software to "migrate" files to another specified format, i.e., it opens one or multiple user-provided files in a user-defined software environment (in case of EaaS the relevant emulation environment ID), the path of the software to be executed, and the target format. Figure 2 provides an overview of the complete workflow for an automation task targeting Microsoft Paint, while Figure 3 displays the corresponding user interface to start such a workflow.

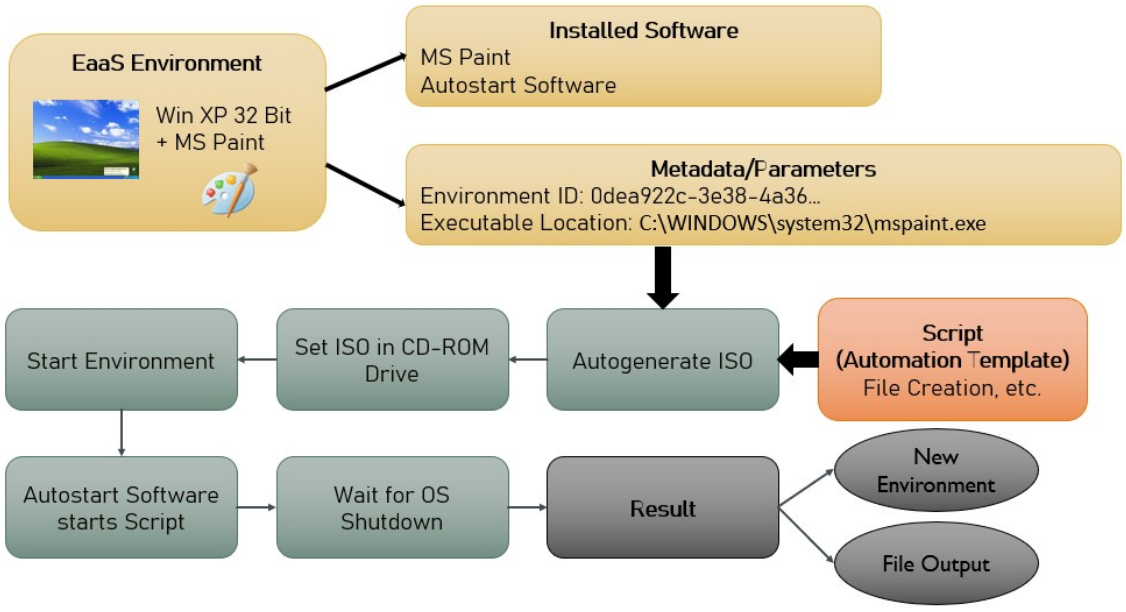

Figure 2. Workflow steps of an automation task targeting Microsoft Paint.

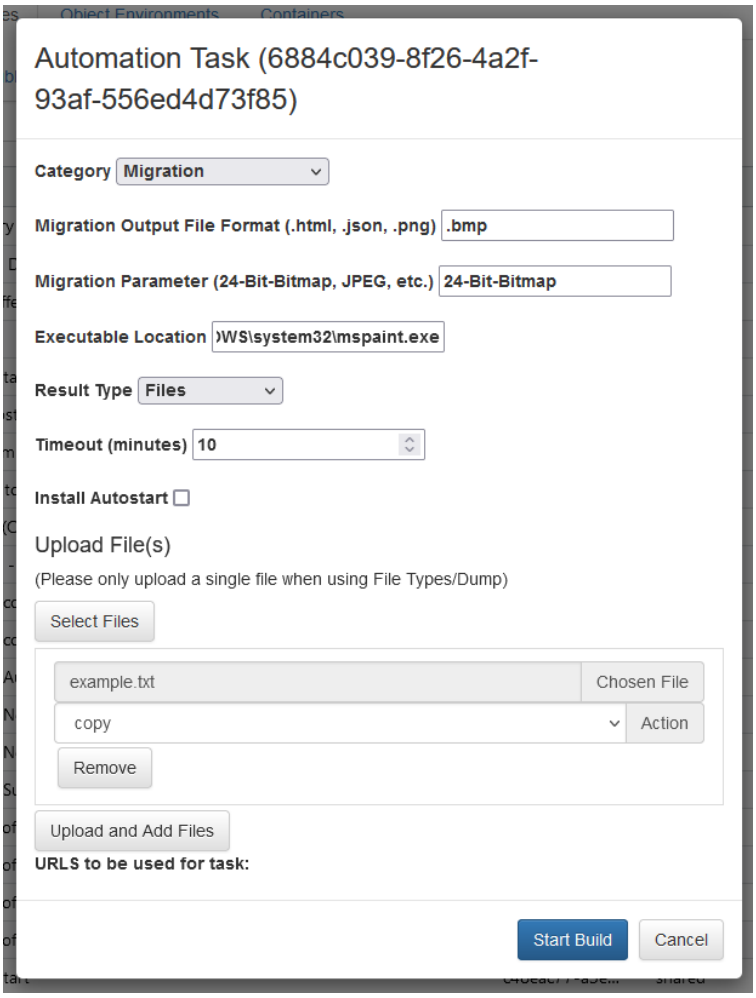

Figure 3. User Interface to configure automation tasks.

All example automation tasks can be invoked through metadata and run in headless mode (cf. Section "Integration with EaaS") to support fully automated workflows, e.g., as batch jobs. Data or other extracted results can be published to a Git repository[4], where follow-up actions can be automatically executed.

To simplify re-use of automation tasks, we created multiple smaller automation templates. The goal is that these can be combined (manually or potentially with software support) to create more complex automation workflows. Currently available software-independent templates include "Run Software", "Open File", "Determine possible output formats", and "Save File".

Such collections of smaller scripts can build the base of an automation library. The goal would be to build such a library for different guest operating system or software, allowing users to quickly configure environments or automate more complex tasks without the need to learn an operating system dependent scripting language first.

Executable Documentation

Using an approach that relies on cooperating with the operating system and/or other software components requires not only the availability of a sufficient tool but also requires the modification of the original software setup by installing an additional tool. In many practical

[4] https://github.com/emulation-as-a-service/File-Format-Information

cases this is not desired or possible. Furthermore, in many cases the documentation aspect is of great importance, i.e., detailed capture of the user's intention (e.g., manually annotated) and detailed capture of interactions to be carried out. Aforementioned automation tools do not always use interaction methods a human user would use to control software, instead, in order to improve determinism, they resort to software interfaces (APIs). Automation tasks built that way offer only limited information for future users about software operation.

An alternative approach is to capture user interactions from human users. Previous attempts have focused on a fully automated recording and replay (e.g., Stobbe et al., 2014). However, not only proved guaranteed deterministic replay of these recordings to be difficult, but a particular problem was also to identify preconditions and the exact expected outcomes of a particular user input. Without knowing the exact correlation between cause and effect, it is not only difficult to implement deterministic replay but also to inform future users on how to yield a desired outcome.

To improve expressiveness, we chose a descriptive approach, allowing users to describe the precondition, input action and desired outcome technically as well as semantically. Capture and replay is therefore restricted to the same information a human user has (i.e., visible screen output) and the same input option. As we only work with visual output however, we can't guarantee a deterministic replay, as there might be certain (pre-)conditions that can't be checked visually.

However, if we can create such highly detailed (replayable) descriptions of user-interactions, we could relax the determinism requirements to gain a more general and more expressive solution, since the main goal is to provide assistive and instructive automation.

Example Workflows using Executable Documentation

As an implementation example, we have extended the EaaS user interface with a toolkit to capture and describe user-actions. The user can describe basic action by defining mouse events (left, right, double click, hovering) or keyboard input to be executed in a defined area. Figure 4 shows this integration of automation recording in the EaaS user interface.

For system synchronization, e.g., to cope with slower/faster systems or random events, the user can choose an area (visual output) as a precondition and wait for it to appear on screen before executing an action. A precondition is currently restricted to the user creating a screenshot and cropping the image to select the area of interest (see Fig. 5). In a future version, we also plan to incorporate OCR text matching. The resulting automation script can then either be saved and re-run within the matching software environment. We currently use SikuliX[5] as a tool to execute the recorded actions. An abstract representation of the actions is transformed by the EaaS backend to a SikuliX script (Fig. 6).

[5] http://sikulix.com/

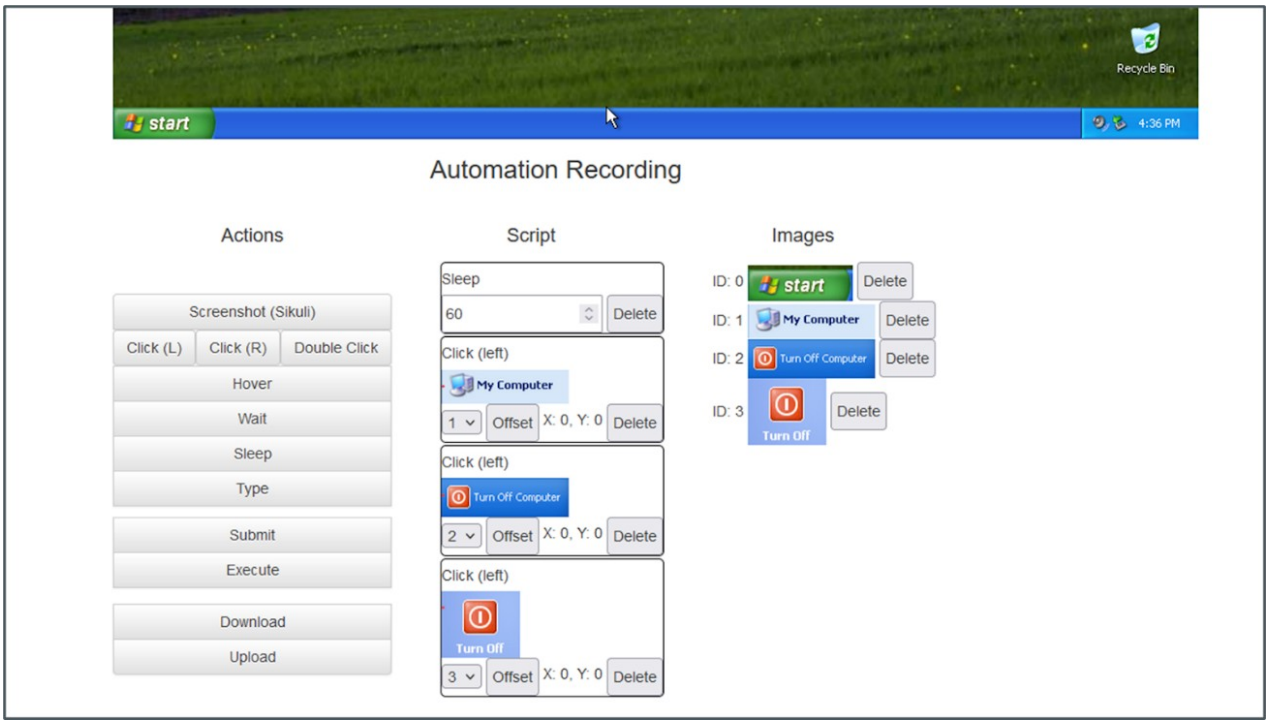

Figure 4. Automation Recording in the EaaS UI.

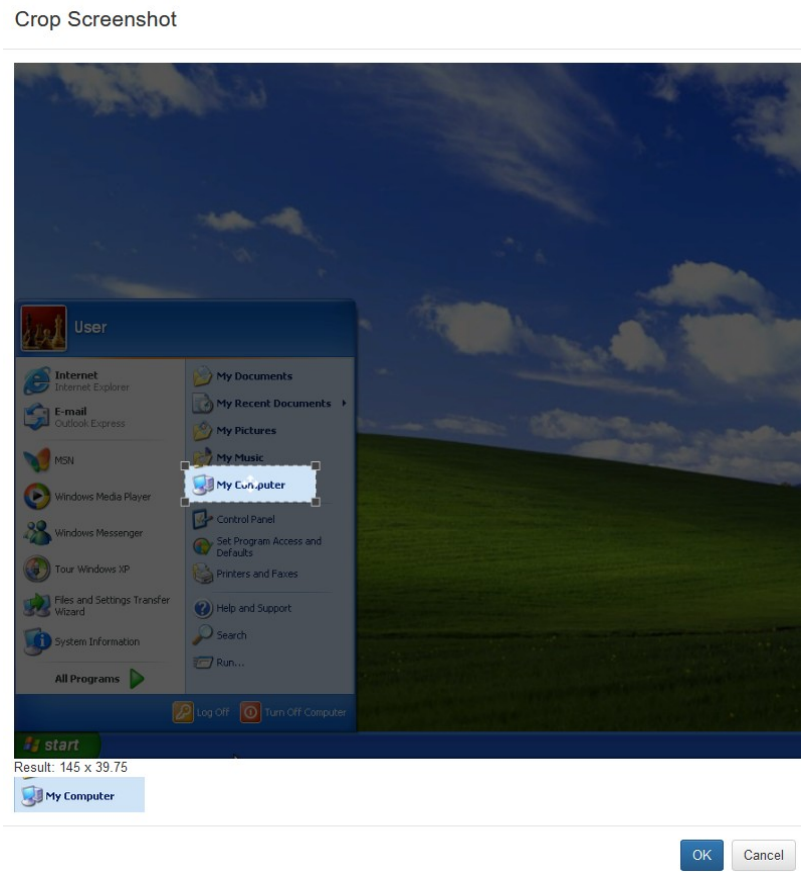

Figure 5. UI for selecting areas of interest.

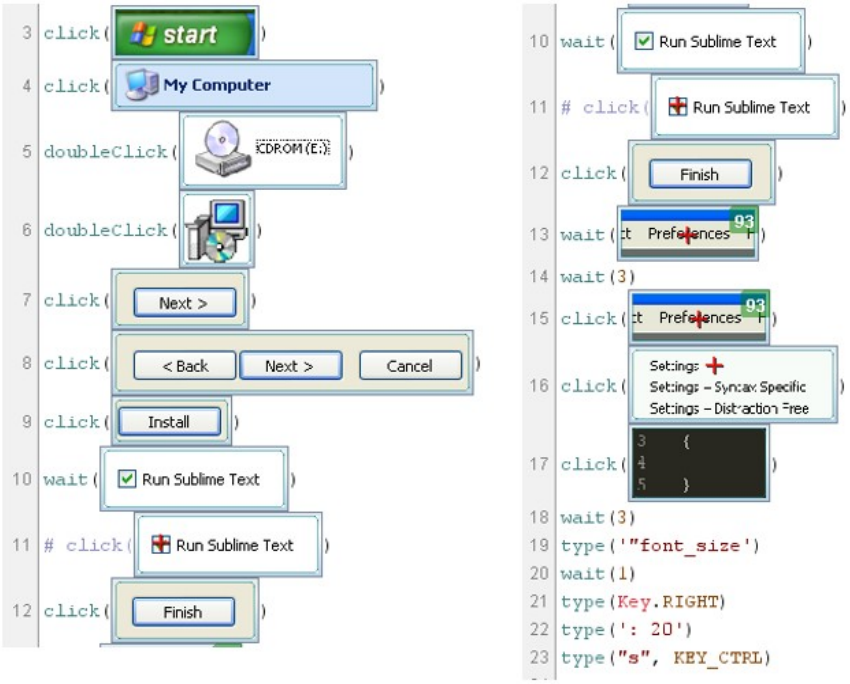

Figure 6. Example installation script.

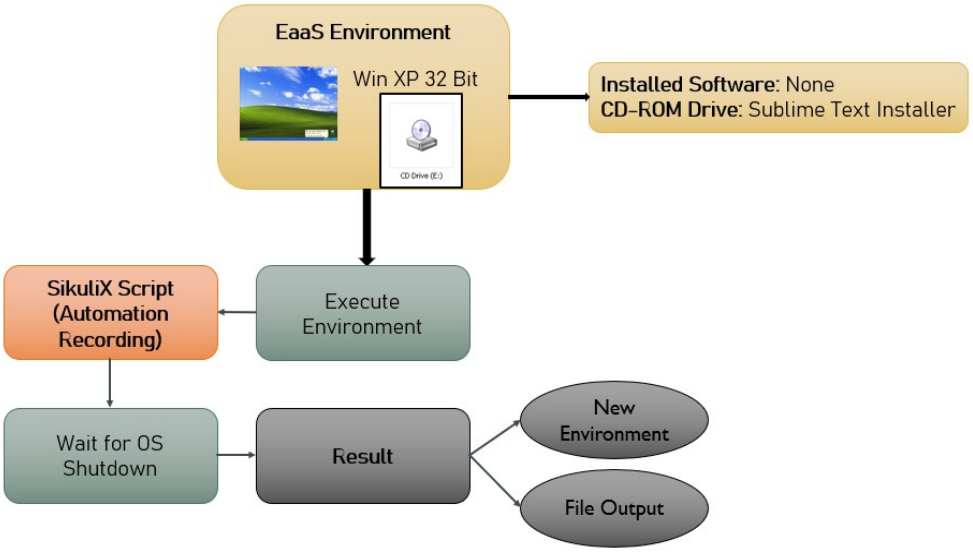

Figure 7. Example workflow for a software installation task.

Captured interactions can be replayed during an interactive session, e.g., to inform or assist users or, alternatively, interactions can be run as automated, scheduled jobs as part of a workflow. The user needs to specify the expected output as well as the captured interactions to be executed.

Figure 7 displays an example workflow for such an automation task. In this example, we use an empty Windows XP environment, where we insert the installer software for Sublime Text using the CD-ROM drive. The example script shown in Figure 6 is executed. After the successful installation, the operating system is shut down. The workflow can then save the state

of newly created software setup (e.g., its disk image). This contains Windows XP with a ready to use Sublime Text installation.

Integration of Automation Tasks with the Emulation-as-a-service (EaaS) framework

We have implemented the execution of automation tasks in the Emulation-as-a-Service (EaaS) framework, allowing a user to execute certain tasks fully automated within emulation environments. Running an automation task involves three steps:

1. the user uploads input files and sets task parameters;
2. while the task is running, its status can be checked in the UI or via a HTTP API.
3. when the task is finished successfully, the result can be retrieved.

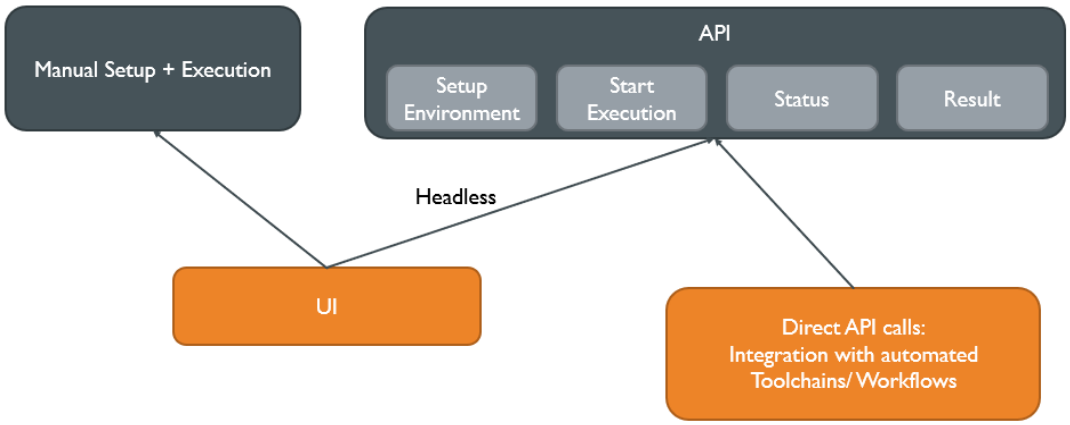

Figure 8. Possibilities to execute automation tasks.

We provide three possibilities to initiate the execution of automation tasks, as displayed in Figure 8: The user can either manually set up the automation task using the UI, observe its execution, and use the software environment before, during, or after execution. Furthermore, tasks can be started in a non-interactive "headless" mode, i.e., no visual output is available during execution but also no user interaction is necessary. The status of tasks can be checked in the UI, and upon completion, the user can retrieve the result. Finally, automation tasks can be executed by directly accessing the automation HTTP API. This allows external components to access automation tasks, which enables the integration into automated toolchains or workflows.

Once all parameters and input are provided, the task is executed and will either produce a valid result, fail, or timeout. Automation tasks can yield one of two result types: The result is either a collection of files or a new emulation environment that is created by saving the environment's state (i.e., its disk) after the automation task has executed successfully. The type of result needs to be specified by the user as different tasks require different results.

Generally, if an automation task creates or modifies files that need to be accessed outside of the emulation environment (e.g., to be further processed as part of a workflow), the files on the disk after the execution are compared to those before it. All new and changed files are then extracted and provided as the output for this automation task.

However, there are multiple cases, where it is desired to save the state of the emulation environment after an automation task, instead of accessing the files directly, e.g., in a case of an installation task. Then, the new environment contains the previously configured machine and additionally the newly installed software.

Conclusion

Our initial experiments and implementation showed that automation can enhance use and re-use of preserved software. Since with currently available technologies it is not possible to implement a platform independent, generic solution that provides guaranteed deterministic replays, we identified domains and use-cases as well as requirements where a platform specific approach outweighs its drawbacks of a non-generic approach. Additionally, we have identified workflows where non-determinism becomes acceptable.

Both solutions are currently integrated into the EaaS framework. Further development is focused to connect multiple automation tasks and to improve the expressiveness of recording executable documentations.

References

Lowood, H. (2011). Perfect capture: Three takes on replay, machinima and the history of virtual worlds. *Journal of Visual Culture* 10(1). doi:10.1177/1470412910391578

Rosenthal, D. S. H. (2015) *Emulation & virtualization as preservation strategies*. Andrew W. Mellon Foundation. Retrieved from https://mellon.org/media/filer_public/0c/3e/0c3eee7d-4166-4ba6-a767-6b42e6a1c2a7/rosenthal-emulation-2015.pdf

Stobbe, O., Rechert, K., & von Suchodoletz, D. (2014). Demonstration of an integrated system for platform-independent description of human-machine interactions. In iPRES 2014 - *Proceedings of the 11th International Conference on Preservation of Digital Objects*. Retrieved from https://hdl.handle.net/11353/10.378720

Weidner, A. J., & Alemneh, D. G. (2013). Workflow tools for digital curation. *Code4Lib Journal* (20). Retrieved from https://journal.code4lib.org/articles/8419