AGH University of Science and Technology in Kraków, Poland

Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science

Automatic Script Generation Based on User-System Interactions

Maciej Golik

Master of Science Thesis in Computer Science

Supervisor: Marian Bubak PhD

Consultancy: Tomasz Szepieniec, MSc

Kraków, September 2014

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

. . . . . .

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI
I TELEKOMUNIKACJI
Katedra Informatyki

Automatyczna generacja skryptów na podstawie interakcji użytkownik-system

Maciej Golik

PRACA MAGISTERSKA
KIERUNEK STUDIÓW: INFORMATYKA

Promotor: dr inż. Marian Bubak

Konsultacje: mgr inż. Tomasz Szepieniec

KRAKÓW, WRZESIEŃ 2014

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiscie i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

. . . . . .

Abstract

Recently, web applications became the trending platform for development of new and rewritten software. Many projects that deal with improving the usability of scientific software focus efforts on web portals commonly known as "science gateways" or "virtual laboratories". Yet still, many users prefer usage of a command line for reasons of speed, advanced features and greater control. Unfortunately, virtual laboratories are usually not compatible with each other (on the level of user experience and data formats) which make it very hard to switch between them in case of a problem or lack of specific feature.

This work presents different approach. Instead of creating a new layer between user and software the proposed solution creates the tracking tool that intercepts communication between a shell and an operating system. Data generated by user and system interactions include the history of executed commands, system calls and shell environment variables. The gathered data can later be used either for analysis and visualization of processes and files or to generate scripts that recreate user's workflow as closely as possible. The tool is supposed to work as the best effort solution and do as much work as possible but allow user to easily verify and manipulate the results. This design features a flat learning curve allowing quick start while providing good results but also providing advanced options for more advanced users and needs. Since interfaces are built using simple text protocols they allow multiple independent implementations for all or only specific modules. The external technologies used in the developed prototype include *strace*, *GNU Bash* shell and *Python* programming language.

This thesis covers all aspects of the tool design. Starting with motivation and background that led to presented solution following with the requirements' definitions. Subsequent chapters cover concept development, limitations and proof of concept implementation. Lastly, the final sections present vision for the future and the summary of the work done.

KEYWORDS: tracking, interaction detection, script generation, workflows, automation, CLI, system calls, analysis, heuristics, user experience

Contents

Chapter 1

Introduction

This chapter presents an introduction to the subject of this thesis. It includes motivation, the description of current trends, the research hypothesis and the detailed goals of this work. It also briefly presents the main objectives of the designed tool.

1.1 Background

The computers as we know today did not change a lot from the time they were created. They changed the form, size and the location where they were stored. They started as machines taking the whole room in the universities and companies and then moved to small boxes in people's houses. The idea of the transistors that build computers stayed the same over the years and only their number on a single chip changed from thousands to millions due to the technology advancements in the process of miniaturization. When this was not enough to push computers further they again started to grow. At that time the technology made a full circle and the computers went back from people's homes to server rooms full of metal, plastic and noise.

Also the information theory developed by Claude Shannon did not change over the years. He came up with the idea that every information can be stored using probability, specifically with probability $1/2$ which is the same as with a coin toss [36]. Although the single bit can only represent two states – true and false, zero and one, black and white – it can be grouped to build more complex structures like bytes. Bytes then can be used to represent any type of information like text, sounds and pictures.

Increasing abstraction level sacrifices the control for accessibility and usability. Every tool imposes another level of abstraction on user although providing specific features. This theory is very evident in comparison of the programming languages [39] where lower level languages are faster in execution and harder to learn while those of the higher level are faster in development

but take more time to execute. High level tools allow users to start quickly while having satisfying results. Unfortunately, as the need for the greater control and efficiency rises user must turn to the lower level solutions.

The scientists are specialists in their respective fields and computers are just another tools required for high efficiency of work. Requiring from the users learning tools built only to create another layer of abstraction is not optimal and should be avoided unless completely necessary. Additionally, when users move between computers the tools they use may not be able to work because of the specyfic environment or missing dependencies. This may require switching to and learning different application.

1.2 Motivation, hypothesis and detailed goals

In order to fix the problems mentioned in the previous section the decision was made to take slightly different approach than most popular solutions which are presented in section 2.3. The motivation of this thesis was to ease use of applications that scientists use on a daily basis by reducing the amount of users' required actions to the minimum. **The solution of this problem was building the tool that will be able to record actions performed by the user, analyse those actions and effects they had on the operating system and files, and finally create script that matches user's workflow as closely as possible.**

The main and the most complex component in the presented solution are advanced heuristics that will be used to parse, analyse and create internal representation. Unfortunately, even the best algorithm will not always succeed or provide optimal data. Because of that the presented tool has been provided as *the best effort* solution and might ask user for guidance in order to improve the results and fix the possible mistakes.

The specific goals the created tool had to fulfil are:

1. Do as much as possible automatically: record actions, match arguments, find relevant data,

2. Have a flat learning curve (or none at all): the tool should be easy to learn but provide great flexibility for advanced users,

3. Work in the environment known to the potential user (no new level of abstraction, transparency): since most programs run in a shell user should not be required to use a browser or GUI for tasks involving their usage,

4. Have no or minimal dependencies (applications, tools, frameworks): this allows the designed tools to be portable and installable without the administrator account,

5. Be portable and designed to use on different machines (PC, supercomputers, without root access): handle multiple environments (paths, file names, variables) which will allow to perform the interaction detection on personal computer but execute the generated script on different machine,

6. Allow the exchange of scripts (and internal representations): use a format that can be easily transferred between users and computers,

7. Give the same benefits as the similar tools that try to solve the same problem but with different approach.

The methodology used to achieve the specified goals consisted of dividing process of the automatic script creation into three parts: tracing, parsing, and producing output (textual analyse, visual representation and executable script generation). The minimization of learning curve could be done by utilizing environment that is already known by users of a supercomputer and scientific software namely the shell (e.g. Bash) along with common commands. All commands executed by the user, running processes and their children, must finally execute a system call. In that moment the designed tool can record interactions between user and system. Those interactions include: launching processes, opening and closing files, and reading and writing to file descriptors. With the information about which process opened which file along with information about what mode (read, write) was used for this operation, the tool can create a *data flow* between different programs. Later this flow can be used to recreate the order of commands executed and the relations between them.

This sort of tracing does not require the adaptation to the created tool from the user. The only required, additional actions are: enabling tracer before starting normal workflow and then using analytic or script generating modules.

Chapter 2

Techniques supporting in-silico experiments

Firstly, this chapter presents the history of the automation and supporting user in optimizing work on computer including social aspect of sharing work and knowledge. Secondly, it overviews current technologies and trends in information technology which can be applied to increase usability of such systems. Next, there are presented current achievements in the field of supporting scientific users in automating repeatable tasks by looking in detail on three such systems: GridSpace, InSilicoLab and Galaxy. Lastly, it points out lacks of competitive solutions and possible ways to improve.

2.1 Human-computer interfaces and user collaboration

The first computers started as mainframes, the big machines shared between multiple users. The way of interaction between users and computers at this time were the punch cards which required high level of skill and carefulness since every mistake could cost hours of time and lots of money used for paper and electricity. Because of that users of the same machine would naturally collaborate, share knowledge and help each other fix mistakes.

The next step in the history was the creation of the terminals. At this time users did not have to be in the same room as the computer because it could be accessed remotely with simple but powerful text mode. This simplification allowed to draw more people to write programs and quickly test and fix mistakes. The remote access greatly reduced entry barrier but lowered the level of collaboration.

Another big step in simplification of computer usage was the creation of the graphical interfaces. This idea opened the concept of computers to everybody by minimizing the learning curve and allowing more people to access,

learn and use the new technology. Graphical programs allowed people without low level knowledge to calculate using spreadsheets, write texts using word processors and paint using graphic programs.

The technology advancements allowed everyone to have a personal computer. They were less powerful than the mainframes but allowed users to work whenever they wanted and without competing for the system resources. This again lead to collaboration reduction because people were keeping all their work, results and scripts on private disks. Only necessary code was shared because of the limitations of transfer methods and lack of vision how their work could help someone else.

The latest presented revolution was the creation of Internet. World Wide Web opened the way to share code, results and knowledge fast and easily with anyone. Now Internet is full of shared scripts and codes, sites like Github [20] with the slogan "Social coding" make big difference in getting started and learning programming.

Unfortunately, they do not fix all the problems. Scientists are rarely found on those pages and when they do they share code that is very specific to their problem. This basically requires the potential user to learn technology and get through the specific problem before he or she can use the found program.

2.2 Trends, technologies and models for software development

Currently, the most rapidly evolving technologies are those based on the web browsers and engines, namely HTML5[1] for layout, CSS3[2] for styling, and JavaScript[3] for manipulation. Those technologies, although created for use on web pages, are now commonly used for desktop tasks. This is possible by utilizing the browser engines: Gecko for Firefox[4], Trident for Internet Explorer[5], WebKit for Safari[6], and Blink for Google Chrome[7]. Web technologies can be used for example for desktop theming like in "Modern" mode of Microsoft Windows[8], Gnome Shell[9] and other. Node.js[10] is using JavaScript engine of Google Chrome to support usage of JavaScript for server side scripting to allow front-end programmers to work on back-end related work.

[1] https://developer.mozilla.org/pl/docs/HTML/HTML5.
[2] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3.
[3] https://developer.mozilla.org/pl/docs/Web/JavaScript.
[4] https://www.mozilla.org/firefox/.
[5] http://windows.microsoft.com/en-gb/internet-explorer/download-ie.
[6] https://support.apple.com/kb/dl1531.
[7] https://www.google.com/intl/en/chrome/.
[8] http://msdn.microsoft.com/en-us/library/windows/apps/dn465800.aspx.
[9] http://www.gnome.org/gnome-3/.
[10] http://nodejs.org/.

The GUI programs allow new users to quickly start using them. Web technologies allow them to be simpler to create and extend. They fulfil promise that Java could not keep: "Write once – run everywhere" [7]. Thanks to this, the programmers do not have to worry about portability, look and technical details and instead they can concentrate on the most important aspect: functionality.

The current trend for software development is making it as close to the subject as possible. It can be achieved by creating programs with close collaboration with users or, thanks to the web technologies, creating them by ourselves. The result of this approach is a great choice of applications, domain specific programming languages (DSLs) and tools. Those programs are often created for a single use, project or person which allows them to be used with the greatest human efficiency. Unfortunately, this user-centric approach hurts portability between users and use cases.

A simple modification to make use of domain specific functionalities often requires full application rewrites. Every tool has its own design and user experience which makes it harder to switch between the competitive solutions. Additionally, those applications are often abandoned after the project has ended or author does not have time for the development because they are created by a single team or person for a single use case. This approach wastes human power behind them and limits the innovations that could be made by collaborating on a common project.

2.3 Virtual laboratories

One of the solutions for steep the learning curve of the command line and low level programming languages are the "workbenches" like Mathematica[11] and MATLAB[12]. Those are desktop programs that target computational sciences and provide a simple syntax similar to the natural language. Their distinguishing features are: visual feedback, graphical representations, hints, history tracking and sharing. On the downside, they target only specific domain making them unusable for different types of computations they do not provide.

Another solutions for steep the learning curve are those using web technologies: *Virtual Laboratories* or *Science Gateways*[13]. There are multiple solutions distinguished by a different range of features, licences, governance model, and targeted use cases. The next sub-sections present three repre-

[11] http://www.wolfram.com/mathematica/.

[12] http://www.mathworks.com/products/matlab/.

[13] Following [35]: "A Science Gateway is a community-developed set of tools, applications, and data collections that are integrated through a portal or a suite of applications. Gateways provide access to a variety of capabilities including workflows, visualization, resource discovery, and job execution services".

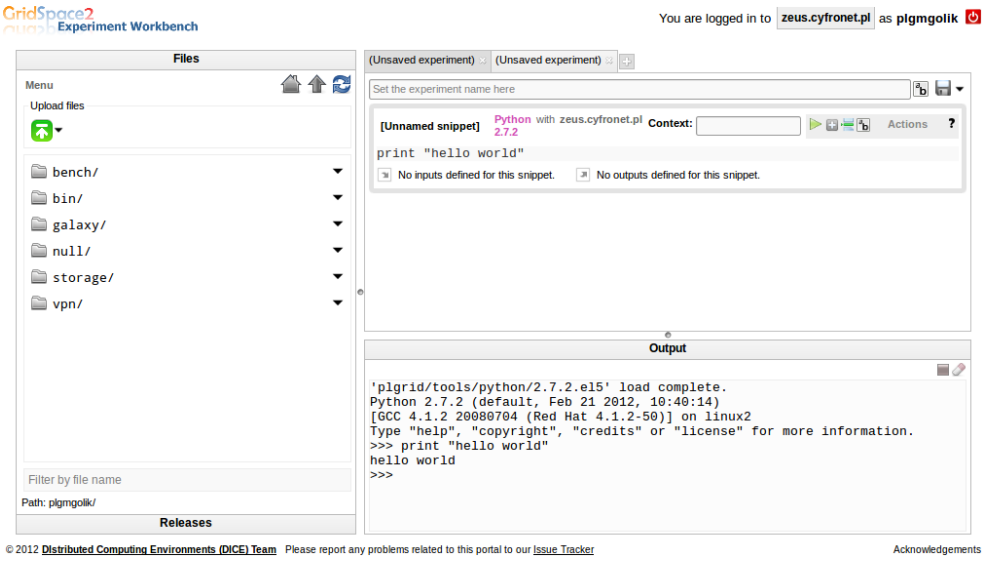

Figure 2.1. Screenshot of GridSpace interface [2]. It shows two columns: files and experiments. Experiments columns are tabbed and each tab contains multiple snippets and output of executed snippets.

sentatives of this category differencing in accessibility, targeted audience and provided features: GridSpace, InSilicoLab and Galaxy.

2.3.1 GridSpace

GridSpace is a "novel virtual laboratory framework enabling researchers to conduct virtual experiments on Grid-based resources and other HPC infrastructures. (...)" [2][6][5][3][24].

GridSpace was created as the generic tool that will allow scientists to access computational resources with ease and consistency. The main windows shown in Fig. 2.1 present the two column layout. The left column allows browsing files on the connected cluster while the right column displays the tabbed list of experiments. The experiments are divided into two rows: the snippets with all source codes and the inputs/outputs management, and the lower row where the output of commands is shown.

The history of commands and snippets can be saved and executed multiple times. GridSpace offers its users the high level of control and flexibility by allowing usage of many scripting languages like Bash, Python and Ruby. That means it can be easily used for working on multiple tasks but will never compete with tools created specifically for one problem.

The technology used for the implementation includes Java and SSH both giving the developers great deal of flexibility and compatible tools.

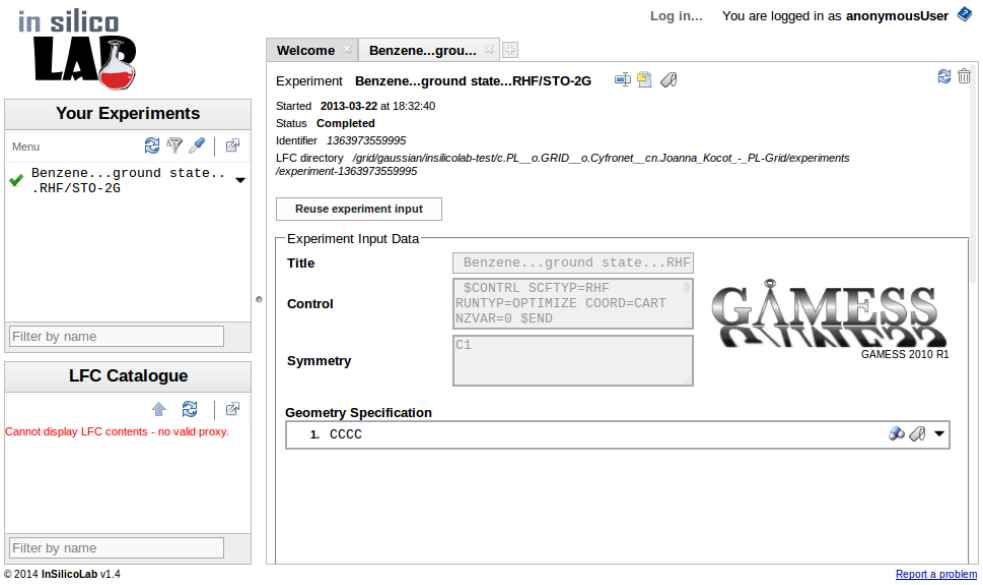

Figure 2.2. Screenshot of the InSilicoLab interface [25]. The screen is split into 3 parts: experiments, history and management; LFC file browser and tabbed experiment details.

GridSpace currently can be executed on multiple clusters, is actively maintained and new features are added when needed. It is a mature software that can easily be customised and integrated into the new tools. The layout and features resemble the tools like Mathematica and MATLAB.

2.3.2 InSilicoLab

InSilicoLab is "a framework of application portals that support e-science research by facilitating the access to computational software deployed on distributed computing infrastructures and the management of data and processes involved in such scientific computations. (...)" [25][29][28][14][27].

InSilicoLab has its roots in GridSpace framework. It was built using the same technologies but with different purpose in mind. The goal was to create experiments for a single use only thus giving users most feedback and maximally accelerating their work.

The main windows of InSilicoLab visible in Fig. 2.2 consist of the three regions: on the left the executed experiments and the list of files while on the right the experiment tabs. The experiment list panel shows currently running, finished and cancelled experiments. This window may later be used to check the results or execute the jobs again.

The experiments in the InSilicoLab portal are created for a specific use

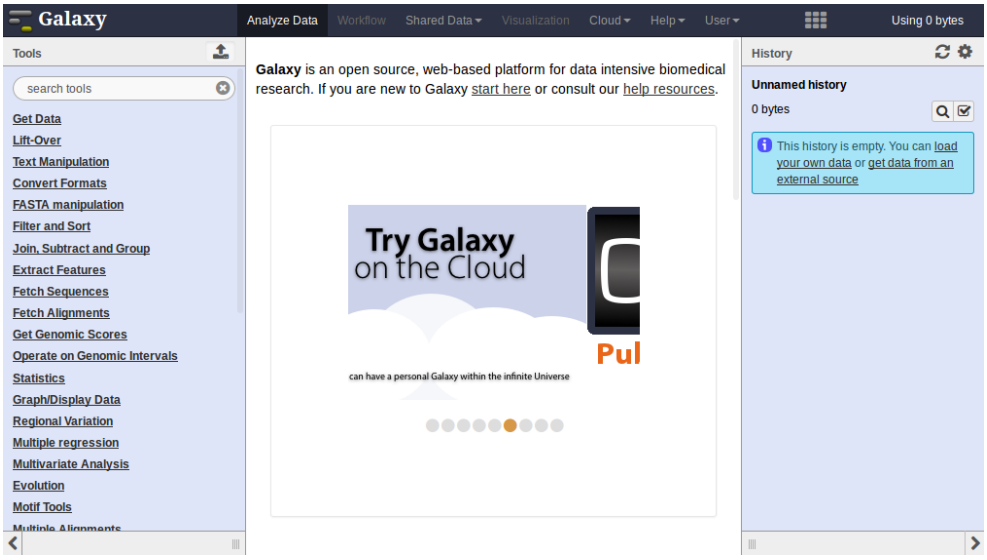

Figure 2.3. Screenshot of Galaxy interface [19]. Shows three column layout: left with the list of tools, center with the tool content, forms and results, and right with the the history of the run commands and options.

only. Currently, there are three deployed portals: for chemistry, CTA[14] and astronomy. The each one is customised to fit various needs of the users, e.g. the chemistry version provides users with charts and tables of energies.

InSilicoLab uses Java, gLite and DIRAC to run jobs on clusters in grid infrastructure. Downside of the user centric approach is the creation of new experiments requires experienced developer. On the other hand, the results of customized portal can be very rewarding with fast and easily manageable results. Unfortunately, InSilicoLab does not currently allow the creation of workflows, so for experiments that have connected inputs there is no solution other than creating septate jobs and providing data manually.

2.3.3 Galaxy

Galaxy is an "open, web-based platform for data intensive biomedical research. Whether on the free public server or your own instance, you can perform, reproduce, and share complete analyses." [19][22][12][4].

Galaxy project was created to support biomedical studies. The web portal has a three column view and can be seen in Fig. 2.3. The left column contains tools categorized in sections. Those tools are installed by the project administrator who can control which tools can be used on their instance. The right column represents command history where all executed jobs are displayed.

[14]Cherenkov Telescope Array.

When job starts user can cancel it or view its current status and output, or execute it with a different set of parameters. The history can be saved for the later use and usually represents user's single workflow. The column in the middle contains the detailed view of the selected tool. At the beginning it shows a form with the specific fields for inputting information. While the tool is running it shows the details of a job and provides access to standard output and error streams.

Galaxy allows users to create their own tools and share them on the common website, where administrators can find them and install on their local instances. Those tools can then be used standalone or, if developer prepared them, as the parts of workflows. The workflows are created by the "drag and drop" technique, similar to the component programming: users draw the connections between outputs and inputs of the tools. Those workflows can also be shared later.

2.4 Drawbacks of presented tools

This and the previous chapters show that creation of a tool that is at the same time easy to use, powerful, and portable between users and environments is a non trivial task. Usually, the designer has to choose if the tool will be effective but targeted only for the limited use cases or provide advanced customization options at the cost of requiring prior knowledge. The best option is to create the tool that can be simply learned, but also easy to extend and modify like for example embedding C code in Python for optimizing computation intensive parts.

All of the mentioned tools in section 2.3 provide users with the assortment of great features. They all ease access to computational resources, underlying software and technologies but at the same time they make users dependent on their solution and loose control of underlying layers like a shell. Being dependent on a specific software can cause problems when the goals of the authors no longer match needs of the users for when the project development is cased.

In case of InSilicoLab and Galaxy when the new software or software version shows up the users have to wait for the developers to add support which can take hours, days or months. If the user depends on the bleeding edge software, he or she has to temporary (or permanently) switch to the console, defeating the purpose and dismissing the effectiveness of those tools. Even if this problem does not appear now as the software is actively developed it is not guaranteed to last long enough and requires users to trust and rely on the specific solution.

InSilicoLab does not currently provide any community features while Galaxy only allows sharing of the workflows built from the building blocks that were already provided by the developer. According to the created re-

quirements GridSpace comes the best. It is built with collaboration in mind and allows sharing work between groups of users. It provides users with the access to low level tools but eases this process by simplifying the access methods, providing graphical wrappers and automating common tasks. Unfortunately, it does not provide domain-specific features, does not allow simple creation of the workflows, and provides no easy way to share them.

Chapter 3

Functional and non-functional requirements

This chapter presents functional and non-functional requirements needed for the correct implementation of the proposed system. Functional requirements are split into the functional modules allowing easier implementation and forcing better software quality from the viewpoint of software engineering. Next, use cases needed for the validation of the tool are presented. Lastly, non-functional requirements which should be taken into consideration while developing software for the better use of available resources are listed.

3.1 Functional requirements

The overview of the tools presented in the previous chapter allows the creation of fields of improvements and the creation of completely new solution based on the re-imagined concept.

The specified requirements are similar to the goals defined in section 1.2 and were as follows:

1. It should be transparent to the user: do not interfere with the normal user's worklow and instead trace and analyse,
2. It should require the minimal prior knowledge,
3. It should be automatic, but not smarter than the user: just the best-effort solution,
4. It should support and embrace collaboration,
5. It should analyse what the user is doing and derive knowledge from his or hers actions,

6. It should allow to recreate user's actions as closely as possible: with the special filters and use of knowledge like repeated actions, overwrites, etc.,

7. It has to transform specific solution to the more generic one.

Those points sum up to the one simple idea: the new tool should transparently learn from user's actions, made in his or hers natural environment, and then recreate those actions by making the generic and parametrized internal representation, that can be later shared, modified and transformed into a specific script.

The following sub-sections divide functional requirements into categories that were later used to create the proof-of-concept implementation and set the direction on the future development. The carefully conducted analysis allowed the creation of the tool that meets the needs of users, does not contain fundamental flaws and allows future extensions [41].

3.1.1 Transparent tracing and best effort automatization

The main objective of the designed system is for it to be as much as possible transparent to the user, so it will not disturb his or hers daily routines. The tracing tool is supposed to gather all required data without forcing the user to run special commands, make forced stops or start over in case of made mistakes. The tracer should only capture all user actions along with the additional meta-data and analysis of the gathered data should be made later.

The automation of the tools relies heavily on the heuristic engine that will make the decisions about parameter matching, data flow detection etc. Since no algorithm can match all use cases it makes the designed tool not fully automatic but rather the **best-effort** solution. In order to cover as much as possible use cases, the tool should not forbid user from helping in making decisions and should allow to make manual improvements, and direct the tool in desired course by using special directives.

Despite the tracking executed processes and opened files the tool should be able to match the parameters of consecutive commands, allowing them to be set to variables and in addition have the ability to run parameter sweep on those arguments[1] in the most optimal form (script, array job[2], etc).

3.1.2 Requirements for post tracing

There are two goals of the analytic tool. The first is to present **relevant** data retrieved from parsing to the user and providing user with the **information**

[1] Parameter sweep means running the code multiple times using unique sets of input parameter values [23].

[2] Array jobs are parts of Job Arrays, that allow user submitting multiple sub-jobs performing the same (while using the same script.), but operating on many data sets [8].

gained by running heuristic engine on output of the tracer program. The information should include, but not be limited to: the opened files and their modes, the amount of data read/written, the number of missing files, the throughput of those files and the frequency of I/O operations. Optionally, user should have access to the representation of the data flow in the text and/or graphical form.

The second role of the analytic tool should be the manipulation of the created internal representation of the data flow and process dependencies. The manipulation should be manual (adding file or program nodes by hand) or automatic (merging two alternative program executions).

3.1.3 Possible outputs of script generation

There are two separate requirements for generated scripts: the language of generated script, and the more important: the type of generated script. The simplest language to generate script is the same language from which the data was gathered. After detecting program flow and matching parameters only pasting of commands with substituted values is required. Implementing generation for other languages is a simple task and it only requires the usage of language specific functions like "subprocess" from Python [16] standard library.

The possible three types of generated scripts are as follows:

- Executable: this is the script that simply reruns *relevant*, user executed commands,
- Batch: the same as above but with the additional "PBS" directives[3]. More can be found in the section 3.1.4,
- Script-generating scripts: the scripts that will generate executable or batch scripts with parametrized values. More can be found in section 3.1.4.

3.1.4 Aiding user in the effective supercomputer usage

Flawlessly migration from the local computers to the supercomputer and/or improvements of the usage of shared resources can be achieved by generating batch scripts (as stated in the section 3.1.3) and script-generating scripts with optimized directives. The optimization to "walltime" and "resource" directives (setting them to the values closely matching real run time allows shortening queue times for all users) can be made by analysing run time of a program, spawned processes, run time depending on the input data size and the additional metadata specified by the cluster administrator (like suggesting the type of node for the specific executable). Also the frequency of I/O

[3] *Portable Batch System* is described on the official site of Adaptive Computing [9].

operation and the amount of the data read can be used to suggest the type of storage on which the data should be stored, for example Lustre for frequent operations and GPFS for storing final results.

The role of script-generating script is to make parameter sweep, generate batch scripts and run them on the proper nodes. The parameter sweep should be made intelligently by utilizing provided features of the resource managers when possible (for example TORQUE provides "array jobs", described earlier) and falling back to generating multiple scripts when no better solution was found.

3.2 Use cases for the validation

In order to ensure the proper direction for the designed software and allow later validation the use cases must be defined. They should cover the basic functions that can show the tool's potential without requiring full system implementation: this allows fast prototyping, incremental improvements and direction reshaping. Since the main focus of this work is to improve the work of scientists, almost all the use cases will cover data flows most commonly appearing in their workflows.

The first two testing examples were prepared just for the testing purposes of this work and represent the most commonly encountered data flows. They consist of multiple files and processes and by proper handling of them the designed tool should cover high amount of the available software. As they do not provide any parameters, they can only be used for validating data flow detection and not parameter matching.

The next two use cases are represented by the two real-world application suites. Turbomole [21] is used in quantum chemistry and represents lots of programs used in this field. The second suite is used in CTA project [10] and consists of multiple executables that are executed depending on the type of the input files. All those executables can alter their behaviour based on the number and the type of the input files and program arguments. Both of those suites can be used for the validation of flow detection, argument management and alternative data flow merging.

Although the main focus of the created software is aiding scientific environment last considered use cases should include the other types of work. Unfortunately, not much tools work on the same basis as the scientific software where data flow can be easily detected because the consecutive processes are connected by read and written files. In order to cover this use case without having to heavily modify heuristic engine of the prototype the one administrative workflow was chosen: the creation of virtual machine using *libvirt* [30].

3.3 Non-functional requirements

The requirements presented in chapter 4 cover the basic but essential functional requirements and general concept of the designed system. This alone can be used to implement the software which will fulfil the specified role although it does not guarantee good user experience. To create the software that not only addresses specified problem, but does it well, and improves users' experience in the targeted field, the additional technical requirements are needed.

The following list enumerates the gathered non-functional requirements and the functional additions that can be used to extend the designed system and improve its usability:

- Tracer directives: the special, control directives that can be used to control tracer behaviour to better suit user needs or to later help the parser with better understanding the data flow,
- Automatic paralleling: the feature that can be very effective in optimizing computer usage. It can be implemented by analysing the data flow and finding the commands which can be executed in parallel, based on the detection of common file that works as a "barrier"[4],
- Automatic checks and warnings: e.g. generated scripts can include features that check if the required files are present in the specified location before the execution,
- Documentation and examples: although the detection of the data flow is transparent, some modules require interaction from the user. Good documentation can help users learn the tool faster, prevent possible mistakes and inform about the advanced features,
- Open, standardized, text based API: allows creating multiple implementations in different languages and with different set of features,
- System modularization: allows the extensibility and exchange of specific module for the different implementation,
- Open source: allows collaboration, continuous improvements and faster error detection and fixes.

[4]Type of synchronization method.

Chapter 4

Concept of the action tracking system

This chapter provides the overview of the system architecture. Starting with the general concept, explaining the system as a whole and reasoning behind the division into specified modules. The chapter also includes the description and analysis of data flows that are the main concept on which the system can be built. Lastly, it includes the description of the environment and tools used in the proof-of-concept implementation.

4.1 General concept

To implement the functionalities covered in the chapters 3 and 4 the system architecture was split into separate modules depending on functionality. Those modules are connected using a simple text based APIs. This design allows interchanging the single module between different implementations in any language and from any developer who can implement required functions and protocol handling. The approach is based on Unix philosophy, proven successful over many years: "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."[1]. Although separated modules cover one simple functionality, they share data structures and basic functions, and for that reason the library with common code was created. The language of choice for this implementation (for most modules) is Python as it provides the programmer with great flexibility and syntax that helps maintain good practices while coding. More importantly, Python as the interpreted language allows fast prototyping and testing. Since the tools are not meant for computational operations, speed re-

[1]Quote by Doug McIlroy, the head of Bell Labs CSRC and inventor of the Unix pipe [15].

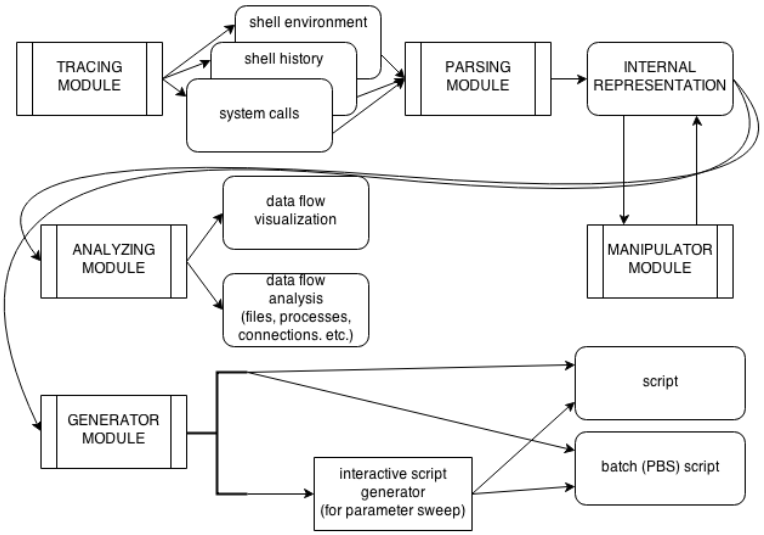

Figure 4.1. The division of the system into independent modules connected by open, textual interfaces. The modules represent the core functionalities of the created prototype and can be further improved to cover all possible use cases and data flows as described in the chapters 3 and 4 and extended as proposed in the chapter 7.

duction does not have negative implication. If needed, speeding the critical parts can be easily implemented in C language by using standard C-Python interface. Lastly, Python is a very popular language as it is installed on all major Linux distributions by default. This goes well with the requirement of portability, transparency and targeted low learning curve.

The separate modules, which create full working system, are presented in Fig. 4.1 and include: **the tracing program** used to record user-system interactions, **the parser** used to analyse data gathered by the tracing program, **the analyser** used to print the information gathered in the process of tracing (found processes, files, I/O operations), **the visualizer** used to visualize data that can be used to verify the detected data flows, and lastly, **the generator** used to generate scripts based on the detected data flows.

To present the usage of all modules one prepared use case will be used – TURBOMOLE. That example consists of one executable used to compute results (*ridft*) and arbitrary number of standard system commands like directory listing and file reading.

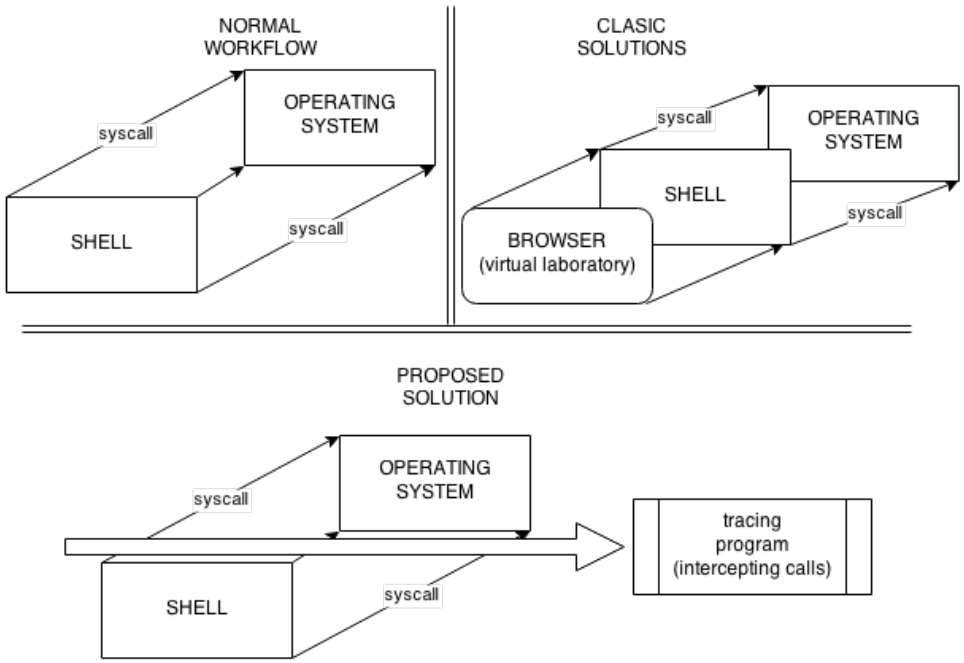

Figure 4.2. The idea of the transparent tracking of user-system interactions with comparison to the normal workflow and *virtual laboratories*.

4.2 Methodology of transparent user action tracing

The main idea behind the project described in this thesis was to elaborate the tool that can improve users daily routines by automatically analysing and repeating executed tasks without breaking their habits and requiring extra actions. The "proposed solution" in Fig. 4.2 presents the way of achieving this goal by creating a transparent layer between user (represented by shell or console program that user directly manipulates and executes commands in) and operating system, that will track (or intercept) all events arising in the process of interaction between user and computer.

Listing 4.1 presents the execution and output of three basic commands that display and manipulate files and directories:

- "pwd": print working directory;
- "ls": list directory/file information, directory content, with the "-a" argument that additionally shows "hidden" files beginning with dot (including "." (single dot) – current directory and ".." (double dot) – parent directory);
- "cd": change directory, with the argument ".." (parent directory).

```
1 % pwd
2 /home/test
3 % ls -a
4 ./  ../
5 % cd ..
6 %
```

Listing 4.1. Exemplary commands entered by the user in the terminal with their respective outputs after the execution. This example includes 3 file/directory manipulation and information commands: "pwd" – print working directory, "ls" – list (directories, files), and "cd" – change directory.

The last line represents the empty prompt line that indicates that shell is waiting for the new command to be typed by the user.

On Unix systems those "high" level commands are used to perform all actions on the system. They may be complex programs, scripts, or short functions. Although they differ in functionality, and their implementation vary greatly in source code length, they all use low level functions provided by the operating system. Those simple instructions are the system calls that are always used to perform actions involving processes and files.

```
1 pwd
2 ls -a
3 cd ..
```

Listing 4.2. The history of previously executed commands (as shown in Listing 4.1) as saved by the Bash shell.

The additional source of a valuable data is the history of commands executed by the user. Listing 4.2 presents the history of commands as saved by the shell from the executions of commands presented in Listing 4.1.

4.3 Finding relations between different processes

As stated in section 4.2, the main source of knowledge is the user-system interaction that manifest itself as a list of system calls with the addition of the history of executed programs. The knowledge obtained in this process may now be used to create the flow between consecutive commands. This connection can later be used to automatically create scripts.

At this point, it may look like the history of commands generated by the shell is just enough for the user to automate tasks by himself or herself as it only requires him or her to copy and paste commands to a file, add execution rights and run it. Unfortunately, the history of commands is, although very helpful, not enough on itself. The usage of the history alone requires user to not make mistakes, run commands only once, and focus only on a single

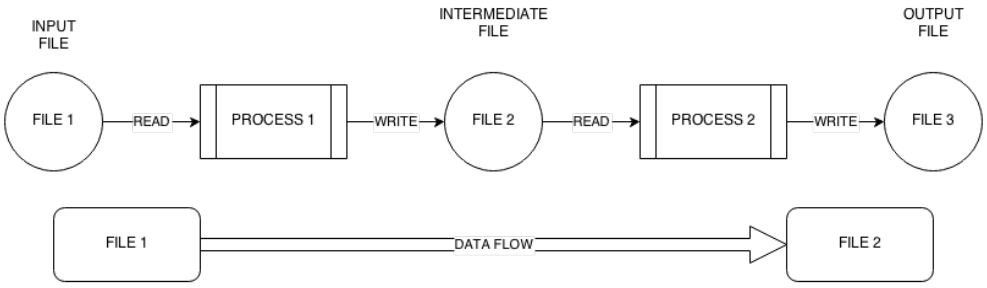

Figure 4.3. The relationship between two different processes and their relationship which manifests itself as the flow of data that is being written by the first process and read by the second process.

task as executing commands not connected to the main objective will also be repeated. Since the history is shared between the sessions and terminals it requires clean-up by hand to obtain only relevant commands. Lastly, the history can not be parametrized automatically, does not handle conditional execution and requires many manual actions to create "optimized" executable script that can be use multiple times.

The data flow is created by analysing the data that are being transferred between the processes in a form of files. Some processes write files while the others read the previously written text or binary data. By implementing the complex heuristics the program can intelligently connect commands and files (it is explained in chapters 4 and 5) to generate an internal representation which can later be used to generate executable scripts.

4.4 Types of data flows, detection and analysis

The main idea on which the concept of this work is based is the detection of the data flows. They are represented by reads and writes to files made by the processes executed by the user explicitly or implicitly inside those processes as presented in Fig. 4.3. The following subsections cover possible data flow shapes, methods of detection and analysis.

4.4.1 Linear flow shape, a chain

The simplest possible data flow includes only one process and one file which is created by this process. The possible example of this kind of flow is the use of *date* command and redirecting output to a file. Slightly more complex example is shown in Fig. 4.4 and contains five-element chain of three files (input, intermediate and output) and two processes manipulating those files. This kind of flow is very straightforward to analyse and recreate.

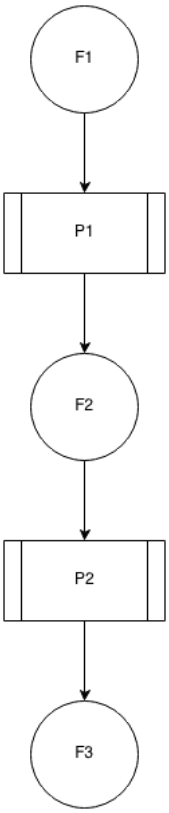

Figure 4.4. The simplest possible data flow: a chain. In this example the chain consists of one input file, one intermediate file and one output file, which are accessed by two processes.

4.4.2 Tree flow shape, parallelization

The slightly more complex data flow is presented in Fig. 4.5. The tree is a combination of multiple linear flows which are connected at some point in a program that requires multiple inputs. The branches which are just linear flows can be safely executed in parallel thus minimizing the total time needed for the whole flow to finish.

4.4.3 Smart flow detection: ignoring redundant data

The smart flow detection is a key feature of the designed system, it differentiates from the other solutions and greatly extends the idea of simple history from provided by the command line shells. Fig. 4.6 visualizes files and process gathered by the tracing program and founds relations and their lacks. The filtering of non meaningful commands can be done by excluding commands and files that are not connected to the final file or group of files. In Fig.

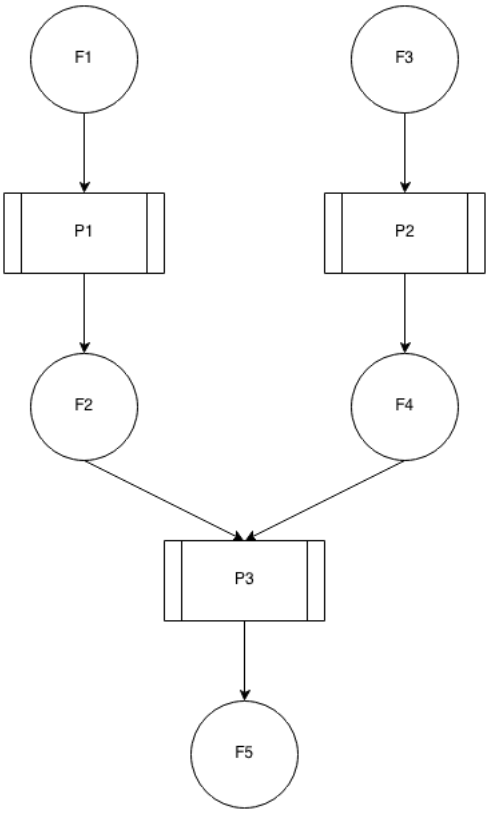

Figure 4.5. The slightly more complex data flow: a tree. In the picture the simplest version of this flow type with only two branches.

process one (P1) is not needed to create file 3 (F3) and can be safely ignored, and not included in the generated script.

4.4.4 Flow corruption prevention

The other important use case to consider is the situation in which the user is testing one of the programs by launching it multiple times. In this case the data flow will be visible as shown in Fig. 4.7. One program (but different process with its own PID) is executed multiple times and saves its output to the same file. By analysing the flags used to open this file (and intermediate operations between subsequent launches), the program can classify (with high, but not 100% certainty) if consecutive executions were needed for a normal flow creation or for testing purposes only. If the consecutive runs were not required, the file was probably opened with the overwrite mode, truncated, or removed before the following runs. Otherwise the file should be opened with append flag.

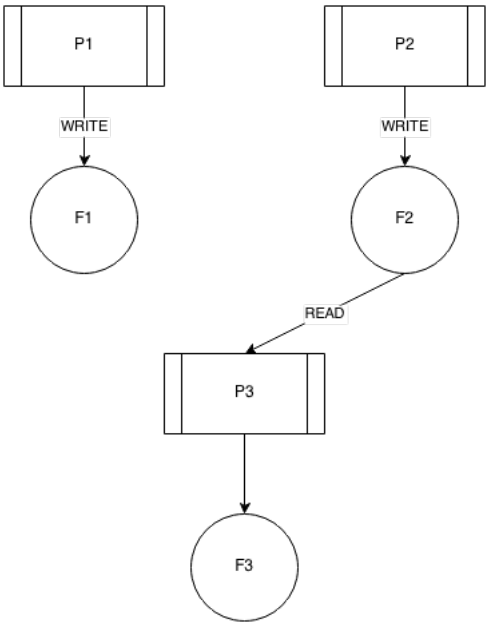

Figure 4.6. The relationships and their lacks can be used to automatically detect if consecutive commands are connected.

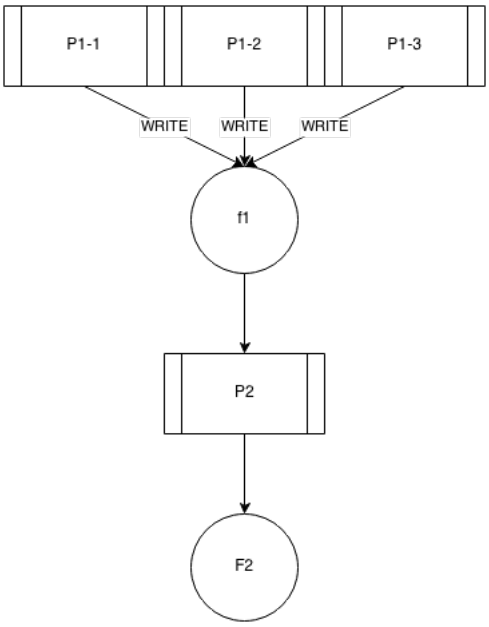

Figure 4.7. The detection of the flow corruption caused by launching one program multiple times can be avoided by analysis of the flags used for opening files.

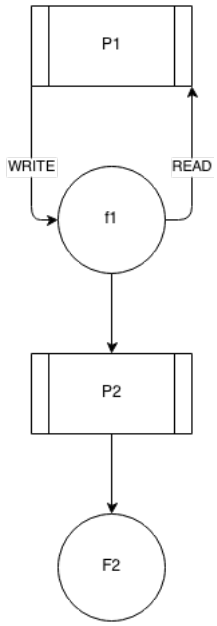

Figure 4.8. Opening file for read and write can cause searching for predecessors to fall into the infinite recursion. To avoid this, the tool should not visit the same node twice while traversing graph.

4.4.5 Flow recursion while opening files for read and write

In all previous flow examples, files were opened for reading or for writing, but not for both. The flow detection is based on going from written file to writing process starting on user defined file. If the process is reading and writing the same file, traversing graph will cause the recursion to be infinite and crash the program. This situation can be avoided by preventing searching function from entering the same node twice. Although the infinite recursion is prevented by the defined rule, it may still be useful to analyse this situation thoroughly. One of the possibilities is when the program is reading the whole file, truncating it and then writing new contents. If this was the case, this file should be treated as two different files. Another point to consider is that although the file was opened for read and write, this does not mean that any read and write operations actually happened and it should be checked explicitly.

4.4.6 Flow parallelism

As demonstrated in the subsection 4.4.2, the parallelization of the traced data flow can be done by running the tree branches at the same time up to commands that depends on both outputs. For the tool it does not matter if in time of tracking the tree branches were executed sequentially or in parallel

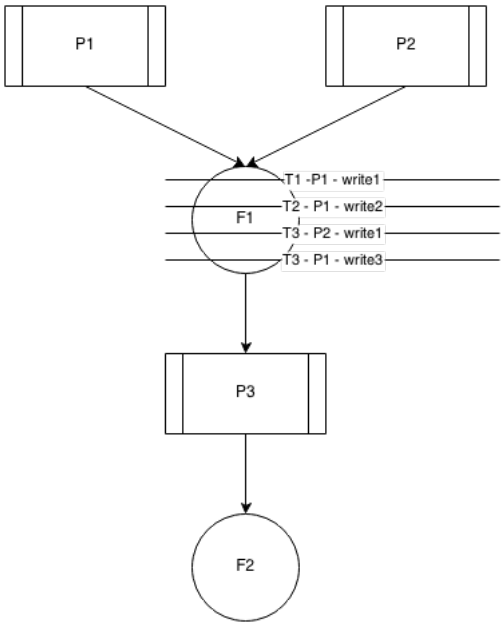

Figure 4.9. In terms of data flows, parallel execution does not make flow parallel. The parallelization of data flows happens only when two processes read and/or write to the same file at the same time.

as long as they join in one point in time. The only situation in which the data flow is actually parallel is when the two (or more) processes write and/or read the same file at the same time. Fortunately, since the system calls are always made in the sequential order the analysis tool can always properly detect data flow and command execution order.

4.5 Merging of alternative data flows

As presented in section 4.4 there are multiple possible data flows. Many programs provide different execution paths depending on the initial conditions like file contents and environmental variables. Some programs consist of multiple executables from which some are more commonly used than the others. This situation creates the possibility that one executable can create different data flows. The simple example is presented in Fig. 4.10 where FLOW1 presents the flow with two executables (first creates the intermediate files from the initial files) and FLOW2 consists of only one executable which is executed already on prepared intermediate files.

Since the user is offered transparent tracking, he or she should not be forced to always present all possible data flows from the beginning. Rather, the designed tool should ultimately allow automatic detection and merging of those alternative data flows. This information can be later used to choose

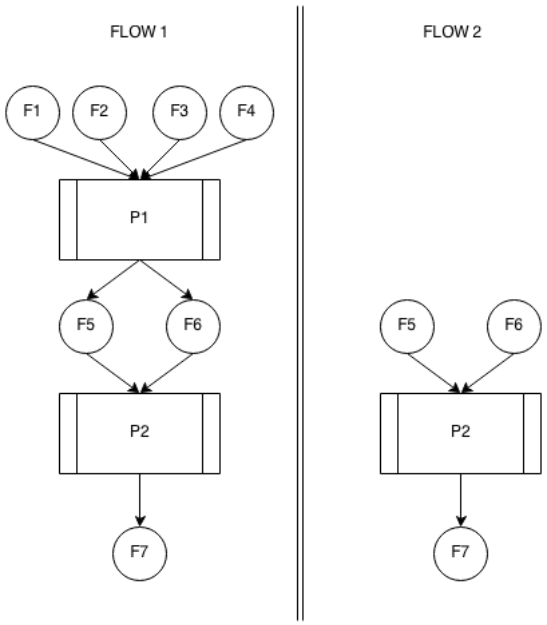

Figure 4.10. The example of two possible data flows for the same application package containing two programs, in which the type of flow is determined by the initial conditions. This can be used to demonstrate "alternative data flow merging" functionality.

a specific flow at the script generation step or even at runtime according to the detected environment and starting conditions.

4.6 Choice and analysis of external dependencies

In order to implement required functionalities swiftly and optimally, the proper analysis and revision of the available and commonly used tools was required. The following subsections provide a short overview of the environment that is being targeted by the designed tool, namely a shell and possible combinations of the commands that may be executed in this environment, and low level functions that are executed by the operating system to perform operations represented by high level commands/programs.

4.6.1 Command Line Interface (*bash*), the execution environment

The designed tool in its current version targets text environment as it covers most of the scientific and administrative tasks performed by the users. Shell, in command line interface is an interactive (or batch) environment, that can be used to execute arbitrary commands and present their results to the users.

Shells function as REPL[2], wait for an input on standard input (usually confirmed by the "enter key"), evaluate the command (call internal function, execute program), and print the results to the standard streams: output and error (usually a screen, but this can be redirected to file or socket). There are two main families of the standard command lines shells for Linux (Unix and Unix-like systems): *sh* (Bourne shell) and *csh* (C shell). According to the Debian[3] popularity poll[4], the *sh* family dominates the contents by high margin (sh-like: 160 000 vs csh-like: 2 500). Based on this data, creating software for *Bash* shell provides project with best possible combination of required development time and possible market requirements.

GNU Bash [18] is GNU Project's shell. It is the default shell in TOP 20 distributions on DistroWatch.com website[5]. This shell is fully compatible with *sh*, contains many additions and improvements over the original and incorporates many features Korn shell (ksh) and C shell (csh) brought to the users.

4.6.2 Possible combinations of commands

Bash supports five types of *commands* which are presented below in the order of priority[6]:

- Aliases: short functions that improve usability. The popular aliases include: "ll" for "ls -l" and ".." for "cd ..",
- Spacial builtins: builtins that are, for historical reasons, treated with special rules, not important in the terms of the created tool,
- Functions: the user defines functions, that offer more flexibility than aliases, usually used for encapsulating the most commonly executed sets of actions, for example: "function cdl cd $1; ls; " which allows user to execute "cdl PATH" that will enter directory and list all files afterwards,
- Builtins: functions built into the shell. Implemented for the purpose of the usability ("type" is a builtin that informs about the type of other commands), speed ("cd" – the external program would be slower), or to provide the essential functionalities of the shell itself ("bg": puts the job into the background),

[2]Read-Eval-Print-Loop.

[3]Debian project [37] – one of most popular Linux distributions.

[4]Debian's popularity poll [38] – voluntaries can opt-in to send statistics of installed packages which are later accessible on this page.

[5]distrowatch.com [31] – the website which catalogues the information about Linux distributions and packages.

[6]On the execution Bash searches the internal structures until it finds first that includes *command* specified by the name entered by the user

- External executables in *$PATH*: include all executables in the system as long as they are directly in the directory specified in $PATH environmental variable.

Knowing the type of command is essential for building a proper internal representation. The external executables are usually the same, for the specified software, on different computers, but aliases and functions are usually user specific. This may prevent the execution of the generated script on a different machine with a different environment. To solve this problem the created program must gather the environment data and inform the user about possible missing command beforehand so they can be manually (or half-automatically) added to the generated script. The analysis of the command type can be done by subtracting three sets of commands: all commands (gathered from the shell history), builtins (known beforehand from the specification), and executed processes (found in the tracing output). This subtraction will leave out "user definied" commands from the remaining groups: "aliases" and "functions".

The other possible classification of the executed commands is in terms of data flows, specified in the section 4.4:

- The commands that modify files: reading and writing from and to files, but also creation and removal, copying and moving, for example: "touch test.txt – creates file",
- The commands that modify file meta data: modifying file attributes, timestamps, etc.,
- The commands that modify the environment: setting, unsetting and exporting variables,
- The other commands: all not included in the previous points, for example: *bg*. *fg*, etc.

The most important commands are naturally those that modify file constants as they create the data flow. On the other hand, without the analysis of the other types the system may not be able to reproduce all required steps to accurately recreate the user executed flow and without the specific environment it may not work at all since it may not find required executables in $PATH. To avoid those problems, in the beginning, possibly the most important builtins in terms of this project should be analysed. Below list include those builtins and the potential problems:

- source, ".": used to execute commands from a file. Firstly, those commands are not visible in the history, secondly they are treated as sub commands of the source (although they are executed in-line as the separate commands), and lastly, the sourced file can not be replaced with

```
1  $ echo $MYVAR
2
3  $ export MYVAR=myvalue
4  $ echo $MYVAR
5  myvalue
6  $ true
7  $ echo $?
8  0
9  $ false
10 $ echo $?
11 1
```

Listing 4.3. Commands and their outputs executed in Bash shell.

the other arbitrary file, but only by one file that was specified as the argument,

- export, set, unset: used to manage the environment which may change $PATH (the order of directories in this variable determines lookup order), or other variables that determine and change the behavior of launched executables,
- exec: used to replace current shell with the specified program. It may include change of the interpreter (e.g. from Bash to Python) which will break the current model of analysis.

Bash allows "connecting" or "chaining" all presented types of commands in multiple ways. Those methods allow better representation of the natural flows and reduction of user time required for implementing them by introducing "syntactic sugar" optimized for shell specific operations. In order to present those operators, four standard commands will be used: *true* – which always succeeds (returns 0 exit status), *false* which always fails (returns 1 exit status), *export* which sets environmental variable and exports it to sub-shells, and *echo* which can print variable to the screen. Before presenting the possible operators and their roles, the one standard flow is presented in Listing 4.3 (note: variable "$?" represents exit code of the last executed command; printing unset variable returns the empty value displayed as the new line).

- ";" – allows entering multiple commands on one line:

```
1  $ echo $MYVAR; export MYVAR=myvalue; echo $MYVAR; true; echo $
       ?; false; echo $?;
2
3  myvalue
4  0
5  1
6  $
```

- "&&": the conditional (and) operator. Executes the next command(s) only if the first command succeed:

```
1 $ true && echo "success"
2 success
3 $ false && echo "success"
4 $
```

- "||": the conditional (or) operator. Executes the next command(s) only if the first failed:

```
1 $ true || echo "success"
2 $ false || echo "success"
3 success
```

- "()": the sub-shell operator. Launches new sub-shell for commands inside parenthesis:

```
1 $ echo $MYVAR; export MYVAR=myvalue; echo $MYVAR;
2
3 myvalue
4 $ echo $MYVAR; (export MYVAR=myvalue); echo $MYVAR;
5
6
7 $
```

The presented operators are built into the shell interpreter and are not visible in system calls. In order to properly analyse the flows containing those characters cross check between the shell history and the list of launched executables must be made.

Bash provides many additional operators and functionalities that are not commonly used or not important in terms of this project. The remaining list of characters that should be analysed by the designed system is presented below (note: cmd – the suitable command):

- "cmd &": the ampersand is used to put the job into the background. Can be utilized for simple multiprocessing,
- "<(cmd)": "process substitution" allows the process to appear as file for another process. It is a simple form of IPC[7],
- "CTLR+Z" keyboard shortcut: can be used to pause the job, which can then be put to the background using *bg* builtin,

[7]Inter Process Communication

- "#": the comment sign. Everything after the comment sign is ignored by the shell. Can be used for the implementation of the special control directives.

The presented lists included the most important information and possible traps that should be considered while implementing the tracing module. Those list are by no means complete and target only specific environment and the current version of GNU Bash.

4.6.3 System call tracing: *strace*

In order to gather the information about system calls, presented in section 4.6.4, the tracing program must be able to intercept the calls between user executed programs and the system. One possible solution to solve this problem is the usage of "strace" [26], the program which can be found in all standard Linux distributions like Debian, Ubuntu, Fedora, Red Hat and all their derivatives. Since it can be found in the standard repositories and is usually pre-installed its code base is mature and tested. It is also widely used by the administrators for debugging purposes. Because of that *strace* can be used as a tool of choice for the tracking system calls instead of programming this kind of software using *ptrace* system call directly or using a preload technique just for the purpose of this project.

Strace can track both system calls and signals sent and received by the tracked process. The process tracing can be done by specifying *command* as *strace* argument or by attaching it to already running program by passing PID[8] to it. While attached, *strace* intercepts all (or the specified list) system calls and signals and logs them to the screen or to the file. The child processes and threads are also tracked, as long as they are executed after attaching. The additional features of *strace* include: logging time of when system call was made and printing overall statistics of how much time each call type took 4.4.

There are three types of *strace* output lines (the sample output is presented in Listing 4.5):

- signals: "— SIGINT (Interrupt) —",
- system calls: "close(3) = 0". Dominant in the output, they consist of the three parts: the call name, the arguments and the return code and the error name in case of error,
- the unfinished/resumed system calls: "select(4, [3], NULL, NULL, NULL <unfinished ...>". The result of functionality for preserving the order of calls between different threads/processes in the process group.

[8] process identifier

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 44.38    0.000213         213         1           execve
 16.46    0.000079          10         8           mmap
 16.04    0.000077          39         2           open
  8.54    0.000041          21         2           fstat
  5.62    0.000027           7         4           mprotect
  3.54    0.000017           6         3         3 access
  2.71    0.000013          13         1           munmap
  0.83    0.000004           4         1           read
  0.83    0.000004           2         2           close
  0.62    0.000003           3         1           brk
  0.42    0.000002           2         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000480                    26         3 total
```

Listing 4.4. Output of *strace* command with '-c' switch, containing the statistics of how much time each system call took for "true" program.

```
execve("/bin/true", ["true"], [/* 60 vars */]) = 0
brk(0)                                  = 0x1411000
close(3)                                = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f92ce0ed700) = 0
mprotect(0x7f92cdeda000, 16384, PROT_READ) = 0
mprotect(0x604000, 4096, PROT_READ)     = 0
mprotect(0x7f92ce107000, 4096, PROT_READ) = 0
munmap(0x7f92ce0ef000, 89297)           = 0
exit_group(0)                           = ?
```

Listing 4.5. Sample, partial output of *strace* executing *true* command

As presented, the output of *strace* provides the administrators and developers with all needed information for the fast problem detection and analysis. Also developers can use it as a flexible tool for detecting application bottlenecks. Without this data it would be impossible to create data flow that is crucial for this project to operate.

4.6.4 Description and usage of selected system calls

System calls are the most basic functions provided by the system to all programs. The operating system provides those methods in order to control permissions and allows architecture independent way of performing tasks on processes, descriptors, files and more. Those calls include starting processes, opening and closing files, manipulating file descriptors, and reading and writing data to open descriptors. Although computational part of the

program codes does not utilize system calls and instructions like conditionals and loops are not represented by those functions, they provide significant amount of the information that can be used to debugging and analysing behaviour of the programs.

Presented below is the list of selected calls from Linux kernel which are recognized by the tool prototype [32]:

- fork [26] – *create a child process*: fundamental call in Unix systems[9]. Along with "exec" allows the creation of the running processes. This chain starts with PID 1 (init) which uses fork and exec to run all programs in the system. The newly created process – "child" – is the exact (except some points mentioned in manual, not important from the point of view of this work) copy of its "parent". Tracing this call allows creating of PID node in the data flow representation,

- clone [26] – *create a child process*: similar to fork, but with different implementation details,

- execve [26] – *execute program*: (usually) called after fork. This call replaces the current process with the new process created from specified file. Tracing this call allows adding the additional information like path to PID node,

- chdir, fchdir [26] – *change working directory*: used to change current working directory to the specified new one,

- open,openat [26] – *open and possibly create a file*: returns the file descriptor to the file specified by "path". Tracing this call allows creation of file nodes,

- close [26] – *close a file descriptor*: closes specified file descriptor,

- unlink, unlinkat [26] – *delete a name and possibly the file it refers to*: removes file (name) from the file system. Used by the programs like "rm",

- socket [26] – *create an endpoint for communication*: creates Unix "domain socket" and returns file descriptor,

- pipe,pipe2 [26] – *create pipe*: creates simple data channel that can be written on one end, and read on the second end. Can be used as IPC[10] between the parent and the child processes,

[9] In current version of Linux "fork()" function call of standard C library, executes "clone" system call underneath.

[10] Inter-process communication

- read [26] – *read from a file descriptor*: reads the specified number of bytes from file descriptor to buffer. Tracing this call allows the analysis of I/O characteristics,
- write [26] – *write to a file descriptor*: writes the specified amount of bytes from buffer to file descriptor,
- dup, dup2, dup3 [26] – *duplicate a file descriptor*: the family of functions allowing duplication of file descriptors. Tracing this system calls allows proper mapping between files (paths) and file descriptors,
- fcntl [26] – *manipulate file descriptor*: allows manipulation of open file descriptors. Those operations include: duplication, descriptor flag manipulation, file flag manipulation, locking and more.

Chapter 5

The prototype implementation

This chapter describes the implementation of all system modules that were presented in Fig. 4.1 and described in chapter 3, i.e. the tracing tool, the parsing tool, the visualizer, the analyser and finally the script generator. The section concerning the parsing tool includes the description of the internal representation that is used to exchange data between the modules. All sections contain basic information about this specific implementation, the most important features of every module and how they relate to each other.

5.1 The tracing tool

The first module – presented in Fig. 5.1 – in the chain of commands is responsible for gathering the data that were created in the process of interaction between the user and the operating system. In accordance with the points presented in the subsection 4.6.1, the targeted shell for this prototype is Bash. This specific tool is the only one not implemented in Python but rather in

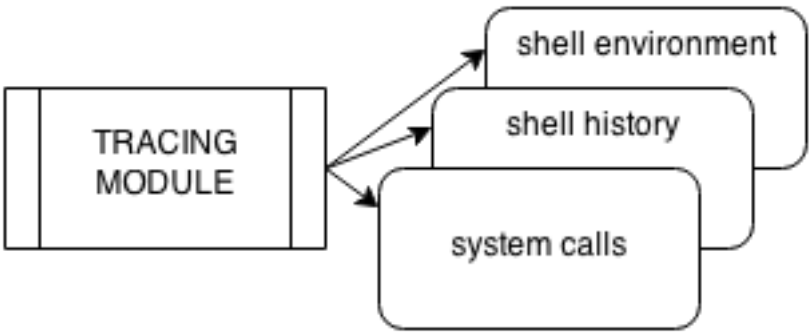

Figure 5.1. Part of architecture diagram: tracing module.

```
1  [plgmgolik@zeus ~]$ tracer.sh
2  Initializing tracer - it make take a while... You can start typing
       after prompt changes to 'monkey-trace: '.
3  The trace session is starting NOW! To leave, press CTRL+D or close
       terminal window.
4  monkey-trace: pwdd
5  bash: pwdd: command not found
6  monkey-trace: pwd
7  /people/plgmgolik
8  monkey-trace: ls
9  bin  trace  turbomole1  turbomole2
10 monkey-trace: cd turbomole1
11 monkey-trace: ls
12 auxbasis  basis  control  coord  energy  mos
13 monkey-trace: ridft > ridft.log 2>&1
14 monkey-trace: tail -1 ridft.log
15  ridft ended normally
16 monkey-trace: exit
17 The trace session has finished successfully!
18 Your TRACE directory was: /mnt/auto/people/plgmgolik/trace
19 [plgmgolik@zeus ~]$
```

Listing 5.1. Exemplary usage of tracing module.

Bash, which was done to minimize the number of processes that have to run between the standard shell and user invoked program.

The tracing tool was written in Bash using its specific functionalities and features [1]. For actual tracing of system calls *strace* program was used. This implementation allows the user who invoked the script to be presented with his or hers standard environment. To ensure the user is aware of the modified environment, he or she is presented with the welcome and closing messages and – more importantly – the modified prompt. Listing 5.1 presents the sample session conducted on the *Zeus* [11] supercomputer.

The first and the last line in Listing 5.1 present the standard prompt on the cluster and the invocation of the tracing program which was added to the user's $PATH variable. The next two lines contain the welcome message, while the two before last present the closing message. The other lines are executed in the traced environment and are the subject to system call, history and environment tracing. The commands executed in the traced environment include misspelled command, directory print, listing and change, and finally the invocation of computational command.

This functionality has been implemented using the following functionalities:

- Bash script: prints messages, prepares environment and launch the proper tracing environment,

```
1 BASH_RC=$(echo "
2 . ~/.bashrc\n
3 PS1=$TRACE_PROMPT\n
4 $CD_ALIAS\n
5 HISTFILE=$HISTORY_FILE\n
6 ")
```

Listing 5.2. Environment modifications used in tracing tool.

```
1 strace -f -o $STRACE_FILE -e "clone,execve,chdir,open,openat,close,
    unlink,socket,pipe,pipe2,read,write,dup,dup2,fcntl" bash --
    rcfile <(echo -e $BASH_RC) -i
```

Listing 5.3. Line which, after environment was set, launches the tracing environment.

- Process substitution: the functionality that allows processes to act like files for other processes. Used to load the environment for the sub shell without affecting the parent shell, presented in Listing 5.2. Currently it does the following things: sources user's default ".bashrc" [40], sets prompt, aliases "cd" command to "cd -P", which resolves symbolic links and sets Bash history file to the tracing directory,

- strace: used to record all system calls.

The final form of the tracing command can be seen in Listing 5.3. This command launches *strace* (with options: save output to file, detect individual processes, record only listed system calls) which will then execute Bash in the sub shell in the interactive mode while loading ".bashrc" file using process substitution from the echoed variable.

The data gathered by the tracing program can be broken into three main categories, which serve different purposes and can be used to effectively generate the output script for the user. The first category consists of meta data gathered on the environment initialization. This contains Bash environment snapshot, along with paths of tracing output directory, runtime directory and script directory. In the future the additional data can include host name and other system information, that can be used to improve portability. The second category includes history saved by Bash used to gather the information about launched commands and their arguments. The last category consists of system calls used to find executed processes, opened files and I/O operations.

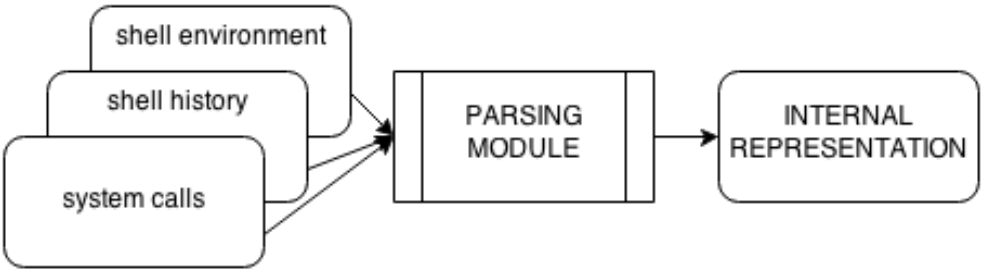

Figure 5.2. The part of architecture diagram: the parsing module.

```
lines_read      :       4284
totall_syscalls :       4179
% syscalls      :       97.54

handled_calls   :       4179
unhandled_calls :       0
% handled       :       100.0

==============================
2446        : write
567         : open
539         : read
466         : close
33          : fcntl
29          : dup2
28          : clone
18          : pipe
18          : execve
17          : socket
16          : unlink
1           : chdir
1           : dup
```

Listing 5.4. Exemplary usage of the parsing module.

5.2 The parser

The second module – presented in Fig. 5.1 – of the designed system is responsible for parsing, filtering and creating an internal representation. In Listing 5.4 the output of the parsing module is presented. Although the user executed only 7 commands the number of system calls exceeds 4100 invocations, while only tracing currently covered system calls. Tracing all system calls for this example exceeds 32000 lines. This amount of data has to be carefully filtered and organized in order to posses any level of significance.

The filtering is done by merging unfinished system calls with their continuations, and by ignoring insignificant system calls and ignoring repeated signals. The organization is made twofold: firstly by creating objects from the processes and files, and secondly by organizing the processes in hierarchy of parents and children.

The parser creates an internal representation which can later be used by other modules and, thanks to the text format, shared between computers and users. The processes contain PID, parent PID, the map of open file descriptors and files connected to them, current directory in the time of execution and program path and arguments. The file objects contain path, flags, mode that was used to open it, along with all I/O operations performed on them. The additional functionality includes optionally saving lines of *strace* that were used to create the process and file objects, or ones that interacted with them.

5.3 The internal representation format

Before continuing to the module that uses the internal representation to analyse, visualize and generate scripts, there are several important decisions to make about the format of the internal representation. Currently, the internal representation created by the parser has a format of the hierarchical list of processes with files attached to them and I/O operations attached to the files. This contradicts with the idea of generating scripts by following the data flows upstream (from the final file to the first input file required to create output), as this is easily done on the directed graph.

Unfortunately, keeping the data in the graph requires the usage of graph database (like currently the most popular Neo4j [33]) or development of own serialization method to hold this kind of data. Moreover the information about time, execution order, consecutive program executions and other additional data is lost, hard to store or mainpulate in graph databases.

In the later stages: visualization and script generation the directed graph proves high usefulness. The graph used to recreate the data flow and generate script is created only from the relevant data after filtering repeated processes, removed files, etc. In this structure files and processes are mapped to the nodes, while the mode (read/write) is used to direct the edge. This graph

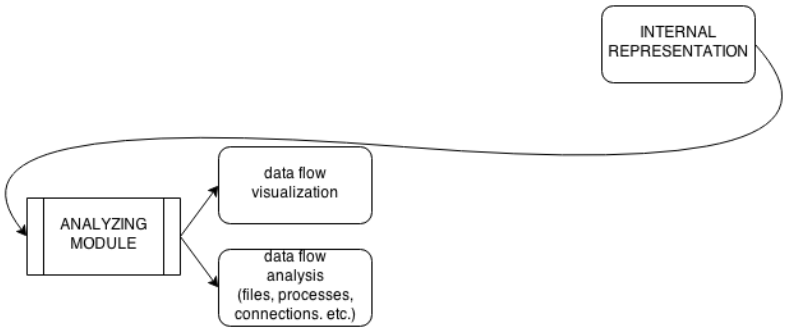

Figure 5.3. The part of architecture diagram: the analyser and the visualizer modules.

structure can ultimately be used to merge the alternative flows of programs because it consist only of the valid, clean and relevant objects. Assuming the validity of all the data the created graph represents the full program workflow.

5.4 The visualizer

The two modules presented in this and the next sections are used to analyse the gathered data and to visualize the detected data flow.

The architecture scheme of the visualizer tool is presented in Fig. 5.3. The command visible in Fig. 5.4 was gathered from the input presented in Listing 5.1. This graph structure can be used to validate the detected data flow or to help the users understand what interactions are made underneath by the programs executed by them.

5.5 The analyser

The more important tool for analysis is the textual analyser presented in Fig. 5.3, which presents more detailed information than the graphical one. The sample tool output is presented in Listing 5.5 and 5.6 and displays the data saved in the same internal representation but printed with different options.

Listing 5.5 provides the detailed information about all files opened by the process *32276* created from the execution of *ridft* program. The data visible on this listing include PID, full path of the executed file, number of bytes read and written to and from files, and number of the operations performed on those files. The data count is also provided for every file along with the specific flags and modes used for opening the file.

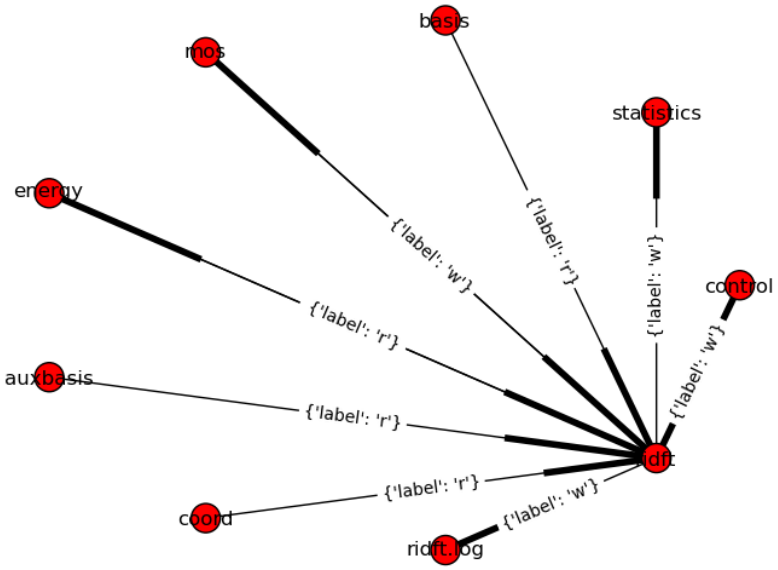

Figure 5.4. The graph created by the the visualization tool. It presents only one command that was filtered as relevant in the gathered data and all meaningful files connected to it.

Listing 5.6 in addition to the data visible on the previous listing, presents the individual I/O operations along with the amount of the data read and written printed in order of appearance.

The analyser program provides the user with multiple filtering options and printing settings. As shown in Listing 5.5 and 5.6 the individual I/O operations can be hidden to give the better overview of all files and processes. Moreover, the files can also be hidden in the situation when only processes are important. By default system files and processes are hidden, but can be shown using special switches. Most importantly, by default the main Bash process is hidden as it is not relevant to this analysis. In special cases it may also be required to show the child processes of user's executed programs, as by default only the processes explicitly executed by the user as shown and their children's files and I/O operations are flattened and displayed as their parents.

This tool can also be used by the administrators to debug programs and investigate their whereabouts. By analysing accessed files and sending signals, the administrator can discover undesired behaviour and validate per-

```
1 [32276] /software/local/Turbomole/6.5/TURBOMOLE/bin/em64t-unknown-
      linux-gnu/ridft [r: 260499 bytes (in 144 op), w: 65551 bytes (in
       1584 op)]
2         ridft.log [bytes: 15324; operations: 389; flags: O_WRONLY|
              O_CREAT|O_TRUNC]
3         /mnt/auto/people/plgmgolik/turbomole1/control [bytes:
              240781; operations: 845; flags: O_RDWR|O_CREAT]
4         /mnt/auto/people/plgmgolik/turbomole1/coord [bytes: 1434;
              operations: 2; flags: O_RDWR|O_CREAT]
5         /mnt/auto/people/plgmgolik/turbomole1/basis [bytes: 8632;
              operations: 8; flags: O_RDWR|O_CREAT]
6         /mnt/auto/people/plgmgolik/turbomole1/statistics [bytes:
              2025; operations: 45; flags: O_RDWR|O_CREAT]
7         /mnt/auto/people/plgmgolik/turbomole1/auxbasis [bytes: 8640;
               operations: 8; flags: O_RDWR|O_CREAT]
8         /mnt/auto/people/plgmgolik/turbomole1/energy [bytes: 1159;
              operations: 11; flags: O_RDWR|O_CREAT]
9         /mnt/auto/people/plgmgolik/turbomole1/mos [bytes: 48055;
              operations: 420; flags: O_RDWR|O_CREAT]
```

Listing 5.5. Exemplary usage of analyser module: displaying files opened by process.

missions.

5.6 The script generator

The script generator, presented in Fig. 5.5, is the last presented and also the final module of the whole system. It is used to generate the executable script from the detected data flow. In the most basic implementation the created script could only include the commands executed by the user with applied filters for de-duplication, removal of misspelled and non required for workflow commands. The current implementation includes two additional options: check for required files, and check for non zero exit code after every executed command.

In Listing 5.7, lines from 1 to 8 present the files that were detected as "written" with their corresponding number, which can be entered as the response to the question asked in line 9 and confirmed in line 10. The content between lines 11 and 16 can be copied and pasted to a text file, saved and launched after adding the permission for execution ("chmod u+x").

```
1  [32276] /software/local/Turbomole/6.5/TURBOMOLE/bin/em64t-unknown-
       linux-gnu/ridft [r: 260499 bytes (in 144 op), w: 65551 bytes (in
        1584 op)]
2          ridft.log [bytes: 15324; operations: 389; flags: O_WRONLY|
               O_CREAT|O_TRUNC]
3                   w -> 28 bytes
4                   w -> 38 bytes
5                   w -> 1 bytes
6                   w -> 76 bytes
7                   w -> 46 bytes
8                   w -> 1 bytes
9                   w -> 1 bytes
10                  w -> 29 bytes
11                  w -> 1 bytes
12                  w -> 1 bytes
13                  w -> 1 bytes
14                  w -> 44 bytes
15                  w -> 1 bytes
16                  w -> 59 bytes[32276] /software/local/Turbomole/6.5/
                        TURBOMOLE/bin/em64t-unknown-linux-gnu/ridft [r:
                        260499 bytes (in 144 op), w: 65551 bytes (in
                        1584 op)]
17         ridft.log [bytes: 15324; operations: 389; flags: O_WRONLY|
               O_CREAT|O_TRUNC]
18         /mnt/auto/people/plgmgolik/turbomole1/control [bytes:
               240781; operations: 845; flags: O_RDWR|O_CREAT]
19         /mnt/auto/people/plgmgolik/turbomole1/coord [bytes: 1434;
               operations: 2; flags: O_RDWR|O_CREAT]
20         /mnt/auto/people/plgmgolik/turbomole1/basis [bytes: 8632;
               operations: 8; flags: O_RDWR|O_CREAT]
21         /mnt/auto/people/plgmgolik/turbomole1/statistics [bytes:
               2025; operations: 45; flags: O_RDWR|O_CREAT]
22         /mnt/auto/people/plgmgolik/turbomole1/auxbasis [bytes: 8640;
                operations: 8; flags: O_RDWR|O_CREAT]
23         /mnt/auto/people/plgmgolik/turbomole1/energy [bytes: 1159;
               operations: 11; flags: O_RDWR|O_CREAT]
24         /mnt/auto/people/plgmgolik/turbomole1/mos [bytes: 48055;
               operations: 420; flags: O_RDWR|O_CREAT]
```

Listing 5.6. The exemplary usage of the analyser module: displaying I/O operations performed on one file by one process.

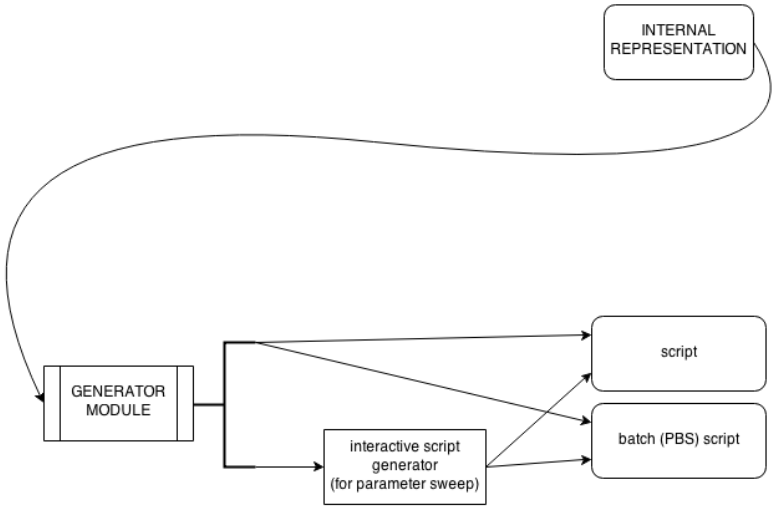

Figure 5.5. The part of architecture diagram: the script generating module.

```
[0] /mnt/auto/people/plgmgolik/turbomole1/ridft.log
[1] /mnt/auto/people/plgmgolik/turbomole1/control
[2] /mnt/auto/people/plgmgolik/turbomole1/coord
[3] /mnt/auto/people/plgmgolik/turbomole1/basis
[4] /mnt/auto/people/plgmgolik/turbomole1/statistics
[5] /mnt/auto/people/plgmgolik/turbomole1/auxbasis
[6] /mnt/auto/people/plgmgolik/turbomole1/energy
[7] /mnt/auto/people/plgmgolik/turbomole1/mos
Please provide number of targeted file [7]: 0
Generating backtrace from file [0] /mnt/auto/people/plgmgolik/
    turbomole1/ridft.log
--- SCRIPT START ---
#!/bin/bash
ridft > ridft.log 2>&1
if [[ $? -ne 0 ]]; then echo 'Command ridft exited with error'; exit
     2; fi
exit $?
--- SCRIPT END ---
```

Listing 5.7. Exemplary usage of parsing module.

Chapter 6

Validation and testing

This chapter describes the installation and usage of the developed tool (codename "monkey"). The next sections present the validation of the tool on the prepared use cases. Lastly, the list of encountered bugs and applied fixes is presented.

6.1 Installation and quick start guide

The system requires only one non-standard component used for the system call tracing: *strace*. It can be installed from the standard repositories (either by *sudo apt-get install strace* or *yum install strace*) or in case of lack of administrative privileges from the source [1]. The other requirements: Bash and Python are installed by default on all the major Linux distributions.

The easy access to the program can be achieved in two ways depending on user preferences: from the command line by adding it to $PATH variable or from the menu of the desktop environment, which supports "Desktop Entry Specification" [17], by manually entering required information to the menu editor or by copying prepared ".desktop" file (also known as launcher) to the specific location. The content of such file is presented in Listing 6.1.

Using this launcher allows user to start the tracing session directly from the graphical interface. After choosing its menu entry the tracing program will launch in the new terminal emulator window (specific for the desktop environment).

Launching parser, analyser, visualizer, and generator can only be made from the command line, as they currently only work in the text interface. All those commands are launched using single executable that will internally execute the specific tool, for example: "./monkey.sh parser".

[1] `http://sourceforge.net/projects/strace/`. – home page of *strace* program on which user may download its source code for the compilation without the administrator privileges.

```
1  [Desktop Entry]
2  Name=mnkey-tracer.sh
3  Comment=monkey tracer
4  Comment[pl]=malpka
5  Exec=~/bin/monkey.sh
6  Icon=Icon=utilities-terminal
7  Terminal=true
8  Type=Application
9  Categories=ConsoleOnly;Utility;
10 # vi: encoding=utf-8
```

Listing 6.1. ".desktop" file for the tracing tool.

6.2 Prototype validation on different use cases

The next three following subsections briefly describe the analysed problem, on which the tools were executed, along with the received outputs and their analysis.

6.2.1 Artificially prepared use cases

Tests on the use cases were considered and executed throughout the whole process of the software design process and implementation. Although simple, those use cases covered the most common usages of the popular programs. The situations which were not covered by those use cases still (after breaking them down and filtering) can be reduced to match those basic, tree shaped data flows. They test the most basic, yet crucial, functionality of the program: detecting data flow by following opened read and write operations.

Fig. 6.1 presents the graph (created in the external program) of processes and files (the nodes) connected by read and write operations (the directed edges). The data flow is built from 6 processes and 8 files and connected by 13 operations.

As expected, the output of the analyser program and visualization shown in Fig. 6.2 matches the real data flow shape perfectly as shown in Fig. 6.1. Additionally, all required data were recorded properly, later validated and the redundant commands were filtered by the parsing program. Those results fully validate the program operation, which allows progressing to the real world tests in the next sections.

6.2.2 Scientific use case: using TURBOMOLE application

One of the use cases prepared for the program involved the tool for quantum chemistry computations – TURBOMOLE. This program, in the process of development, was used only to test the limited number of features and

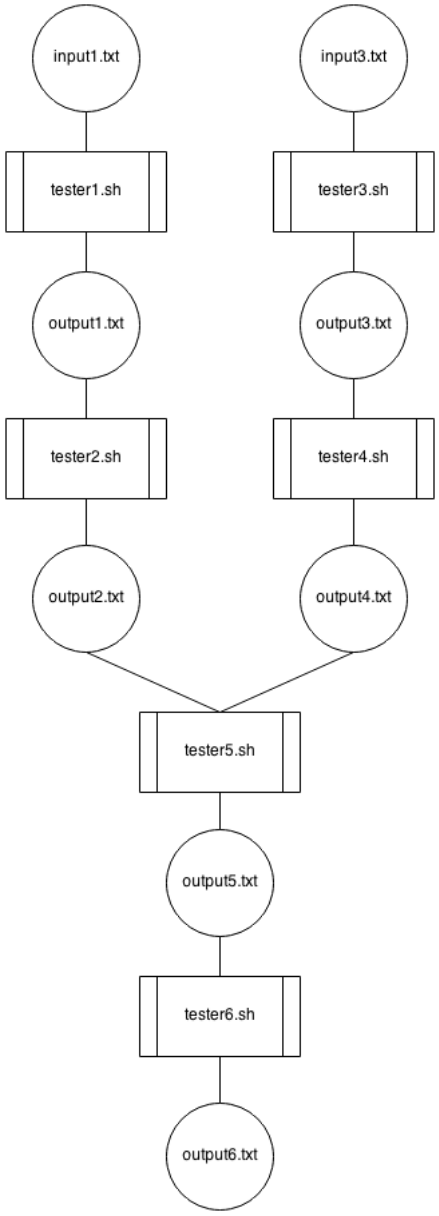

Figure 6.1. Use case that was used throughout the whole development process: covers the most basic, but the most important aspects of the designed system. This tree shaped structure includes 8 files and 6 processes that operate on those files.

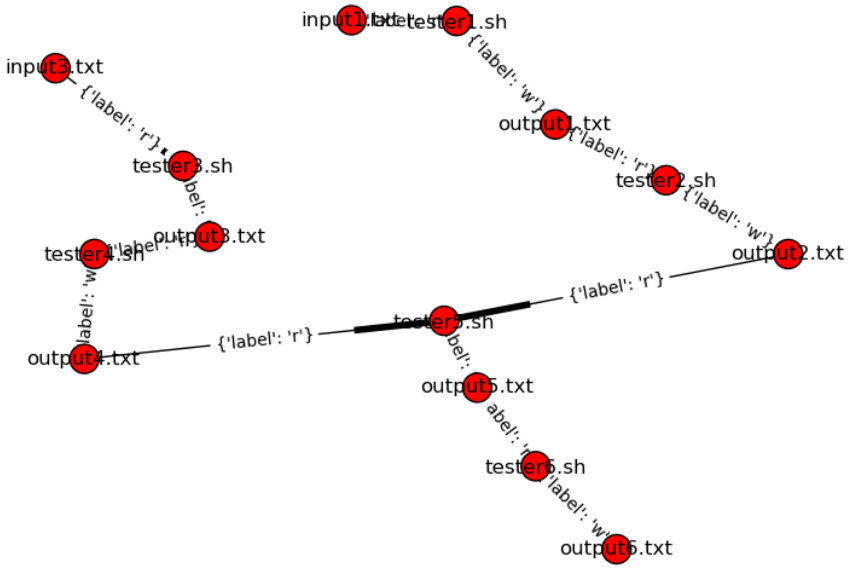

Figure 6.2. The data flow detected by the system perfectly matches the real data flow shown in Fig. 6.1. Presented structure was generated by the visualization tool. It includes processes and files as the nodes and the directed edges according to read or write operation.

```
1 x2t water.xyz >coord 2>preparation.txt
2 define < define.config >> preparation.txt 2>&1
3 ridft > ridft.log 2>&1
```

Listing 6.2. Exemplary usage of Turbomole suite – utilized in the system validation.

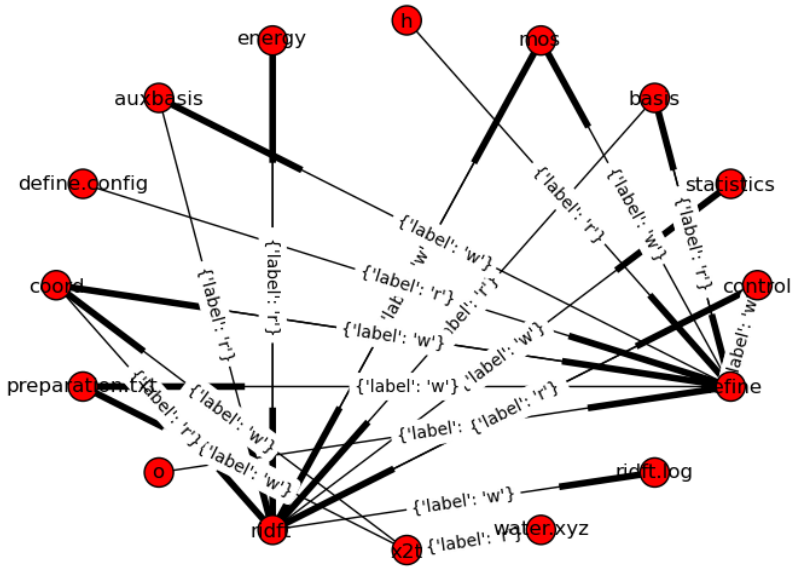

Figure 6.3. Graph generated after the tracing, parsing and visualizing gathered data for TURBOMOLE suite. The presented structure is complex and contains nodes connected by two-way directed edges which can lead to the infinite recursion.

never all parts of the system. The usage of this suite is presented in Listing 6.2. Those commands are specific in a certain way: they operate on the files with hard-coded names, the program searches for them in the current directory. Another functionality tested by this use case is the manipulation of standard streams: redirecting them to the files and to the different file descriptors ("2>&1": redirect standard error to standard output).

The detected data flow is presented in Fig. 6.3. The most important things that can be read from this graphical output are the names of files opened implicitly by the executed processes and fact that those files are opened for read and write. This opening mode leads to the unexpected behaviour which manifests itself by making consecutive processes depend on each other as can be seen in Listing 6.3. Although in line 10 user requests to generate script needed to recreate the file "coord" (which is modified only by the first two commands), the generator creates script that contains all used commands as visible in lines 20, 22 and 24. As stated before, this is caused by the file being opened in read and write mode by all executed commands.

```
1  [0] /mnt/auto/people/plgmgolik/turbomole2/coord
2  [1] /mnt/auto/people/plgmgolik/turbomole2/preparation.txt
3  [2] /mnt/auto/people/plgmgolik/turbomole2/control
4  [3] /mnt/auto/people/plgmgolik/turbomole2/basis
5  [4] /mnt/auto/people/plgmgolik/turbomole2/mos
6  [5] /mnt/auto/people/plgmgolik/turbomole2/auxbasis
7  [6] /mnt/auto/people/plgmgolik/turbomole2/ridft.log
8  [7] /mnt/auto/people/plgmgolik/turbomole2/statistics
9  [8] /mnt/auto/people/plgmgolik/turbomole2/energy
10 Please provide number of targeted file [8]: 0
11 Generating backtrace from file [0] /mnt/auto/people/plgmgolik/
       turbomole2/coord
12 --- SCRIPT START ---
13 #!/bin/bash
14 if [[ ! -f /software/local/Turbomole/6.5/TURBOMOLE/jbasen/o ]]; then
        echo 'File [/software/local/Turbomole/6.5/TURBOMOLE/jbasen/o]
       not found!'; exit 1; fi
15 if [[ ! -f water.xyz ]]; then echo 'File [water.xyz] not found!';
       exit 1; fi
16 if [[ ! -f /software/local/Turbomole/6.5/TURBOMOLE/jbasen/h ]]; then
        echo 'File [/software/local/Turbomole/6.5/TURBOMOLE/jbasen/h]
       not found!'; exit 1; fi
17 if [[ ! -f define.config ]]; then echo 'File [define.config] not
       found!'; exit 1; fi
18 if [[ ! -f /software/local/Turbomole/6.5/TURBOMOLE/basen/h ]]; then
       echo 'File [/software/local/Turbomole/6.5/TURBOMOLE/basen/h] not
        found!'; exit 1; fi
19 if [[ ! -f /software/local/Turbomole/6.5/TURBOMOLE/basen/o ]]; then
       echo 'File [/software/local/Turbomole/6.5/TURBOMOLE/basen/o] not
        found!'; exit 1; fi
20 x2t water.xyz >coord 2>preparation.txt
21 if [[ $? -ne 0 ]]; then echo 'Command x2t exited with error'; exit
       2; fi
22 define < define.config >> preparation.txt 2>&1
23 if [[ $? -ne 0 ]]; then echo 'Command define exited with error';
       exit 2; fi
24 ridft > ridft.log 2>&1
25 if [[ $? -ne 0 ]]; then echo 'Command ridft exited with error'; exit
        2; fi
26 exit $?
27 --- SCRIPT END ---
```

Listing 6.3. Output of the generator module executed on the data gathered while validating system on the selected use case: Turbomole.

```
1 monkey-trace: cp template.xml machine.xml
2 monkey-trace: cp template.qcow machine.qcow
3 monkey-trace: sed -i 's/<RAM>/1024/g' machine.xml
4 monkey-trace: ./start_vm.sh
```

Listing 6.4. Use case for the administrator workflow: virtual machine creation and launch.

Other than mentioned above, no other flaws were found while testing this use case.

6.2.3 Administrator's use case: virtual machine creation

The last use case was created to test the other then scientific types of workflows. Listing 6.4 shows the commands that can be performed by the system administrator to create a virtual machine using *libvirt*. The first two executed commands are *cp*: used to copy template files to the new location. The next command – *sed* – is used to replace a string that specifies amount of RAM for a virtual machine in XML file. The last command is a Bash script that executes "virsh define" and later "virsh start".

Fig. 6.4 presents the data flow created from the traced actions shown in Listing 6.4. The analysis of the visualizer output leads to finding out of the problems predicted in the design phase. Without the algorithm modifications, the script "./start_vm.sh" was not visible on the generated image at all because it was not writing any data. This happened because the processes that do not write any files are (by default) removed from the graph, because in terms of data flow it is not relevant. To fix this, the special switch that allows displaying those kind of processes was added and the process was shown as the new node in the graph. Unfortunately, the "./start_vm.sh" script is still stripped from the script generation component's output because there is no file that can be set as the final target of the data flow, and that is connected to this process with write operation.

As predicted, the current implementation prevents this software to be used in the environment in which processes are not connected by reading and writing common files, or there is no final output file.

6.3 Discovered problems and shortcomings

Although the tool validation was successful and no fundamental flaws were detected some minor bugs and lacks were found. Since they were not foreseen at the design stage but still before the final validation they are presented in this section and not in chapter 7. All found lacks and problems are presented below along with possible fixes:

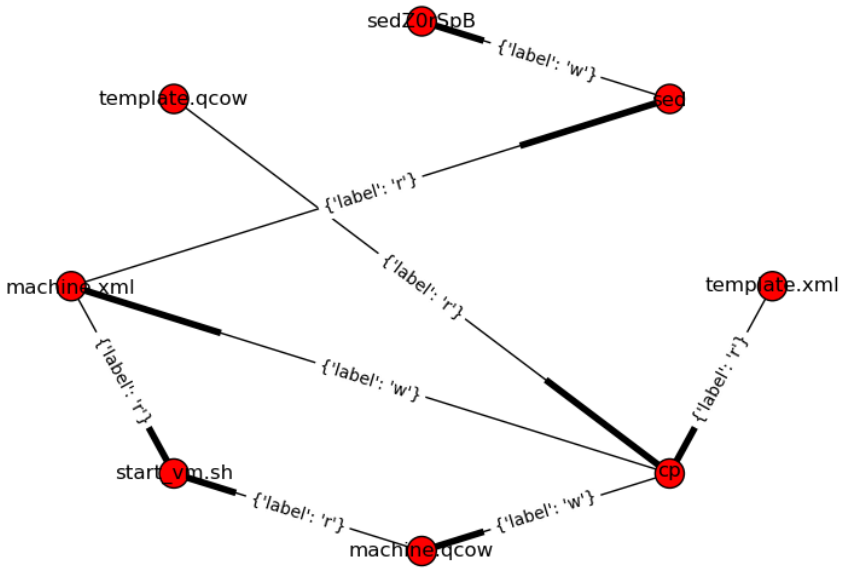

Figure 6.4. Data flow created from tracing actions performed by the administrator to create a virtual machine using *libvirt*. The "start_vm.sh" node does not open any file for writing. This prevents it from being a "target" node in script generation.

- Missing handling of the environmental variables: this problem requires the implementation of the initial environment read and watching for the modifications of variables for the specific processes,
- Different versions of *strace*: which occurs if the systems are running outdated versions of this utility. It can be circumvented by implementing more complex regular expressions, and by testing the versions that changed API,
- Different paths on different machines: may be bypassed by searching for the executables in $PATH, before launching commands and using the command name instead of the full path,
- Overwhelming analyser output: may be solved by adding more information and/or more filtering options. Another possibility is implementing interactive command line interface for example using *ncurses* library,

- Missing important system calls: more system calls can easily be added to the library after a short analysis of the calls format and functionality,
- Slowdown of the executed commands: this problem can not be easily fixed as it is caused by the process of the tracing itself. The parser tool (specifically underlying *strace*) has to intercept the system calls and save them to the file causing slowdown because of additional computing and I/O operations, However, the parser is only needed until an internal representation is (properly) created, so this point should not be interpreted as a problem but rather as a known limitation,
- Handling the other types of workflows: this can be done by fixing all the above listed problems (mainly environment handling), and by advancing heuristic module.

Chapter 7

Direction of the extensions

"The software is finished when it is obsolete" – this is the common saying in computer engineers community. This chapter focuses on extending the designed prototype and the vision of the whole tool suite. The first section covers modifications that are needed in order to implement the functionalities covered in the chapters 3 and 4. The next two sections cover new major functionalities that – if developed correctly – will greatly improve the usability of the system and may induce more users to use it. The last section covers some extra features that can later be added to cover even more possible use cases and user needs. Before attempting to extend the current implementation with new major features, firstly crucial functionalities missing from the prototype but mentioned in the chapters 3 and 4, and in section 6.3 need to be fully implemented.

7.1 Tracking and heuristic improvements

The first and the most obvious improvement is the implementation of the additional system calls namely calls include *stat* and *access*: for reading file information and tracking file permissions management. Another system call worth implementing is *wait*: it can be used to better understand the program behaviour and bottlenecks. The next group of calls consist of *geteuid*, *getuid*, *getegid*, and *getgid*: they are used to read the *effective* and *real* user data, and their addition will provide the possibility to detect errors caused by wrong permissions. The system calls like *brw*, *mmap* and similar are used to change program's data segment size, map files into the memory (to speed up processing), and perform other operations on the memory. The analysis of these functions will lead to better coverage of the possible data flows in the specific conditions. The last calls worth analysing include *rt_sigprocmask* and *rt_sigaction*, which are used for the signal handling and manipulation. After implementing this group of calls handling of "signals" should be im-

plemented, as they are commonly used to control the program behaviour.

The second direction of the development should focus on the parameter matching and analysis. This feature is crucial for the proper detection of the data flows as parameters often include file names and directory names. The analysis of the paths in the program parameters helps heuristic engine in making decisions about the types of files opened by the program. The files that are hard-coded in the program's sources should be treated in a different way than the file paths that are provided explicitly. Even other evaluation should be made for the files specified by shell's wildcards. The other important feature connected to the parameter analysis is the matching of the arguments between multiple commands. There is important problem to consider while doing parameter matching: sometimes the arguments with the same names (e.g. "–force" in one program forces some kind of operation while in other it is used to specify force in Newtons), but for different programs are not correlated with each other, while in other situations the options that perform certain action or set certain value (e.g. "-f" for one program and "–force" for other) for different programs are spelled differently, and can not be easily matched. This functionality is highly dependent on the program type and should be considered as highly unpredictable as the provided result quality can vary significantly.

The last group of the features to implement concerns the better analysis of Bash features, especially the management of environmental variables. Reading environmental variables is important as they are used for many purposes, with the most important of them: holding list of locations where binary files ($PATH) and shared libraries ($LD_LIBRARY_PATH) are stored. This includes the initial environment read and all further modifications including those in the subshells. The most important of the lesser Bash features includes the analysis of the conditional instructions (as they can alter data flow direction), loop instructions and "globbing" used by users to provide multiple file names to a single program as this differs (from the point of view of the program) significantly from providing all paths by hand. The extended analysis of Bash functionalities can be implemented in the parser and backed up on tracing program by utilizing *trap* on the "DEBUG" singal, functionality in Bash.

7.2 Graphical parameter matching, data flow manipulation and merging

Although, in accordance with the chapter 4 a transparent tracking is a key feature and advancement over the graphical tools it does not mean that graphical extensions can not be added to the tool suite to improve user experience. Such graphical tools can be added as the optional features in places where the graphical interface provides more advantages than disadvantages.

```
class ChdirSyscall(Syscall):

    regexp = re.compile('([0-9]+)\s+chdir\("([\S]+)"\)\s+=\s+(\-?[0-9]+)(\s+.*)?')

    @classmethod
    def parse(cls, line):
        matches = cls.regexp.search(line)

        pid = int(matches.group(1))
        path = matches.group(2)
        retcode = int(matches.group(3))

        syscall = {'pid': pid, 'path': path, 'retcode': retcode}
        return syscall
```

Figure 7.1. The screenshot from Eclipse IDE showing variable highlight. The similar technique can be used to implement the graphical parameter matching in the future software iterations.

There are currently two of the uses where it can make the work easier for users: the graphical parameter matching and the graphical alternative flow merging and management. Also by implementing the graphical tools more users can be attracted to the project which will lead to more sharing possibilities that are explained in the next section.

The graphical parameter matching is an extension of the third point in section 7.1. As stated there, the process of the parameter matching is very complicated and prone to algorithm errors, which can often easily be fixed by the user. The console is a primary environment for the created tool so it should be possible to manipulate connections between the parameters in the command line in as simplest way as possible. Unfortunately, operating on this kind of data, and displaying it in the console is hard to implement, and user may find it hard to correlate different parameters. This kind of problem is commonly solved in IDE[1] as shown in Fig. 7.1. The highlight for all instances of a single parameter can also be applied to distinguish program names, files, variables and constants. The visualisation of those elements, for the most people, will be much more effective for achieving good results in the parameter matching than the command line interface.

The second application of the graphical interface in the designed system, in which it improves user experience, is the graphical manipulation and merging of the alternatives of data flows. The interface for this feature could borrow ideas from the tools like Galaxy, described in the section 2.3.3, which uses the drag and drop web interface to create workflows. This idea does not contradict with the one of the main system concepts – the transparency – which is supposed to keep user in his or hers normal working environment, because it is the optional, additional feature. Moreover, it is supposed to be

[1] Integrated Development Environment.

only used to *improve* (as defined in the chapter 4, the system is created as the best-effort solution) detected data flow graph, but does not require user to create if from the scratch or developer to prepare any initial data. This approach is a good compromise between the current generation of tools and the concept presented in this work.

7.3 Platform for sharing

The last major feature goes in completely different direction, and is a natural response to the hypothesis in the section 2.1. One of the main points in that chapter was that the collaboration between users leads to more innovation and pushes business and science forward because common problems are solved quicker and wheel does not have to be reinvented in different places at the same time. That kind of approach is now very visible in software development and DevOps[2]. The two most bright examples of this trend are: GitHub [20] and Docker [13].

The first one is the biggest player in the market of the open source code hosting platforms. It allows users to use its technology for free as long as their projects are visible for other people (it also providing paid private repositories). The vast majority of the projects on this site are licensed using open source licenses like *MIT* or *GPL*. Developers use GitHub for storing *git* repositories, creating *wikis*, issue tracking and release planning. This global and open community collaborates on many projects by creating patches for the favourite software, reporting bugs or even just by learning on other people experience and creating a great new projects of their own.

The second example of sharing platforms is Docker, which is the extension of Linux containers – "operating system–level virtualization". It extends the idea of *LXC*, by providing wrappers that ease the usage of lower level tools. The most important feature of this platform is the community repository which can be used by anyone to create and share images of different systems and packs of software. Docker is now a fast growing project thanks to the vibrant community that provides new and experience users with the high amount of ready-to-use images. If the user does not find the combination of software needed, he or she can quickly create the new *container* and share it with the others.

Those examples show the superiority of the community based projects. The basics for the collaboration are built into this projects' core, as it is based on the open, text protocols. The social sharing platform that could be created for this project is only required to share the data prepared by already developed tools. The database provided by the proposed platform could hold the traces for different programs, parse objects, ready-to-use data-flows or even final scripts with parameter sweep.

[2]Development and Operations.

This idea can bring the great advantage in the scientific community: new users can quickly start with the new software without requiring to learn all commands and combinations. They can start with the generated script that will look for the dependencies or with the visualisation of data flow that will help understand how the software works. This platform could also be used by developers and administrators to prepare to show problematic use cases of software.

The design of this platform would require creating the command line tools and web site which will be used to browse all gathered data. Later on, a graphical client could also be added which would be integrated with the extensions mentioned in the previous section. Thanks to the open API and modular design the social platform could be designed by different teams or by community members.

7.4 Other possible improvements

The last section of this chapter includes some minor additional improvements to the tools which combined can lead to a better usability and usage efficiency:

- Support the other resource managers: while generating scripts it should be possible to generate directives for the different resource managers,
- More target languages for the script generator: now the only supported language is Bash, it would allow more flexibility if more languages could be implemented,
- System global monitoring: there are tools that can monitor the whole system (e.g. Monks [34] – Linux Procmon alternative) and not only on the terminal. This could potentially allow to record all day of activities instead of focusing on just one "special" window,
- GUI monitoring: currently only actions in the terminal are tracked, another worthy addition would be tracking the interactions with GUI (if possible),
- Suggesting the types of storage based on the frequency and amount of the data read and written to the files.

Chapter 8

Summary and Future Work

The work on the development of the tools presented in this thesis started as a research project intended to learn more about the nature of (low level) interactions between user and computer. Those interactions are reflected in system calls that are used underneath high level instructions and perform the most basic operations. Those operations, although basic, provide the great deal of information about the opened and closed files, read and written data and handled signals. Those pieces of information can be used to analyse program behaviour, find bugs and bottlenecks, but also to recreate the user actions performed on the higher levels. As the project evolved the new direction was set: using gathered data to transparently learn user's actions and recreate them in a *smart* way as closely as possible.

The list of the goals presented in section 1.2 along with validation is shown below:

1. *Do as much as possible automatically*: This point is unmeasurable because as the technology progresses it is always possible to do more. The current implementation along with the presented design and possible improvements will provide user with high amount of usability,

2. *Have a flat learning curve (or none at all)*: this requirement was met. The transparent tracking and simple commands in the text interface along with carefully considered user experience allow users to quickly start with this project and learn the advanced features when needed,

3. *Work in environment known to the potential user (no new level of abstraction, transparency)*: fully met. The tracer is using the standard Bash shell with only slightly modified prompt. Other tools act like the standard text tools in Unix-like systems,

4. *Have no or minimal dependencies (applications, tools, frameworks)*: met. The tools have only two dependencies: *Python* which is installed by default

in almost all Linux distributions and *strace*, which may not be installed by default in all distributions, but then it is available in standard repositories,

5. *Be portable and designed to use on different machines (PC, supercomputer, without root access)*: partially met. The created prototype does not yet consider all possible use cases needed for the portability between machines. On the other hand, on the level of protocols and software stack portability is ensured by open, text API, the open source implementation and minimal dependencies.

6. *Allow the exchange of scripts (and internal representations)*: fully met. The open text format is used to store system data. Additionally, the plans for the future social platform will extend this concept,

7. *Give the same benefits as similar tools that try to solve the same problem but with different approach*: the internal and simple user testing on the prototype allows to check this point as met in the current state. Unfortunately, the full evaluation of the efficiency can only be made after the implementation of all crucial features. More complicated tests along with the anonymous polls will fully verify the proposed idea.

The designed prototype implements only a subset of features required for the validation of hypothesis and does not cover all possible use cases and has deficiencies in user experience like handling of not allowed input. The main focus of the future work should be on the implementation of those remaining use cases and the important additional features like script parametrization, *PBS* directives and improvements to portability. Later on, to increase the usefulness of this tool, all the features presented in chapter 7 may be implemented as needed.

Although the tool suite was designed primarily as the tool of aiding (scientific) users work by automating workflows it has another potential use case. All the tools up to the visualizer and the analyser can be used by administrators (and possibly by users and developers) to analyse software behaviour, i.e. find bottlenecks, access files and executed programs, find bugs, backdoors and detect the unwanted access. Possible improvements in this direction include more detailed information and live-updating graph while the traced program is running.

Finally, only the deployment in the production environment will allow to verify the usefulness of the designed system. If idea will be proven successful for the users the system will allow better collaboration which will lead to more open science that benefits us all.

List of Figures

List of Listings

Bibliography

[1] *Advanced Bash-Scripting Guide*. URL: http://www.tldp.org/LDP/abs/abs-guide.pdf.

[2] ACC Cyfronet AGH. *GridSpace — DIstributed Computing Environments (DICE) Team Website*. URL: http://dice.cyfronet.pl/products/gridspace (visited on 07/31/2014).

[3] Mikolaj Baranowski. "Optimization of Application Execution in Virtual Laboratory". MA thesis. AGH University of Science and Technology, 2011.

[4] Giardine B, Riemer C, Hardison RC, Burhans R, Elnitski L, Shah P, Zhang Y, Blankenberg D, Albert I, Taylor J, Miller W, Kent WJ, and Nekrutenko A. "Galaxy: a platform for interactive large-scale genome analysis". In: *Genome Research* (2005).

[5] M. Bubak, B. Baliś, T. Bartyński, E. Ciepiela, W. Funika, T. Gubała, D. Harężlak, M. Kasztelnik, J. Kocot, M. Malawski, J. Meizner, P. Nowakowski, and K. Rycerz. "Experiments in GridSpace Virtual Laboratory - Principles and Examples". In: *KUKDM in Zakopane, Poland*. 2011.

[6] E. Ciepiela, P. Nowakowski, J. Kocot, D. Harezlak, T. Gubala, J. Meizner, M. Kasztelnik, T. Bartynski, M. Malawski, and M. Bubak. "Managing Entire Lifecycles of e-Science Applications in the GridSpace2 Virtual Laboratory - From Motivation through Idea to Operable Web-Accessible Environment Built on a Top of PL-Grid e-Infrastructure". In: *Building a National Distributed e-Infrastructure - PL-Grid - Scientific and Technical Achievements* (2012).

[7] computerweekly.com. *Write once, run anywhere?* URL: http://www.computerweekly.com/feature/Write-once-run-anywhere (visited on 07/31/2014).

[8] Adaptive Computing. *Fields: Job Arrays*. URL: http://docs.adaptivecomputing.com/suite/8-0/basic/help.htm#topics/moabWebServices/7-references/resources/jobArrays.htm (visited on 07/31/2014).

[9] Adaptive Computing. *Introduction*. URL: http://docs.adaptivecomputing.com/suite/8-0/basic/help.htm#topics/torque/0-intro/introduction.htm#Batch (visited on 07/31/2014).

[10] cta-observatory.org. *CTA Home*. URL: https://portal.cta-observatory.org/Pages/Home.aspx (visited on 07/31/2014).

[11] cyfronet.pl. *Zeus – Komputery Dużej Mocy w ACK CYFRONET AGH*. URL: https://kdm.cyfronet.pl/portal/Zeus (visited on 07/31/2014).

[12] Blankenberg D, Von Kuster G, Coraor N, Ananda G, Lazarus R, Mangan M, Nekrutenko A, and Taylor J. "Galaxy: a web-based genome analysis tool for experimentalists". In: *Current Protocols in Molecular Biology* (2010).

[13] Inc. Docker. *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/ (visited on 07/31/2014).

[14] A. Eilmes, M. Sterzel, T. Szepieniec, J. Kocot, K. Noga, and M. Golik. "Comprehensive Support for Chemistry: Computations in PL-Grid Infrastructure". In: *eScience on Distributed Computing Infrastructure, Achievements of PLGrid Plus* (2014).

[15] faqs.org. *Basics of the Unix Philosophy*. URL: http://www.faqs.org/docs/artu/ch01s06.html (visited on 07/31/2014).

[16] Python Software Foundation. *Overview — Python v2.7.8 documentation*. URL: https://docs.python.org/2/ (visited on 07/31/2014).

[17] freedesktop.org. *Desktop Entry Specification*. URL: http://standards.freedesktop.org/desktop-entry-spec/latest/ (visited on 07/31/2014).

[18] Inc Free Software Foundation. *Bash - GNU Project - Free Software Foundation*. URL: http://www.gnu.org/software/bash/ (visited on 07/31/2014).

[19] galaxyproject.org. *The Galaxy Project: Online bioinformatics analysis for everyone*. URL: galaxyproject.org/ (visited on 07/31/2014).

[20] Inc. GitHub. *GitHub · Build software better, together.* URL: https://github.com/ (visited on 07/31/2014).

[21] TURBOMOLE GmbH. *TURBOMOLE: Program Package for ab initio Electronic Structure Calculations*. URL: http://www.turbomole.com/ (visited on 07/31/2014).

[22] J Goecks, A Nekrutenko, J Taylor, and The Galaxy Team. "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences". In: *Genome Biology* (2010).

[23] Jeremy Kepner Hahn Kim Albert Reuther. *Writing Parallel Parameter Sweep Applications with pMatlab*. https://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pMatlabv2_param_sweep.pdf. MIT Lincoln Laboratory. 2011.

[24] D. Harezlak, M. Kasztelnik, E. Ciepiela, and M. Bubak. "Scripting Language Extensions Offered by the GridSpace Experiment Platform". In: *Building a National Distributed e-Infrastructure - PL-Grid - Scientific and Technical Achievements* (2012).

[25] InSilicoLab. *Summary | InSilicoLab*. URL: http://sciencegateways.org/what-is-a-science-gateway/ (visited on 07/31/2014).

[26] Michael Kerrisk. *Linux man pages online*. URL: http://man7.org/linux/man-pages/ (visited on 07/31/2014).

[27] J. Kocot, T. Szepieniec, M. Sterzel, D. Harezlak, M. Golik, T. Twarog, and P. Wojcik. "InSilicoLab: A Domain-specific Science Gateway". In: *CGW'12 Proceedings* (2012).

[28] J. Kocot, T. Szepieniec, P. Wójcik, M. Trzeciak, M. Golik, T. Grabarczyk, H. Siejkowski, and M. Sterzel. "A Framework for Domain-Specic Science Gateways". In: *eScience on Distributed Computing Infrastructure, Achievements of PLGrid Plus* (2014).

[29] J. Kocot, D T. Szepieniec, Harężlak, K. Noga, and M. Sterzel. "InSilicoLab – Managing Complexity of Chemistry Computations". In: *Building a National Distributed e-Infrastructure - PL-Grid - Scientific and Technical Achievements* (2012).

[30] libvirt.org. *libvirt: The virtualization API*. URL: http://libvirt.org/ (visited on 07/31/2014).

[31] Unsigned Integer Limited. *DistroWatch.com: Put the fun back into computing. Use Linux, BSD.* URL: http://distrowatch.com/ (visited on 07/31/2014).

[32] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001.

[33] Inc Neo Technology. *Neo4j - The World's Leading Graph Database*. URL: http://www.neo4j.org/ (visited on 07/31/2014).

[34] Alexander Nestorov. *alexandernst/monks · GitHub*. URL: https://github.com/alexandernst/monks (visited on 07/31/2014).

[35] ScienceGateways.org. *What Is a Science Gateway? | ScienceGateways.org*. URL: http://sciencegateways.org/what-is-a-science-gateway/ (visited on 07/31/2014).

[36] C.E. Shannon. *A Mathematical Theory of Communication*. Bell System Technical Journal, 1948.

[37] SPI et al. *Debian – Uniwersalny system operacyjny*. URL: https://www.debian.org/ (visited on 07/31/2014).

[38] SPI. *Popularity Contest Statistics – Debian Quality Assurance*. URL: https://qa.debian.org/popcon.php (visited on 07/31/2014).

[39] Harold Abelson; Gerald Jay Sussman; Julie Sussman. *Structure and Interpretation of Computer Programs (2 ed.)* MIT Press, 25 July 1996.

[40] tldp.org. *Sourcing a File*. URL: http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/x237.html (visited on 07/31/2014).

[41] Larry L. Yourdon Edward; Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press, 1979.