

CLI Crawler

- A tool for command line interface discovery



LUND UNIVERSITY
Campus Helsingborg

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:
Oskar Jermakowicz
Martin Ekberg

© Copyright Oskar Jermakowicz, Martin Ekberg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2015

Abstract

Many systems within the IT infrastructure have a Command Line Interface (CLI) for configuration changes. Some of these systems may expose a Configuration Management interface over a web service but this web service usually only exposes a fraction of the configuration possibilities in a CLI. Thus it would be of great help to investigate how a framework for automated CLI discovery can be developed, which is what this bachelor's thesis is about.

One objective of the bachelor's thesis was to determine the best possible way to access the command structure of CLIs and to determine how a CLI discovery application can be developed. The other objective was to develop such a prototype. Such a CLI discovery application must support exporting the result of the discovery process into a YANG model (a hierarchical modeling language for NETCONF) in the future.

A prototype, CLI Crawler, was developed. CLI Crawler was designed to be as automated as possible, however during the discovery process user interaction is required in order to help CLI Crawler get past certain obstacles. Such an obstacle could be when a CLI requires a certain input that only the user has knowledge of. At first CLI Crawler connects to a remote system with the use of Secure Shell (SSH) or Terminal Network (Telnet). Thereafter the discovery process is started which traverses all of the possible commands, modes and attributes in a certain CLI. During such a discovery process the command structure is both being printed in real-time in the GUI as a hierarchical tree structure and added to a database which will be used for exporting the command structure as YANG in the future.

CLI Crawler shows that it is possible to develop a framework for automated CLI discovery. However more work and research has to be done before CLI Crawler will become a viable way of discovering and representing a CLI's command structure. For instance more CLIs have to be integrated with CLI Crawler in order to make them compatible with the discovery process.

Keywords: Command Line Interface, discovery, automation, YANG, remote connection

Sammanfattning

Många system inom IT infrastrukturen har ett Command Line Interface (CLI) som används för konfigurationsinställningar. Vissa av dessa system kan använda sig av ett Configuration Management gränssnitt på en web-service, dock så brukar en sådan web-service endast ge tillgång till en viss del av konfigurationsmöjligheterna i ett CLI. Därmed hade det varit till stor hjälp att undersöka hur man kan utveckla ett ramverk för automatisk CLI discovery, vilket är examensarbetets syfte.

Ett mål med examensarbetet är att undersöka det bästa sättet att komma åt kommandostrukturen av ett CLI och undersöka hur en CLI discovery applikation kan utvecklas. Det andra målet är att utveckla en prototyp för CLI discovery. En sådan applikation måste stödja export av kommandostrukturen till en YANG modell (som är ett hierarkiskt modellerings-språk för NETCONF) i framtiden.

En prototyp, kallad CLI Crawler, har utvecklats. CLI Crawler har utvecklats att vara så automatisk som möjligt men användarinteraktion är nödvändig för att hjälpa CLI Crawler med vissa hinder. Ett möjligt hinder kan vara då ett CLI behöver en viss input som endast användaren känner till. Först så ansluter CLI Crawler till ett fjärrsystem genom Secure Shell (SSH) eller Terminal Network (Telnet). Därefter startas discovery-processen som traverserar genom alla möjliga kommando, moder och attribut i ett visst CLI. Under en discovery-process så lagras kommando strukturen i realtid i CLI Crawlers GUI som en trädrepresentation och i databasen som i framtiden kommer att användas för att exportera kommandostrukturen till YANG.

CLI Crawler bevisar att det finns möjlighet att utveckla ett ramverk för automatisk CLI discovery. Dock så krävs det mer arbete och undersökning innan CLI Crawler blir ett bra sätt att undersöka och lagra kommandostrukturen av ett CLI. Exempelvis så måste fler CLier integreras med CLI Crawler för att göra de kompatibla med discovery processen.

Nyckelord: Command Line Interface, discovery, automation, YANG, fjärranslutning

Foreword

This bachelor thesis is the result of the final course of the Bachelor of Science in Computer Engineering program at the Faculty of Engineering at Lund's University. The thesis was carried out at Data Ductus in Malmö.

We would like to thank Data Ductus for giving us the opportunity of doing our bachelor thesis with them. Sincere thanks to our supervisor Göran Edin at Data Ductus for providing a lot of valuable information and helping us when needed.

We would also like to thank our supervisor Christian Nyberg and our examiner Christin Lindholm at the Faculty of Engineering at Lund's University for aiding and supporting us with valuable input throughout the thesis work.

Oskar Jermakowicz & Martin Ekberg.

Table of contents

1. Introduction	1
1.1 Background	1
1.2 Purpose	1
1.3 Objectives	1
1.4 Questions at issue	2
1.5 Constraints	2
2. Method	3
2.1 Way of working	3
2.1.1 Pre-study.....	3
2.1.2 Implementation.....	4
2.1.3 Finalization.....	4
2.2 Source criticism	5
3. Technical background	7
3.1 Command Line Interface	7
3.2 SSH and Telnet	8
3.2.1 Ganymed SSH-2 for Java	9
3.2.2 Apache Commons (Telnet)	9
3.3 Hierarchical data model	10
3.3.1 YANG	10
3.3.2 Extensible Markup Language.....	11
3.3.3 Unified Modeling Language	11
3.4 Expect	12
3.4.1 Expect-for-Java	12
3.5 Neo4j	13
3.5.1 Web service	13
3.6 Graphical user interface	14
3.6.1 Java Swing.....	14
3.6.2 JavaFX.....	14
3.7 VirtualBox	15
3.8 Dynamips	15
4. Analysis	17
4.1 SSH and Telnet connection establishment	17
4.2 GUI toolkit	18
4.3 Database	18
4.4 Expect	19
5. Result	21
5.1 Overview	21
5.2 Functionality	22
5.2.1 Discovery algorithm.....	22
5.2.1.1 <i>Utilities</i>	23
5.2.1.2 <i>Tree representation</i>	24
5.2.1.3 <i>Resuming a CLI Crawler discovery</i>	25
5.2.2 Database.....	25
5.2.3 Communicating with the remote shell	27
5.2.4 GUI.....	28
5.2.4.1 <i>Setup</i>	29
5.2.4.2 <i>Output and debug</i>	30

5.2.4.3 Customization	31
5.2.4.4 Command structure.....	31
5.2.4.5 Managing the GUI during a discovery	31
6. Conclusion	33
6.1 Further development.....	34
7. Terminology	37
8. References	39
Appendices	43
Appendix A – Code snippets.....	43
Appendix A.1	43
Appendix A.2.....	44
Appendix A.3.....	44
Appendix A.4.....	45
Appendix B – Screenshots	47
Appendix B.1	47
Appendix B.2.....	48
Appendix C – Diagrams, charts and figures	49
Appendix C.1	49
Appendix C.2	50
Appendix C.3	51
Appendix C.4	52
Appendix C.5	53
Appendix C.6	53
Appendix C.7	54
Appendix D – User manual.....	55

1. Introduction

1.1 Background

Automation is important when managing large IT infrastructures since manual solutions are more time consuming and require more resources. The key challenge today is to interface with the systems within the IT infrastructure, which may for instance be done with options such as Simple Network Management Protocol (SNMP), Network Configuration Protocol (NETCONF), Simple Object Access Protocol (SOAP) or Representational State Transfer (REST).

Many of the systems within the IT infrastructure have SNMP to expose events and a Command Line Interface (CLI) for configuration changes. Even if a system exposes a Configuration Management (CM) interface over a web service, the web service only exposes a fraction of the configuration possibilities of the CLI. This is especially true for older systems. Thus it would be of great help to develop an automated framework for discovering the structures of CLIs since it has not been previously done and there is no information or documentation on how such a framework could be developed.

The thesis work was done at Data Ductus in Malmö which is an IT consulting company that specializes in advanced solutions within telecommunications and IT. These advanced solutions are for instance Network Management Solutions, Software Defined Networks, system development, integration, infrastructure & visualization, Machine-to-Machine and mobile & web development. There are currently around 150 employees in all of the Data Ductus offices which are located in Malmö, Skellefteå, Uppsala, Stockholm, Luleå in Sweden and in Longmont in the US.

1.2 Purpose

The purpose of the thesis is to investigate a method for discovery of certain CLI based systems and decide whether it is possible or not to develop a framework for automated CLI discovery.

1.3 Objectives

In order to fulfill the purpose of the thesis there are two objectives that the thesis work is focused on achieving.

The first objective is to analyze several ways to approach a CLI's command structure and determine how a CLI discovery application may be developed.

The second objective is to develop a prototype in Java, CLI Crawler, which connects to a remote system through SSH or Telnet to discover the systems CLI command structure. This prototype must be executable and maintainable on a Linux Platform. The discovery progress should be stored in some way so that the process can be resumed on a later occasion and in the future support that the result of the discovery is exported into a YANG model, a data modeling language for NETCONF.

Such a YANG model would include the following:

1. Classes.

2. Associations and dependencies between classes.
3. Attributes within a class.

The result of the objectives should be as follows:

1. A pre-study showing a high-level design on one selected CLI.
2. A high level design on a CLI discovery application. The application shall navigate through a CLI by using SSH or Telnet to connect to a system.
3. A prototype implementing a subset identified by the pre-study.

1.4 Questions at issue

In order to complete the objectives it is important to analyze several factors that are important to consider during the thesis work. These factors are shown below as questions at issue, answering them during the thesis work is important in order to achieve the best possible result of the objectives.

1. Which CLI would be the most relevant to investigate and show as a high-level design in the pre-study?
2. How can one access a CLI's command structure?
3. How can the CLI's command structure be stored before being exported to YANG?
4. How can the application be developed in order to be as automated as possible?
5. Are there any open-source software which may be used as help for the application?

1.5 Constraints

The thesis work was limited in some aspects, which may have limited the scope of the result. Below is a list of several constraints that may have limited the thesis work in some way.

1. The application cannot require any license for development and execution.
2. Due to time constraints it is only feasible to investigate a limited amount of CLIs.
3. There is no information, documentation or any ideas on how such a discovery algorithm could work.
4. It may not be possible to make the application fully automatic, which would mean that some functionalities would need user interaction.
5. The application will not be compatible with all CLIs.
6. Due to the scope of the thesis work, exporting the CLI's command structure as YANG will not be implemented. However CLI Crawler will be designed to make it possible to implement YANG support in the future.

2. Method

2.1 Way of working

During the thesis work Scrum (Kniberg & Skarin, 2010) was used as the project model. Scrum provides the ability to easily adapt during a specific project which leads to more flexibility during the thesis work. Other factors that make Scrum suitable for the thesis work are for instance that most of the thesis work is done at Data Ductus with a near connection to the customer and that the thesis work would be iterative as seen in Figure 2.1. Both the thesis workers and Data Ductus also have previous experiences with Scrum so the choice of project model fits very well with the thesis work.

The thesis work is divided into two week sprints where each sprint starts with a planning session and ends with a demo and retrospect. Each sprint has daily morning meetings. The initial project plan can be seen in Appendix C.7.



Figure 2.1 shows a visualization of the way of working during the thesis work.

As seen in Figure 2.1, the thesis work is divided into two main phases which both consist of three sprints each, the pre-study phase and the implementation phase. After the implementation phase comes a short finalization phase (consisting of no iterations) which focuses on finishing the last bits of the thesis work as an entity. The thesis report was being worked on during the whole span of the thesis work, but the majority was written during the second half of the thesis work and in particular during the finalization phase.

2.1.1 Pre-study

The main purpose of the pre-study is to determine the most efficient ways of implementing CLI Crawler and becoming familiar with the tools to be used. Learning from various tutorials available on the internet and gathering knowledge from personnel at Data Ductus was of great help. A specific network device simulator was chosen to be focused on since one of the thesis objectives is to provide a high-level design of a certain CLI during the pre-study. The network device simulator was chosen since it was well documented, structured and that Data Ductus was familiar with it.

There were two important factors during the pre-study. One was to investigate available open-source software that could be of use for CLI Crawler and the other was to construct the foundation of the discovery algorithm. Functions such as SSH and Telnet establishment, stream handling among others are critical for CLI Crawler and since these functions may

have already been developed, investigating available open-source APIs was important. To know which of these APIs would work best with CLI Crawler required basic understanding of the algorithm, how it will work and what it will need to function properly. Thus it was best to investigate various APIs in parallel to constructing the foundation of the algorithm. The simulated CLI was the main focus when constructing the foundation of the algorithm, as it was viable to first focus on one particular CLI and understand it before moving on to other CLIs.

During the pre-study a lot of the foundation for CLI Crawler was developed, which was only to be expected after focusing on APIs and the logic of the algorithm. Even though most of the pre-study was developing, the phase still included a lot of research and gathering of knowledge.

2.1.2 Implementation

After the pre-study, the foundation of CLI Crawler was finished and now had to be further developed. This phase consisted of almost only programming.

In general, the way of working consisted of four steps;

1. Prioritizing which feature to develop.
2. Writing the logic in pseudo-code.
3. Translating the pseudo-code to Java.
4. Testing and fixing potential issues.

At first, it had to be determined which feature is currently important to develop, this was done during discussions both internally between the thesis workers but also with Data Ductus. After a choice was made, the best way to implement the new feature had to be determined which was also done with discussions between the thesis workers. Usually pseudo-code was written since it was easily editable and highly readable, thereafter it was translated into Java code. Once translated it was tested and sometimes altered in order to make sure it fulfilled the desired functionality.

Even though most of the APIs that were needed were identified during the pre-study phase, it was noted that additional APIs may be needed. An example of this is JavaFX which was used for the GUI. At some stages it was also noted that some changes had to be done in previously developed features in order to make new features work properly. Thus the workflow was not only based on new functionalities but also on the features and APIs that were established during the previous phase.

2.1.3 Finalization

The final phase, which had no strict sprints or iterations was used to finish everything that was left to do which includes both continuing to develop CLI Crawler and to finish the thesis report. The finalization of the report was done during this phase, the foundation of it and the majority of the sections had already been written but putting everything together and writing the final sections was still to be done.

In parallel to the finalization of the report, the planning of the presentations was also being worked on (one presentation for the university and one for Data Ductus). While writing the report one had an idea of what would be relevant to include in the respective presentations, which naturally led to planning the presentations.

2.2 Source criticism

Most of the references (which can be found in section 8) used for this thesis report are internet sources. Technology is constantly under evolution and most technology related topics are rapidly evolving. With such a fast change in various technologies, the most relevant and up-to-date information is often found on the internet even though older articles and literature can cover common knowledge within a certain topic.

Before choosing a certain source to reference in the report the source was analyzed in order to determine whether it was trustworthy or not. Different types of sources were analyzed with different factors in mind, generally the analysis was based on the following factors:

1. Who is the publisher? Who is the author?
2. When was the source published? Is there newer information about the specific topic?
3. Are there any hidden motives behind the source?
4. Does the source originate from any other sources?
5. Are there any other sources that have the same type of information?

These factors are inspired by and based on the guide *Källkritik på Internet* (Alexanderson, 2012).

CLI Crawler uses multiple APIs which are referenced to throughout this report. Before choosing each API the two following factors were considered in order to determine the legitimacy of the source:

1. Is the API published by the developer(s) or an affiliated part with the developer(s)?
2. Do other sources such as forums, blogs etc. recommend the API?

There are usually multiple APIs to choose from that all have similar functionality, so the main factor to consider is whether the API is recommended by other sources and whether the API is published by the developer(s). The references of the report refer to the primary source of each API which is as reliable as a source can get.

Another type of source used in this report are RFCs. RFCs are published by the Internet Engineering Task Force (IETF) that has a world-wide reputation. RFCs are however not always primary sources since the RFCs can be written by parts that are not affiliated with the IETF, but since they are published by the IETF they can be seen as legitimate.

Other sources consist of web-pages or articles that are published by different companies or organizations that are seen as trusted within the IT industry. These companies and organizations have good reputation and many other sources recommend them. These web-pages and articles are also primary sources and after considering the factors in the beginning of this section the sources were seen as legitimate and trusted.

3. Technical background

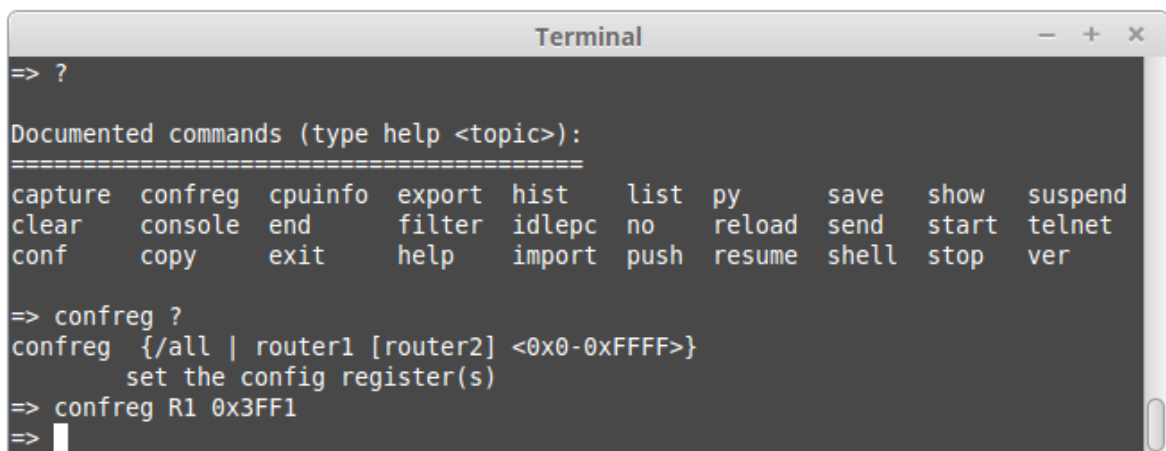
This section contains several sub-sections where each explains a relevant technical aspect of the thesis work. This is to provide knowledge about relevant information that will be needed in order to understand the result of the thesis work.

3.1 Command Line Interface

A command line interface (CLI) is an interface which is used for text-based communication between a user and a system or software. The communication is based on typing commands into a text-based console and then receiving the feedback or output back the same way. Simply put, it works as a chat between a user and a system or software.

Different CLIs have different purposes. An example may be a UNIX shell which is a command-line interpreter which is used on most Linux platforms, one may operate the system with the traditional graphical user interface (GUI) however the UNIX shell provides the user with the ability to do it with commands instead. The command prompt on Windows systems provides the same ability and works in a similar way.

CLI was the main way for interaction with a computers system and software in the early years of computing. However, it is still very relevant in the IT industry as it provides a lot of flexibility for developers. Developers can make automated software that navigates and performs operations by sending commands to a CLI and parsing the output. An example of a CLI can be seen in Figure 3.1.



```
Terminal
=> ?

Documented commands (type help <topic>):
=====
capture  confreg  cpuinfo  export  hist    list  py      save  show  suspend
clear    console  end      filter  idlepc  no    reload  send  start telnet
conf     copy     exit     help    import  push  resume shell  stop  ver

=> confreg ?
confreg {/all | router1 [router2] <0x0-0xFFFF>}
       set the config register(s)
=> confreg R1 0x3FF1
=>
```

Figure 3.1 shows an example of a Cisco router 7200 CLI emulated in Dynamips.

Different types of network devices such as routers and switches can have their own CLI which is used for configuration changes of the device. A CLI is usually divided into a hierarchical structure which consists of different modes which contain their own set of commands. Commands are used for various changes but some commands are dedicated purely to entering other modes with other commands, for instance “config” is often used to enter the main configuration mode of a network related CLI.

Commands can sometimes be used as they are initially represented but many commands have different attributes which may be appended to the commands. The purpose of attributes is usually to configure a certain setting of the command that is being sent. The

attributes are often required in a specific order to send a command and in some cases they may be optional. This varies between different commands based on their purpose. Some commands are strict with the way the attributes can be appended but some commands may take attributes in any possible order and any possible amount of them, these are often referred to as lists.

The CLIs of different devices may vary, usually on the following factors:

1. The command structure.
2. How the output is presented.
3. The prompt.
4. How to navigate.

The main difference is of course the command structures, different CLIs have different commands and therefore different configuration changes can be made. However, different CLIs also have their own way of doing things – how they present the output, how one navigates through the CLI and how the prompt looks. The three last factors are in fact the most important to CLI Crawler, as they determine how the full command structure can be accessed in a certain CLI. How these factors impact CLI Crawler is further explained in section 5.2.1.

As for the CLI in Figure 3.1, one can display all the commands in a certain mode by typing a question-mark. To see the possible attributes of a command, one can type the command followed by a question-mark. This way of working is very specific for the CLI in Figure 3.1 and even though other CLIs are often based on the same principles they can still have variations in some aspects.

3.2 SSH and Telnet

Secure Shell (SSH) and Terminal Network (Telnet) are two network protocols defined in respectively RFC 4253 (Ylonen & Lonvick, 2006) and RFC 854 (Postel & Reynolds, 1983) by the IETF, which with the use of TCP as the underlying transport protocol provide remote access to systems such as Linux and UNIX-like servers as shown in Figure 3.2.

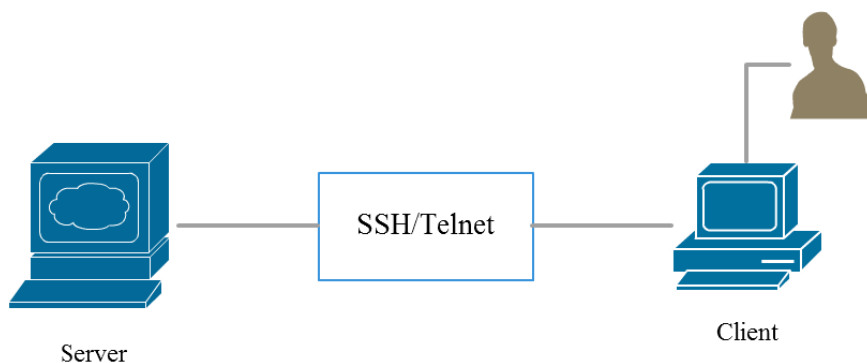


Figure 3.2 shows a visualization of a SSH or Telnet connection.

The remote access establishment is essentially used to execute shell commands in a remote system where the local terminal appears to be the terminal in the remote system. While SSH establishes a secure channel between the end points (for example with RSA and DSA encryption), Telnet has no such functionality for security which is the main difference

between the two. The disadvantage with SSH could however be the fact that encryption and decryption increases the computing time.

To establish a connection between the client and the host it is required that the correct username, password, IP address and port number is provided for the remote system. The standard TCP port 22 is generally assigned for SSH servers and TCP port 23 for Telnet servers (Study CCNA, n.d.). Even though a password is needed to establish the connection, Telnet is not secure enough as it sends and receives the password (and the same goes for other types of data) in clear text while SSH sends and receives it encrypted.

OpenSSH is an implementation of the two versions of SSH protocols, SSH1 (SSH) and SSH2 (SecSH) and contains a suite of connectivity tools used for hosting an SSH server. In CLI Crawler OpenSSH and Telnet is used to host a SSH and Telnet server on a Linux platform. The server will run the CLI that is discovered by CLI Crawler.

To connect through SSH or Telnet to a remote system requires quite complex programming, however many developers have already developed such functionality thus there are various types of open-source Java APIs that may be used for this functionality.

3.2.1 Ganymed SSH-2 for Java

The CLI Crawler uses the Ganymed SSH-2 for Java API for the SSH connection. The standard procedure of using this API can be seen in Figure 3.3.

```
Connection connection = new Connection(hostname, port);
connection.connect();
Session session = connection.openSession();
// get the input and output stream
// perform some operations
session.close();
connection.close();
```

Figure 3.3 shows the standard procedure of using Ganymed SSH-2 for Java.

As seen in Figure 3.3, one has to first establish the connection, which is done by connecting to a certain hostname and port, and then opening the session. Once the session is open and the connection is established, one can perform operations to communicate with the remote system. Usually this is done by getting the sessions input stream with `session.getStdOut()` and the output stream with `session.getStdin()`. These streams are then used to handle the text that is being sent over the remote connection, making one able to simulate a shell.

3.2.2 Apache Commons (Telnet)

For the Telnet connection in CLI Crawler, the Apache Commons API is used. This API provides various tools such as functionality for Telnet connection. The standard procedure of using Apache Commons for Telnet can be seen in the code snippet in Figure 3.4.

```
TelnetClient telnetClient = new TelnetClient();
telnetClient.connect(hostname, port);
// get the input and output stream
// perform some operations
```

Figure 3.4 shows the standard procedure of using Apache Commons for Telnet.

As seen in Figure 3.4, this procedure is very similar to Ganymed. First a connection is established and then one has access to the input and output stream with `telnetClient.getInputStream()` and `telnetClient.getOutputStream()` which are then used to handle the communication over the remote connection in the exact same way as Ganymed.

3.3 Hierarchical data model

Many systems use different data modeling languages for both high and low-level purposes to store data in a format that is made to be easily managed by both humans and computers. Common languages that represent the data in a hierarchical model are for instance Extensible Markup Language (XML), Unified Modeling Language (UML) and YANG.

As for CLI Crawler, a hierarchical data model will be used to represent the CLI's command structure and then further used for configuration changes which make YANG the most viable option. It may however be important to understand the differences between YANG and other hierarchical data models to see why YANG is most suited for the purpose of CLI Crawler. Thus the three most viable options are explained in this section.

3.3.1 YANG

YANG is a data modeling language that is used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF) which is defined in RFC 4741 (Enns, 2006), NETCONF remote procedure calls and NETCONF notifications. YANG is defined in RFC 6020 (Bjorklund, 2010).

YANG is a relatively new data modeling language (published in September 2010) compared to other data modeling languages. YANG focuses on factors that other data modeling languages cannot handle as well, such as configuration management and being highly human-readable. Even though YANG is different from XML, it can be converted to a XML syntax called YANG Independent Notation (YIN). This allows applications with existing XML parsers to work with YIN.

YANG models a structure, a hierarchy of data similar to a tree structure as opposed to being object-oriented. YANG has various nodes such as parent and child nodes like a regular tree structure, but it also has other functions, such as dependencies, data types and different ways of modeling this hierarchy.

In YANG, the model is defined by a module where various kinds of meta-data is stored such as namespace, revision and description among others as seen in Figure 3.5.

```
module router {
  namespace "http://namespace/";
  description "Description...";
  revision 2015-01-01 {
    description "Initial revision.";
  }
}
```

Figure 3.5 shows the foundation of a model in YANG.

The module is what contains all of the different nodes in the model. In YANG there are four different types of nodes; container, leaf, list and leaf-list nodes. Each node has an id (or name) and either some kind of value or a set of sub-nodes.

```
container resources {
  leaf port {
    type uint16 {
      range "1 .. 9999";
    }
  }
  container other {
    leaf-list domain {
      type string;
    }
  }
}
```

Figure 3.6 shows the various nodes in a YANG model.

Figure 3.6 is a very simple example showing the basic hierarchical structure and the very basic functionality in YANG. A container node is a node that contains a set of sub-nodes, which can be any of the four node types, in this case there is a container `resources` and a container `other`. In the `resources` container there is a leaf node, `port` which has some kind of value (in this case `uint16` with a certain range). There are many different types that are already defined in YANG, such as `int` or `string` among others, one can also define own types if needed. The leaf-list `domain` in the `other` container is a list of “domains” which have the type `string`. The fourth node type is not present in this example, but it is the `list` node which has the purpose of grouping various leaf nodes into a list, making the list node a set of leaf nodes. For a more extensive example of a YANG model, see Appendix A.1, it includes all four types of nodes and also includes the module statements.

3.3.2 Extensible Markup Language

XML is used for many different systems and is used to define a set of short rules in a document, using a standard encoding format. XML itself is not a markup language, rather it is a “meta-language” that may be used to create markup languages for a specific application. The tags used to markup the document are formatted in a hierarchical structure as seen in Figure 3.7.

```
<system>
  <router>
    <port>22</port>
  </router>
</system>
```

Figure 3.7 shows how XML is used to represent hierarchical data.

3.3.3 Unified Modeling Language

UML is a general-purpose modeling language that represents a system’s architecture. UML was released in the late 90’s with the purpose to provide the development community with a stable and common design language. The language is mainly used to develop and build computer applications. The application is represented by several class diagrams that visually display the data architecture. This makes UML quite different as compared to

YANG or XML but ultimately it fulfills the same purpose in a different way. UML examples may be seen in Appendix C.2 through C.6 which show the design of CLI Crawler (these UML diagrams are further explained throughout section 5).

3.4 Expect

Expect is used to manage the input and output (IO) controls from a remote connection through a shell. When commands are executed in a sequence from a remote machine the program needs to know when it is appropriate to execute the next command. This is to prevent the application from crashing, which could occur if two commands would collide during an execution. It also provides closures for getting the complete output log of an executed command.

Because the remote server sends a continuous flow of data to the output stream when commands are executed, a closure is needed to separate the commands in the output. The issue is that there can only be one instance of an output stream (for example Java `System.out`) during the applications lifespan, which is why the stream cannot be opened and closed between commands.

This is solved by guessing (expecting) a single or multiple keywords and patterns in the output, then wait for the keyword to be detected by the expect API. Usually the command prompt is expected, since it indicates that the system is ready to accept new input. However the prompt will change depending on which system is used and on what level the user is located in the system.

There are different types of Expect tools for different programming languages. For CLI Crawler, Expect-for-Java was used. Expect-for-Java is simple but effective, with support for modification if needed.

3.4.1 Expect-for-Java

The standard procedure of Expect-for-Java can be seen in Figure 3.8.

```
Expect expect = new Expect(inputStream, outputStream);
expect.send("ls\n");
expect.expect("$", "#", Pattern.compile("(?:^|>(?:$|.*)"));
System.out.println(expect.before + expect.match);
```

Figure 3.8 shows the standard procedure of using Expect-for-Java.

First, one has to create the Expect object, for this an input and output stream is needed. With the method `expect.send()` the client sends some kind of command or information to the remote system, in this case "ls\n" which would display the current folder contents on a Linux platform. After a command is sent, some kind of output is expected, the expected output can be defined with the `expect.expect()` method, which takes a string or a number of strings (or combined with regular expressions (regex)) that it then expects. The output contents are thereafter stored in two variables in the expect object which can be accessed through `expect.before + expect.match`.

Expect-for-Java has functionality to log all of the procedures during execution, making it easier to troubleshoot. The log functionality is done with the use of the Apache Log4j API (The Apache Software Foundation, n.d.).

3.5 Neo4j

Neo4j is a graph database (Neo Technology Inc, n.d.) that uses the Cypher query language, similar to the Structured Query Language (SQL). The graph will represent the command structure of the CLI and will be constructed in real-time during the program execution. The database is also used to resume the state of the discovery algorithm where the application was terminated. Each command found during the discovery process will be inserted into the graph as nodes. Each node will have its own unique id that can be used to receive the node. A node can also contain different metadata such as labels, properties and relationships.

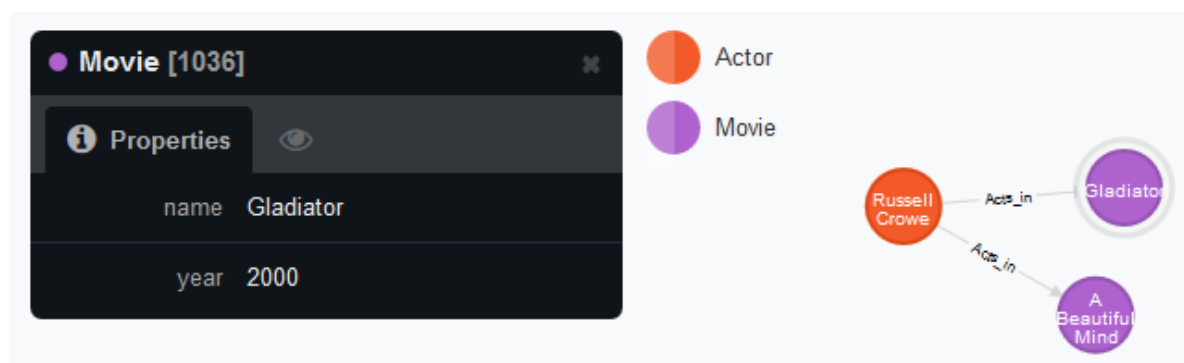


Figure 3.9 shows how data is stored in a Neo4j database.

The graph in Figure 3.9 has the labels Actor and Movie. The relationship-arrows in the graph state that the actor Russell Crowe acts in the movies Gladiator and A Beautiful Mind, note the direction of the arrows. The box to the left displays the selected node which has the assigned properties name and year.

3.5.1 Web service

Neo4j provides a web service located at <http://localhost:7474/browser/>, where localhost is the hosting IP address and 7474 is the port number. The web service is used to visually display the graph as shown in Figure 3.9, send Cypher queries (as shown in Figure 3.10) and distribute the contents of the database.

The web service is hosted as a REST API, which is an architectural style that is based on web-standards and the HTTP protocol. A REST architecture consists of resources. A resource can be defined as an object with a type, associated data, relationships to other resources, and a set of methods that operate on it and may be accessed via a common interface based on the HTTP standard methods, which are typically URIs. REST allows that resources different representations such as HTML, plain text, JavaScript Object Notation (JSON) or XML (Oracle, 2013). The Neo4j REST API uses JSON to represent resources, so that it can be used from many languages and platforms (this is further explained in section 5.2.2).

```
CREATE (m:Movie {name:"Gladiator", year: "2000"})
```

```

MATCH (a:Actor), (m:Movie)
WHERE a.name = 'Russell Crowe' AND m.name = 'Gladiator'
CREATE (a)-[r:Acts_in]->(m)
RETURN r

```

Figure 3.10 shows how the Gladiator node is created and how the relationship is set between the actor and movie with the Cypher language.

3.6 Graphical user interface

3.6.1 Java Swing

Swing is a GUI toolkit for Java which is a part of the Java SE (Holm, 2007). The basic idea is that the GUI is created from a collection of Swing components such as buttons and text fields that are placed on a JFrame (the main window).

3.6.2 JavaFX

JavaFX is a GUI framework for Java applications. JavaFX's long-term aim is to eventually substitute Swing as the standard GUI framework for Java SE, however Swing will remain a part of the Java SE at the present time (Oracle, n.d.).

The basis for a JavaFX application is the `Application` class in the JavaFX framework that essentially enables multithreading by creating an `application` thread. This requires that the GUI class extends `application` and overrides `start(Stage)`. The primary stage is constructed by the platform and is the top level JavaFX container. An application may create other stages, but they will not be primary stages.

The main class creates an instance of the GUI class when `Application.launch(String[])` is invoked, therefore no constructor is needed. Instead, the `start(Stage)` method from the GUI class is run by the `application` thread when `Application.launch(String[])` is invoked. This `start` method is where one places the code to be executed (just like a regular main method in Java).

JavaFX uses a visual layout tool, JavaFX Scene Builder to build applications. This is done by placing out containers in a hierarchical order in the Scene Builder. Each container will have its own properties such as size, position and id among others. The id is used to call the container externally from the Java class. The containers are stored in an `.fxml` file by the Scene Builder and are translated into an xml format. Each container can be accessed from the GUI class by first loading the `.fxml` file and then loading the container from its id as shown in Figure 3.11.

```

FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("file.fxml"));
Button button = fxmlLoader.getNamespace().get("myId");
button.setDisable(true);

```

Figure 3.11 shows how one can load a component from an `.fxml` file.

At first, the `.fxml` file has to be loaded into the Java project which is done with the `FXMLLoader` object. Thereafter the `FXMLLoader` object works as a loader for the various components that the `.fxml` contains. In Figure 3.11, the `FXMLLoader` loads a button with an id `myId` and stores it as a `Button` object. After a component is loaded, various operations

can be performed on the component, in Figure 3.11 the button is disabled right after loading it. In this particular case, the button was loaded from the .fxml file in Figure 3.12.

```
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import fxml.MyGroup?>

<MyGroup xmlns:fx="http://javafx.com/fxml">
  <children>
    <Button fx:id="myId" text="Click Me!"
onAction="#handleButtonAction" />
  </children>
</MyGroup >
```

Figure 3.12 shows the structure of an .fxml file.

An .fxml file contains information and values for each of the components in a certain scene, in Figure 3.12 the .fxml file has one component – a button. Even though the files are created with the use of Scene Builder, they may be manually modified and created without the use of it.

3.7 VirtualBox

VirtualBox is a cross-platform virtualization application used to emulate virtual machines (VM) (Oracle, n.d.). VirtualBox can load multiple guest operating systems (OS) under a single host system and supports Linux, Mac OS X, Windows and Solaris as guest OS. Each VM can be started, stopped and paused independently during the emulation and at exit the state of the VM can be saved and resumed at the same point.

VirtualBox was of great help during the development of CLI Crawler, a Linux platform was virtually emulated with the use of it. The CLIs that were tested during the development were installed on a Linux platform (Ubuntu and LinuxMint in the thesis work) which were reached by connecting through SSH or Telnet.

Since CLI Crawler has to work on a Linux platform, and the fact that CLI Crawler is developed in Windows, one could easily move the CLI Crawler application from Windows to the virtually emulated Linux platform to test it.

This provided increased flexibility during development as everything could be done on the same computer. The development was done on computers provided by Data Ductus, which had Windows installed on them. If the computers had a Linux platform as the operating system, no VirtualBox would be needed, but this was not optimal since the thesis workers did not have any previous experience with Linux platforms.

3.8 Dynamips

Dynamips is an open-source tool for emulating Cisco based routers on the command line. The principle is that the user who has a Cisco image, can virtually emulate this image using Dynamips. This provides various possibilities such as being able to understand, test and experiment on a virtual router, instead of doing it on a real router where experiments may have consequences. (GNS3, n.d.)

Since Dynamips emulates these Cisco routers on the command line, one can connect to a shell (such as a Linux shell) using SSH or Telnet and start the emulation. This is very relevant to the thesis since the emulation is on the command line which is the way CLI Crawler discovers a CLI's command structure.

Dynamips was used during the development of CLI Crawler to simulate CLIs. This made it possible to investigate them and test them with CLI Crawler.

4. Analysis

Understanding the background and purpose behind network configuration and its various tools such as CLI or YANG and how the application may be developed was very important, this could only be done by gathering and analyzing a lot of information (and open-source APIs which may be of use) from various sources.

Most of the information was gathered from people who work with the specific topics, for example employees at Data Ductus. Usually done by discussing a topic (often a topic that was being worked on, during a demonstration) and taking notes for future references, or getting tips of where one can read more about a certain topic on the internet. The discussions often provided quantitative information which was then further discussed in order to filter and prioritize the most important information.

An important part during the first phase of the thesis work was examining various available open-source software and APIs which could be of help for the application. Functionality which the application requires may have already been developed and published as open-source, which would result in that specific functionality to not have to be developed in the thesis work leading to the capability of focusing on other areas. However the APIs could not require any license for development or execution either as per Data Ductus request. This turned out to not be a major constraint since there were plenty of open-source APIs to choose from.

4.1 SSH and Telnet connection establishment

For some functionality, such as SSH and Telnet connection establishment there were multiple APIs to choose from even though most of them shared the same functionality. To find the optimal one depended heavily on a few factors, it needed some functionalities such as sending single commands and receiving the output in an efficient way.

Different APIs had to be investigated and while there was a lot of different APIs to choose from, a lot of them had minor issues which impacted CLI Crawler. Most of the issues were noticed when entering a CLI and trying to send commands within it. For instance some special characters were not fully supported which would result in odd characters to display and sometimes even terminate the connection to the remote host. This led to the choice of Ganymed SSH-2 for Java for SSH (Plattner, n.d.) and Apache Commons (The Apache Software Foundation, n.d.) for Telnet as they were seen as the most all-around stable APIs. However the choices of APIs were made before an expect tool was used and since the expect tool provided a lot of improvements it could have potentially made other APIs viable to use too.

For the Telnet connection there were two options to choose from – Apache Commons or telnetd (Wimberger, 2007). Apache Commons was chosen due to being simpler and more straight-forward to use than telnetd. However the choice of SSH API was not as easy, there were multiple good APIs to choose from that performed the desired functionalities well such as JSch (JCraft Inc, n.d.), J2SSH Maverick (SSHTOOLS, n.d.) or Ganymed SSH-2 for Java. J2SSH Maverick had no free-to-use license which did not make it a viable option and Ganymed SSH-2 for Java had better documentation and was much simpler to integrate with CLI Crawler than JSch which ultimately led to the choice of Ganymed SSH-2 for Java.

4.2 GUI toolkit

An important factor to consider was the choice of how CLI Crawler will look and how the user will interact with the program (since user interaction was a requirement) and whether it will have some kind of graphical interface or if it will be command line based. At first, CLI Crawler was developed to be command line based but soon enough the thesis workers realized that it may be better to develop a GUI due to the amount of different functionalities that are needed. This led to investigating different GUI toolkits for Java.

At first, a GUI prototype was developed with Java Swing (Holm, 2007), a screenshot of this GUI can be seen in Appendix B.1. It was quickly noticed that Swing was not optimal for CLI Crawler as it had several functional issues. The ignore list would sometimes not display at all and the scroll function in the various windows (especially in the output window) would not work in some cases.

A choice had to be made, either to solve the issues (which may not even be possible in a realistic timeframe) or investigate if there is any other effective GUI toolkit for Java. Since more functionalities had to be implemented into the GUI and the fact that there already was multiple issues with the current GUI a decision was made to see whether there is a potential substitution for Java Swing.

JavaFX (Oracle, n.d.) was noticed as a potential substitution. After investigating how JavaFX works and what possibilities it provides it was quickly noticed that JavaFX is most likely a better toolkit for the GUI. JavaFX is simpler to work with and it has many more functionalities providing much more possibilities for CLI Crawler. This led to the construction of a new GUI (which can be seen in Appendix B.2), which was both better looking and had better functionality (as well as more control over the functionality) than the GUI prototype that was made with Swing. The JavaFX GUI is also a lot easier to manage in the long run (to add, remove or change existing components or change entire functionalities) due to both having a better documentation and due to the JavaFX Scene Builder which provides a simple way of managing the GUI which may be important for further development.

4.3 Database

As per Data Ductus request, CLI Crawler should work in a way so that if one starts the discovery algorithm and then stops it prematurely, it should be possible to continue the algorithm from the stopping point on another occasion. For this to be possible, some kind of database had to be used to store progress.

Data Ductus recommended a graph database called Neo4j (Neo Technology Inc, n.d.). After investigating Neo4j, it was noticed that it would work very well with CLI Crawler as it could store database entries and dependencies both graphically and in table-form in a very efficient way. Since Neo4j was so efficient with CLI Crawler during the pre-study phase and that Data Ductus had previous experience with it, no other database tool was investigated as Neo4j had all of the desired functionality.

4.4 Expect

For the SSH and Telnet connection to work properly and for it to be able to handle single commands some kind of expect tool was needed.

Data Ductus recommended Expect4j (Verges & Ryan, n.d.), however it requires a very complex implementation in the project before working which was a time based issue. Failure of correct implementation would also lead to commands not being able to execute at all.

A decision was made to see if there are any other Expect tools for Java that were simpler and more straight-forward. ExpectIt (Gavrilov, n.d.) and Expect-for-Java (Dong, n.d.) were two options which were fairly similar. The choice ended up being Expect-for-Java as it had available source code and was much easier to implement (and to modify if needed).

5. Result

The result of the thesis work is a CLI Crawler prototype. In this section, the high-level design and the different functionalities of CLI Crawler will be explained.

A user manual for CLI Crawler can be seen in Appendix C.

5.1 Overview

In order to understand the different functionalities of CLI Crawler it is important to first get an overview over the application and how it works.

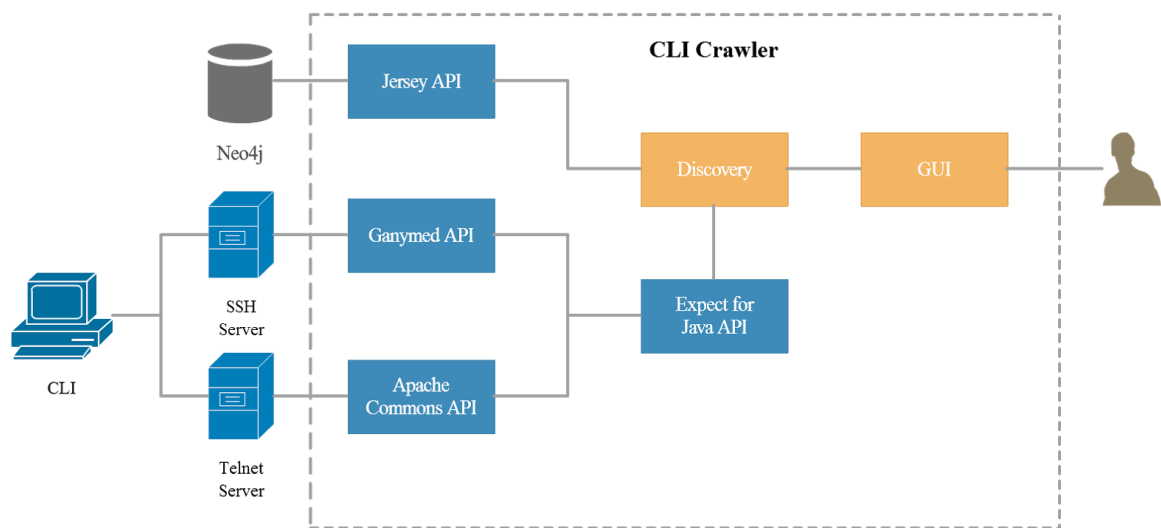


Figure 5.1 shows a context diagram of CLI Crawler.

The context diagram in Figure 5.1 gives an overview of how CLI Crawler interacts with external systems. The box labeled CLI Crawler displays the sub-systems that are included in the prototype. The items outside of the box represent the external systems that are used by CLI Crawler.

The CLI figure represents any CLI that is located on a remote server which can be accessed through SSH or Telnet. The Ganymed and Apache Commons APIs are clients used to communicate with their corresponding SSH or Telnet server. The clients use the Expect-for-Java API to regulate the IO flow between the server and CLI Crawler. Discovery is the core of the whole system and controls each background process used for the CLI discovery. It also manages the inbound traffic coming from the server and database. The database contains data which is extracted from a CLI during the CLI discovery and is accessed through the Jersey API, this data will thereafter be used to render a YANG model (which is not implemented in the prototype). The GUI is used to manage all interactions between the discovery algorithm and the user.

The context diagram represents a very general picture of CLI Crawler and the external systems, but it may also be important to understand the inner design of CLI Crawler. A representation of the packages within CLI Crawler and the classes within the packages can be seen in Appendix C.1 through C.6 as UML diagrams. Appendix C.1 represents CLI

Crawler as an entity, showing how the different packages and classes interact with each other without showing the contents of the classes. The contents of the classes can be seen in the UML diagrams representing the different packages in Appendix C.2 through C.6.

5.2 Functionality

5.2.1 Discovery algorithm

The discovery algorithm relies on the navigation of CLIs and their structure. It is handled by the `discovery()` and `discovery(String, int, Command, ArrayList<String>)` methods in the `Discovery` class.

Consider the pseudo-code in Appendix A.3 and A.4, these appendices show the basic functionality of the algorithm. At first the `discovery()` method is called, this method will initialize the algorithm, find and go through all of the top commands at the root of the CLI. For each of the top commands it will call the recursive method `discovery(String, int, Command, ArrayList<String>)` method on that particular command, this extended discovery method is recursive and has the job to go through all of the attributes, subcommands and modes of a certain top command depth-first. In order to manage a good structure of the algorithm and in order to store all of the information that is used and processed during the algorithm several utility classes are used as help, these are further explained in section 5.2.1.1.

Most CLIs support the use of question-mark to get an output of all possible commands and to check which attributes are possible for a certain command. One can just write a question-mark to get all of the commands at the root of a mode and one can write the command with a space and question-mark at the end to find the possible attributes for a certain command. This is the basic way to navigate through a CLI and is also the way the algorithm navigates throughout the discovery process.

The principle is that after sending a certain command, the output is parsed and split into words and thereafter each word is sent either as an attribute or command to see what happens and based on that to determine what to do next.

Since there may be some attributes or commands that can either cause issues or are not relevant to discover, CLI Crawler provides the opportunity to ignore certain attributes or commands which is done through GUI interaction. Commands that are ignored are removed from the output of each mode and attributes that are ignored are removed from all outputs which will make sure that they are not tested during the discovery algorithm.

Before the algorithm sends the words, either as an attribute or command, it will analyze the output. There are a lot of factors that can affect which words are going to be sent (if any at all), the most important ones are mentioned below.

- If a word is in the attribute or command ignore list it will be removed from the commands or attributes output.
- It is possible the output led to a new mode, then a new recursion is started.
- In some cases the output has a timeout which indicates that an input is needed which makes the algorithm test several test-commands in order to see if they are possible inputs, this includes strings, integers or booleans. If no valid input is detected by the algorithm the user will be needed in order to provide a valid input.

- An output from a command may be displaying just a set of files and folders. This means that the actual attribute to the command is either a file or path, therefore the output is edited so that each word of it is not tested. Instead a test-string is tested to see what further attributes the command has.
- It is possible that a command can take any string as its attribute, any integer or a certain range of integers. This is also detected by the algorithm as in cases where a command can take a string as attribute the output may have some kind of description which consists of words and therefore these words would be shown as possible completions to the command.
- If the output contains the tested command it may mean that the output is either showing the full command and its attributes or showing an example on how the command is used. The algorithm will in these cases try to detect such occurrences and test whether the potential attributes are real attributes.
- If the output contains some word that contains an ignored attribute or command the user will be asked first whether this word should still be tested. The same goes for certain special characters which are known to cause issues.
- If the prompt is in the output it will be removed in order to not test unnecessary words.
- The algorithm will check whether the output has occurred in the previous recursions in order to make sure the application does not end up in an infinite loop.
- Some CLIs may have support for abbreviations, for instance you may be able to type “c” instead of “config”. The algorithm will also detect these cases in order to avoid storing the same command more than once in the tree and graph.

In some cases there can be occurrences of words that do not provide an invalid output (from a description or similar) even though they are not a possible attribute or command. If the output for instance displays a set of files and folders, the algorithm will detect this and not send all of the displayed files and folders as attribute but instead a string as the attribute is just the path of a certain file. The same principle is used to check a lot of other possible scenarios, such as a command being able to take any string or integer as attribute, or if the command can take a range of integers as attribute.

It is also important to consider that different CLIs may have different ways of navigating through them and that the navigation may change after a software update of the CLI. For some CLIs it may be possible to use tab-completion while for others it is not possible. Since changes between different CLIs cannot entirely be predicted it is important to make the algorithm as general as possible. Making such an algorithm general is a complex task, so some specific factors have to be considered such as keywords for error messages or keywords for commands which can exit modes since the algorithm needs to have something to rely on to know whether it is testing a real command or not.

During the algorithm, a node-graph is constructed in the Database (which is further explained in section 5.2.2) and a tree is also constructed to show the user what the algorithm is doing in real-time by displaying a constantly updated tree in the GUI (this is further explained in section 5.2.1.2).

5.2.1.1 Utilities

To support the discovery algorithm there are some utility classes in the `util` and `algorithm.util` packages. `util` contains three classes – `TextUtilities` which provides several methods for handling text, `SimpleTree` for the tree representation (which

is further explained in section 5.2.1.2) and `Triplet` which is used to group three objects together.

To handle useful information about the CLI there are several CLI-specific utility classes in the package `algorithm.util`. These were implemented in order to have a well-structured algorithm and to easily be able to add more useful information to the objects in the future if needed.

Modes are stored with the use of the `Mode` class during the algorithm in order to avoid discovering a specific mode more than once. A `Mode` object stores three objects:

1. A list containing the commands in the root of the mode (which are stored as `Command` objects).
2. The `URI` node of the command that initially entered the mode.
3. The output the mode provides upon typing question-mark in the root of the mode.

As seen above, a `Mode` contains a list of commands which are stored as `Command` objects. It is important to note that a `Mode` object will only contain the root commands, but that a `Command` object is used for both attributes and commands in other cases. A `Command` object is used to group information about a certain attribute or command, with the main use of sending useful information about a discovered attribute or command to the next recursion of the algorithm. A `Command` object stores five objects:

1. A `Triplet` object containing three properties for the command as strings. The first string is the actual command that is sent and used during the algorithm, the second one is the name of the command (which can be different from the actual command as for instance a command with the name “screen-width” can have a certain number as its actual command) and the third one being the type of the command. In some cases, some of the properties are not used and are therefore stored as null in the `Triplet`.
2. The command as text.
3. The prompt at the place from where the command is sent.
4. The output of the command stored as an `Output` object.
5. The `URI` node of the command.

Finally, there is an `Output` object used for storing outputs. The `Output` object will only contain two pieces of information – the actual output represented as text and a boolean which indicates whether a timeout occurred when getting the particular output (the purpose of the boolean is further explained in section 5.2.3).

5.2.1.2 Tree representation

During a discovery a hierarchical tree is constructed in real-time which is constantly printed to the GUI in order for the user to see a hierarchical high-level design of the CLI. The purpose of this is to tell the user how much of the CLI the algorithm has discovered and give an overview of the CLI’s command structure. The tree is built by using a `SimpleTree` object during the discovery.

The tree structure is stored in a two dimensional list (matrix) which permits modifications to be made without the need to loop through the entire tree. The class is designed to allow an application to build a tree in real-time by inserting nodes depth-first. This means that it is

not possible to add new nodes to a previous branch with a higher index in the matrix, where x is the index. Figure 5.2 explains this further.

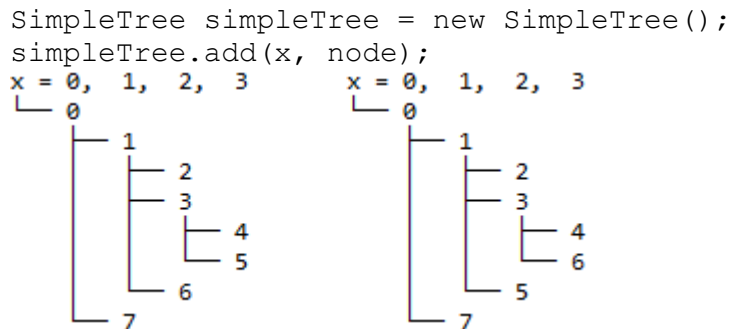


Figure 5.2 shows an example of two trees.

The left tree in Figure 5.2 shows a possible scenario for the insertion of nodes in a sequence of 0 to 7. The right tree shows a scenario that is not possible for the insertion of nodes in a sequence of 0 to 7 (note the order of the nodes 6 and 5).

`SimpleTree` is invoked in `Discovery` each time a command or attribute is found. This requires that the algorithm can calculate how far it has traversed in a CLI, to position the node on the correct branch.

5.2.1.3 Resuming a CLI Crawler discovery

Upon resuming a discovery the method `lastInstance(String)` will be called in order to get a list of the last command and attribute combination that was tested and in order to print out the tree that was displayed during the previous discovery process. This list will contain the commands and attributes that are needed to be sent in order to return to the stopping point of the previous discovery process (which means that already discovered commands will not be included in this list).

The discovery process is essentially the same even when resuming a discovery, however some parts are skipped. Each command and attribute will be sent as usual, but the output will be parsed from the index of the command or attribute in the list to the end of the output in order to not discover the already finished commands and attributes. Each time this parsing occurs the command or attribute in the list is removed.

During the parsing of some outputs user interaction could have been needed and when resuming a discovery these moments should not have to be repeated, this is solved by the boolean `resuming` in the `Discovery` class. Before each user interaction `resuming` is checked so that it is not true in order to perform the user interaction. Once the list has been emptied `resuming` will be set back to false. `resuming` is also used to determine whether a certain command or attribute should be added to the tree and database, which in the case of a resume it should not – as it already is added to both.

5.2.2 Database

The `Neo4jGraph` class is used to make transactions to the Neo4j REST API, the class relies on the Jersey API to do this. The Jersey API is an open-source RESTful Web Services framework that allows RESTful web services to be implemented in a Java servlet

container. Figure 5.3 shows how the Jersey client library may be used to communicate with the Neo4j RESTful web service.

```
Client client = Client.create();
WebResource resource=
client.resource("http://localhost:7474/db/data/cypher");

String query = "MATCH (c:Command { name:'Autowizard' }) RETURN c";

ClientResponse response =
resource.accept(MediaType.APPLICATION_JSON)
.type(MediaType.APPLICATION_JSON)
.entity("{\"query\": \"" + query + "\"}")
.post(ClientResponse.class);

String data = response.getEntity(String.class);
```

Figure 5.3 shows how one can implement a client to send cypher queries.

The URI <http://localhost:7474/db/data/cypher> from Figure 5.3 is the HTTP endpoint used to send cypher queries via the Neo4j REST API. The `Client` is used to connect to the specified endpoint, where the transaction takes place. The `WebResource` is used to post the query as a JSON resource and get the response from the REST API. The response is captured by the `ClientResponse` as a list of string headers and columns, containing HTTP representations of the field values such as node, relationships, properties, labels or other types of metadata used in the database. The method `response.getEntity(class)` is used to convert the client response into `String`, an example of the response is shown in Appendix A.2.

A node is represented by the URI http://localhost:7474/db/data/node/{node_id} where `{node_id}` is the unique id of a node, which can be used to create a node in the database. By appending [/labels](#) to the node URI it is possible to modify the nodes label. It is also possible to add properties to a node by appending [/properties/{property_name}](#) to the node URI, where `{property_name}` is the name of the property type.

In the `Discovery` class it is necessary to pass on nodes with each iteration during the recursive process of `discovery(String, int, Command, ArrayList<String>)`. Since a node can be represented by an URI http://localhost:7474/db/data/node/{node_id}, it is possible to pass on the URI to the next iteration which is done by storing the URI in a `Command` object which is thereafter passed on to the next iteration. This makes it possible to get a node without using queries and traversing the database. The method `createNode()` in the `Neo4jGraph` class is used for this purpose, it creates an empty node in the database and returns the URI location of the node. The `Neo4jGraph` methods `addLabel(URI, String)`, `addProperty(URI, String, String)` and `addRelationship(URI, URI, String, String)` are used to add labels and properties to the node, using the node URI and corresponding HTTP endpoint (`addRelationship(URI, URI, String, String)` uses the URI of both nodes that are connected).

The method `sendTransactionalCypherQuery(String)` is used to send custom Cypher queries and is based on the code in Figure 5.3. The method is mainly used to read from the database, usually when `MATCH`, `WHERE`, `RETURN` and `DELETE` query statements are

required to get or manipulate data in the database. The method returns the client response received from the server.

When returning a node property, the response must be reformatted in order to access the relevant data in the headers and columns of the client response. The response is parsed by the method `getProperty(String, String)` and returns a set of strings containing the property values received from each node in the response.

There are two types of trees in the graph. A tree containing the CLI's structure and a tree containing the list items used in the customization lists of the GUI (which are further explained in section 5.2.4). Each tree has a root node with the property "name" containing the name of the CLI, to avoid data being overwritten by another instance.

The nodes connected to the CLI's command structure have these properties:

1. name – The name of the command.
2. command – True if the command is not an attribute.
3. container – True if the command is a YANG container.
4. type – The type of the command.
5. input – The actual command that is sent during the discovery.
6. mode – True the command returns a mode.
7. carriageReturn – True if it is possible to press enter on the command.

The root-node in a tree containing list items have the label "ListView" and may be connected to nodes with the labels "Command" and "Attribute" that represent each list in the GUI. These nodes have the property "name" that contains the name of the command that is placed in the list.

5.2.3 Communicating with the remote shell

The communication between the discovery algorithm and the remote system is handled by three classes;

1. RemoteShell
2. Expect
3. Output

`RemoteShell` is the primary class out of the three, it is the class that connects to a remote system using SSH or Telnet and is also the class which sends commands and receives outputs.

```
RemoteShell remoteShell = new RemoteShell(hostname, port,
username, password);
remoteShell.startSSH();
Output commandOutput = remoteShell.send(command);
System.out.println("Sent the following command: " + command);
System.out.println("Received the following output: " +
commandOutput.toString());
```

Figure 5.4 shows the standard procedure of using a RemoteShell object.

At first a `RemoteShell` object has to be created, thereafter the connection is established with the use of `startSSH()` or `startTelnet()`. Upon calling the respective start method an `Expect` object is created within the `RemoteShell` object (which is part of Expect-for-

Java) in order to handle the input and output streams. This `Expect` object is then utilized within the send commands in `RemoteShell`.

There are three types of methods that will send some kind of command and receive some kind of output.

1. `getPrompt()` will send an empty message in order to get an empty output and then parse the out everything but the prompt and return the prompt.
2. `send(String)` will send a certain command and return the output.
3. `sendCheck(String, boolean)` will send a certain command with a question-mark at the end of it. Since CLIs may vary, some CLIs may not need an enter to be sent after the command since the command may be triggered right when the question-mark is typed and the output is then displayed, hence the use of the boolean `needEnter`. In some CLIs that trigger the command when the question-mark is typed the command may still be on the prompt which can cause some issues when the next command is sent. In order to work around this, after the output is displayed a `ctrl + u` keystroke is sent in the form of a byte (which is the default Linux keyboard shortcut to remove anything that has been typed on the current line).

Even though the output is just a string, it is important to know how the output was obtained, which is the reason the send methods in `RemoteShell` return `Output` objects which contain both the output in string but also a boolean `hadTimeout`. As a command is sent, a prompt is expected and is usually found. However if no prompt is found, an exception is thrown by the `expectOrThrow(Object...)` method from `Expect` which is called in the `send(String)` method. This exception is caught and then the prompt is no longer expected, instead a regular expression is now expecting the last character in a string (with the use of `Pattern.compile(".$")`).

If an exception was thrown an `Output` object is returned with the output and the boolean `hadTimeout` set to `true`, while if no exception was thrown `hadTimeout` would be set to `false`. Once the command is sent and the output is received, one can obtain the output by calling the `toString()` method in the `Output` object and one can know whether a timeout occurred by calling `hadTimeout()`.

The `hadTimeout` boolean indicates on whether the output was obtained by finding the prompt or not. This is very useful information as in some cases a sent command does not immediately provide an output but can in some cases need a certain value to continue. Thus the output does not contain the prompt but a line stating that the user has to type in some kind of value to be able to continue. By detecting whether a timeout occurred, one can see if the command is not finished but needs additional information, but it can also help in identifying potential issues such as crashes (which could for example make the CLI Crawler jump out of the CLI or jump back into the root of the CLI).

5.2.4 GUI

The GUI is built using the JavaFX platform and is managed by the `Main` class in the `application` package. The `Main` class initiates the GUI and thereafter the GUI is managed by the `Discovery` class during the discovery algorithm which makes use of an inner class `ComponentManager` which is further explained in section 5.2.4.5.

The GUI consists of two setup-windows used at launch of CLI Crawler and a main GUI layout. The main GUI layout which can be seen in Appendix B.2 is designed to be fully customizable and scalable during execution. Each window may be resized, closed and repositioned as needed. It consists of the following four windows:

1. Debug
2. Output
3. Customization
4. Command structure

The different windows and the GUI functionalities are further explained throughout this section.

5.2.4.1 Setup

When CLI Crawler is started, the login screen is prompted which lets the user enter the connection information to the server and the Neo4j database. A screenshot of the login screen can be seen in Figure 5.5.

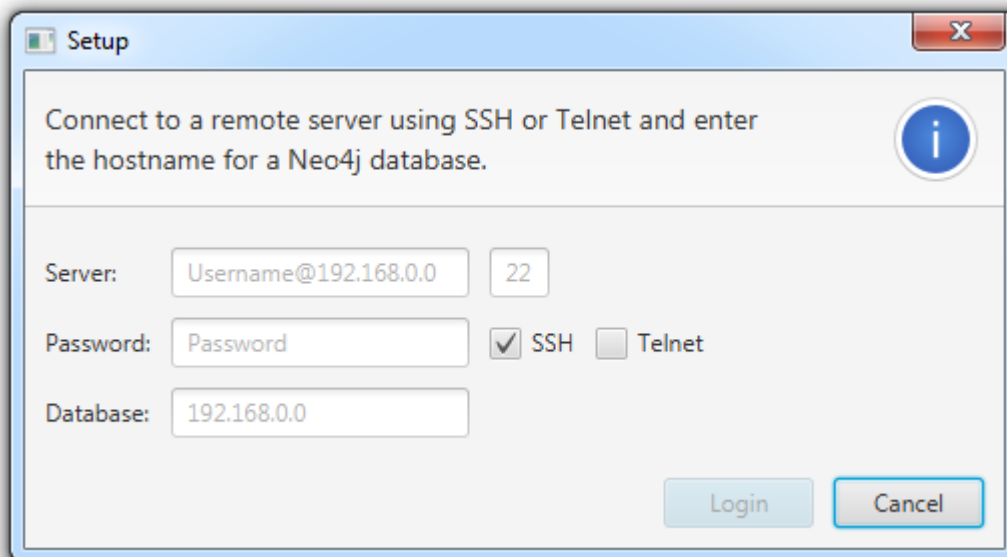


Figure 5.5 shows the connect window of CLI Crawler.

Once connected to both the server and Neo4j database a second screen is displayed – the console screen. A screenshot of the console screen can be seen in Figure 5.6. In the console the user is required to run the CLI in order for the discovery process to work since CLI Crawler cannot know where the user has the CLI installed. The console screen also lets a user select a previously used CLI from a drop-down menu or add a new one if it is a new CLI that is being discovered. If a previously used CLI is selected the algorithm will resume where the previous discovery was terminated.

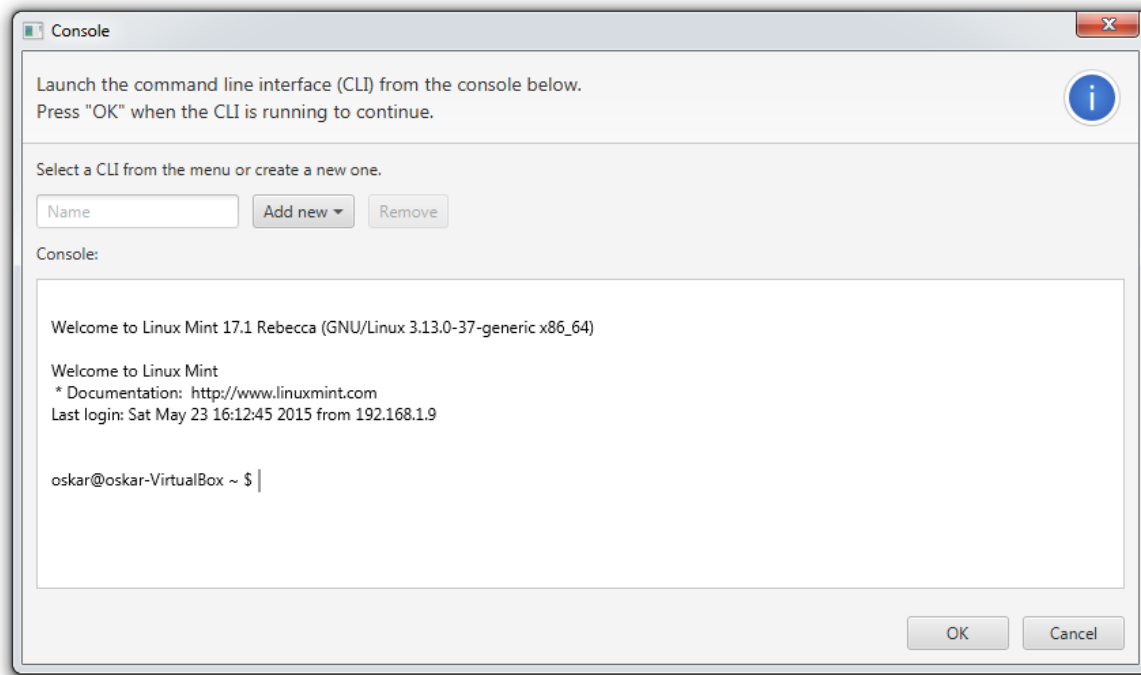


Figure 5.6 shows the console window of CLI Crawler.

5.2.4.2 Output and debug

The output and debug windows are used to handle user interaction, a screenshot of these can be seen in Figure 5.7. The output window is used to display outputs provided by commands during the discovery. This happens when user interaction is prompted and thus the output for the command is displayed to aid the user. The actual user interaction and instructions is managed in the debug window at the same moment.

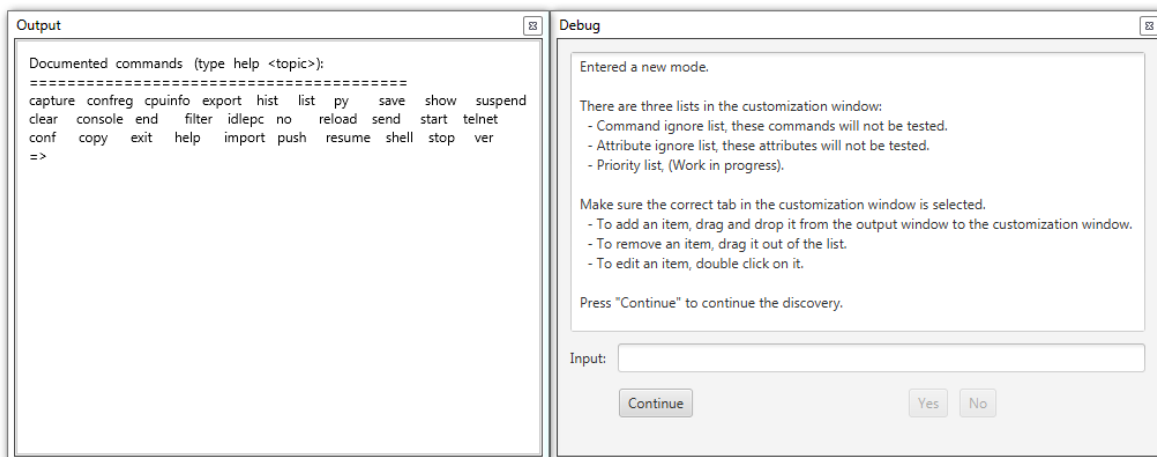


Figure 5.7 shows the output and debug windows.

Each time the algorithm enters a new mode the algorithm will pause and wait for the user to press continue in the debug window, during this stage it is possible for the user to add items to the lists in the customization window (which is further explained in section 5.2.4.3) by dragging items from the output to a list.

Since user interaction may be needed several times during a discovery, the algorithm will pause and ask for a certain user input through the debug window. The input is often just a simple yes or no button click but sometimes an actual input may be needed.

5.2.4.3 Customization

The customization window is used to display the ignore lists used for attributes and commands and also to manage priorities. Each tab in the customization window contains a certain list. One can either drag-and-drop items from the output window to add them to a certain list or use the input field at the bottom of the customization window. It is also possible to adjust the internal order of the items in the list which is important when managing the priorities in the priority list. One may also double-click on a certain item in a list to edit it.

5.2.4.4 Command structure

The command structure window is used to display a tree representing the current CLI's command structure. The command structure is updated and displayed in real-time during a discovery process for the tree that is being discovered. This is as previously mentioned done by using the utility class `SimpleTree` in `Discovery`.

5.2.4.5 Managing the GUI during a discovery

Since the `Main` class cannot be initialized, the GUI objects need to be sent over to other classes through constructors in order for other classes to manipulate the GUI components. The solution was to create a `ComponentManager` class within the `Main` class.

`ComponentManager` provides the ability for manipulation of certain GUI components outside of the `Main` class. The idea is that the `ComponentManager` object is created within the `Main` class and thereafter sent through a constructor to `Discovery`.

`ComponentManager` has methods which provide the ability to modify certain components, and more can be added easily if needed. It could have also been possible to simply make `ComponentManager` return the different GUI objects, however one would need make use of `Platform.runLater(new Runnable() {})` in order to call methods in the GUI objects which would lead to a lot of more code and less human readability. This way of designing `ComponentManager` makes it so that all of the GUI component manipulation is in one place, simply to provide easier and more understandable programming.

`ComponentManager` may not be the best possible way of designing this particular feature, but it is very sufficient since CLI Crawler does not need many GUI manipulation calls. However if there is a need of many GUI component manipulations, `ComponentManager` should perhaps return the GUI component objects and a private class should be implemented within `Discovery` which makes use of for example the strategy pattern in order to execute the GUI manipulation calls.

6. Conclusion

It is possible to develop a framework for CLI discovery which is shown by the CLI Crawler prototype. The most important factor for such a framework is the accuracy of the discovery, the discovery algorithm needs to discover the CLI's command structure very precisely since the structure will thereafter be used to make configuration changes. Another important factor is to make it as automatic as possible, but the more automatic it is, the risk of inaccuracy increases, so the ultimate solution is a balance between both factors. Even though it may not be possible to make the framework fully automatic it seems to be possible to make it very close to fully automatic with little need of user interaction.

By considering the questions at issue which are previously mentioned in section 1.4 and answering them during the thesis work has helped with achieving the objectives of the thesis work. The questions at issue may also be seen below. They are answered and analyzed throughout section 6.

1. Which CLI would be the most relevant to investigate and show as a high-level design in the pre-study?
2. What is the best way to access the CLI's command structure?
3. How can the CLI's command structure be stored before being exported to YANG?
4. How can the application be developed in order to be as automated as possible?
5. Are there any open-source software which may be used as help for the application?

CLI Crawler uses the principle of navigation of CLIs as foundation for the algorithm and tests different words detected in outputs to see if they are commands or attributes. Since it does seem like the framework could become close to fully automatic with this way of accessing and managing the CLI throughout the algorithm it does seem like this is a viable way to perform CLI discovery. The main issue with this way of discovering a CLI's command structure is the total time it takes to discover commands and attributes (this is further discussed in section 6.1).

The main focus during the pre-study and the development of CLI Crawler was a simulated CLI using a network device simulator, as it was seen to be the most relevant CLI to focus on since Data Ductus is very familiar with it. It is well documented and structured which means that information about the configuration possibilities of it are well known. Since the configuration possibilities were well known it was easy to verify whether the different functionalities of CLI Crawler worked as intended when testing them during development. To develop a discovery algorithm while focusing on this CLI naturally led to the need of considering many factors in order to make the algorithm as accurate, general and automatic as possible since it is quite a large CLI.

CLI Crawler does manage to discover most of the simulated CLI with a few minor exceptions (which are discussed in section 6.1) and display a hierarchical high-level representation of the CLI's command structure as a tree in the GUI and as a graph in the Neo4j database. This hierarchical high-level representation is accurate and fairly easy to understand but the best possible way to represent the structure of a CLI would be as YANG which is the next important step in the further development.

Most of the algorithm turned out to be general, as a result of this the algorithm should be of use for discovering other CLIs too (with perhaps some changes). Since the simulated CLI was the primary focus and not that many other CLIs were investigated it is difficult to say

how well CLI Crawler would work with other CLIs but it is certain that the foundation of the algorithm and CLI Crawler will be useful when expanding the program and making sure it works with more CLIs during further development. Investigating more CLIs in order to determine what needs improvements is therefore vital to make CLI Crawler a viable program to use. There are also a few other factors to consider before making CLI Crawler a viable solution for CLI discovery which are discussed throughout section 6.1.

The final conclusion is that a framework for CLI discovery is very possible which is proven by the CLI Crawler prototype. There is still work to be done before CLI Crawler will be viable to use for CLI discovery, but with some further development there is definitely possibilities of making CLI Crawler both precise and automatic while being able to discover the command structure of many different CLIs.

6.1 Further development

CLI Crawler provides the foundation of a CLI discovery framework, a lot of the features can be developed to become even more effective and features that make use of user interaction can become more automatic. There is a lot of small changes that can be made to CLI Crawler that do not really affect the discovery framework itself but makes the program better in general. This could be for example features such as saving connection information at the set-up screen or making the console in the console window work better and be more consistent. But as mentioned previously, the main focus is to make an accurate and effective algorithm for CLI discovery.

The algorithms computing time could potentially be reduced, there are currently four factors which take a lot of computing time during the algorithm.

1. Timeouts.
2. No list detection during algorithm.
3. Printing the tree in the GUI.
4. Analyzing commands.

If timeouts are handled in a different and more efficient way (by expecting in a different way and perhaps not waiting for exceptions) it would reduce the computing time by a lot, since it can currently take between one and four seconds just to handle a timeout for a certain command. Another factor that takes a lot of time is that lists (which are previously explained in section 3.1) are currently not detected during the discovery, this leads to testing the same commands multiple times since it will test all possible combinations of a certain list. The idea was to detect lists during the export from the database but the optimal solution would be to somehow detect lists during the discovery, since then all possible combinations will not have to be tested. Displaying a tree in real time in the GUI is a tricky task since previous entries in the tree have to be edited in order to maintain the tree structure. Currently the tree is restructured for every entry that has to be added which leads to unnecessary computing time, although this does not affect smaller trees that much it will still take noticeable computing time for larger trees which means that the computing time used by the tree will increase over time during the discovery. When analyzing commands there are a lot of minor functionalities that are working, but together they can lead to a lot of computing time. Making sure what a command does and what its attributes are uses a lot of different functionalities, an example is that every command is for example tested with a test-string and a test-integer to check whether the attribute could be any possible string or any possible integer. Most of the words in an output are also tested to see whether that word

is a possible attribute or command. All of this adds up which makes minor functionalities like these take a lot of computing time together.

In the CLI Crawler prototype, user interaction can be needed in some stages during the discovery and thus the program is paused and waits for a user input, but it is not easy for the user to predict when the algorithm will stop and need a user input to continue. The best solution would be to let CLI Crawler remember which modes or commands need user interaction and save the discovery of those modes or commands for the end of the discovery, this would lead to everything else that can be discovered automatically being done first and then at the very end of the discovery process all of the user interaction would be handled.

A settings window in the GUI can also be useful for CLI Crawler. Currently there are manually coded keywords in the `Discovery` class which are for instance used to determine whether a certain word gave an invalid output. The user should be given the possibility to edit such keywords if needed (or for instance navigation methods) which would be done in a settings window. If a new CLI has to be tested and the CLI has different ways of navigation or the way it displays outputs the user could perhaps still discover the new CLI with a couple of changes in the settings. Another potential solution would be to develop some automatic process that is run in the beginning of a discovery which detects the keywords and navigation methods itself. For this to be as efficient as possible it would be of help to investigate a lot of different CLIs as CLI Crawler was developed with the focus on mainly the one simulated CLI.

There may be certain commands which impact the CLI itself, common commands are quit or exit which exit the CLI or commands related to the prompt or the screen size which can change the prompt or the size of the screen which can have a major impact on CLI Crawler. While these commands may not be relevant to discover as they will most likely not be used for configuration changes it can still be a good idea to handle them.

Most of the improvements mentioned throughout this section improve existing features of CLI Crawler, there are still however two major features that are left to develop in order to make CLI Crawler viable to use for CLI discovery.

The first one being the detection of dependencies within a CLI, there may be a command which is dependent on another command before it can be run. This is currently not detected but is important to consider as some command may for instance not display its attributes if it is dependent on another command to be run first, this would therefore be stored as a YANG leaf-ref which would reference the dependencies with a link between each other in the YANG model. The idea would be to detect a dependency and then let the algorithm skip the command where the dependency was detected and come back to it at a later time to see if it still is dependent on something and perhaps make use of user interaction to solve the dependency.

The second one is exporting the CLI structure as YANG. The CLI structure is stored in the Neo4j database, so essentially all that the program needs to do is to go through this database and write YANG formatted files. In order for this to work properly, more metadata may have to be stored in the properties of nodes during their insertion to the database. If metadata is stored in a general way this could provide the possibility to not only export the

CLI structure as YANG but perhaps also as other formats if other formats become relevant to use with CLI command structures.

7. Terminology

API	An application programming interface is a specification used as an interface to let software components in a system communicate with each other.
CLI	Command line interface – a text-based interface for communication between a user and a system or software.
CLI Crawler	The prototype that was developed during the thesis work.
Cypher	A query language used by the graph database Neo4j, similar to the Structured Query Language (SQL).
Discovery algorithm	Algorithm used to detect a CLI's command structure.
Dynamips	Command line based emulation tool for Cisco routers.
Expect	Tool for handling interaction (input and output streams) with a text-based program such as a shell.
Expect-for-Java	Expect tool specifically made for Java.
JavaFX	Java based GUI framework.
JSON	JavaScript Object Notation is a language independent human readable data format which is mainly used for transmitting data between a server and a web application.
Mode	A CLI is often divided into modes which contain their own set of commands.
Neo4j	An open-source graph database, implemented in Java.
NETCONF	NETCONF is a protocol which is used as a communication tool between client and server. With this the client can manage and manipulate network devices and configuration data on the server.
Network device-simulator	A service instance in a network of real or simulated devices.
Prompt	A character or a sequence of characters that indicate that the shell is ready to receive input.
Regular Expression	A regular expression is a combination of characters used for text parsing and pattern matching.
REST	A software architecture style for designing network-based applications.
(Text) Shell	Text based user interface.
SSH	Secure Shell network protocol.
Swing	GUI framework included in Java SE.
Telnet	Telnet network protocol.
XML	Extensible Markup Language is a language made to be both human and machine-readable.
YANG	High-level data modeling language made for specifically configuration and state data changes by NETCONF.

8. References

Alexanderson, K., 2012. *Källkritik på Internet*. [Online]

Available at: <https://www.iis.se/docs/Kallkritik-pa-Internet.pdf>

[Accessed 09 06 2015].

Bjorklund, M., 2010. *A Data Modeling Language for the Network Configuration Protocol*.

[Online]

Available at: <http://tools.ietf.org/pdf/rfc6020.pdf>

[Accessed 09 03 2015].

Dong, R., n.d. *Expect-for-Java*. [Online]

Available at: <https://github.com/ronniedong/Expect-for-Java>

[Accessed 11 03 2015].

Enns, R., 2006. *NETCONF Configuration Protocol*. [Online]

Available at: <https://tools.ietf.org/html/rfc4741>

[Accessed 10 04 2015].

Gavrilov, A., n.d. *ExpectIt*. [Online]

Available at: <https://github.com/Alexey1Gavrilov/ExpectIt>

[Accessed 09 04 2015].

GNS3, n.d. *Dynamips*. [Online]

Available at: <http://www.gns3.net/dynamips/>

[Accessed 24 03 2015].

Holm, P., 2007. In: *Objektorienterad programmering och Java*. Lund: Studentlitteratur, p. 247.

JCraft Inc, n.d. *JSch - Java Secure Channel*. [Online]

Available at: <http://www.jcraft.com/jsch/>

[Accessed 14 04 2015].

Kniberg, H. & Skarin, M., 2010. *Kanban and Scrum, making the most of both*. [Online]

Available at: <http://www.infoq.com/minibooks/kanban-scrum-minibook>

[Accessed 25 03 2015].

Neo Technology Inc, n.d. *How to use the REST API from Java*. [Online]

Available at: <http://neo4j.com/docs/stable/server-java-rest-client-example.html>

[Accessed 15 04 2015].

Neo Technology Inc, n.d. *Neo4j*. [Online]

Available at: <http://neo4j.com/>

[Accessed 25 03 2015].

Oracle, 2013. *What Are RESTful Web Services?*. [Online]

Available at: <https://docs.oracle.com/javae/6/tutorial/doc/gijqy.html>

[Accessed 09 04 2015].

Oracle, n.d. *Is JavaFX replacing Swing as the new client UI library for Java SE?*. [Online]
Available at: <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>
[Accessed 09 04 2015].

Oracle, n.d. *JavaFX - The Rich Client Platform*. [Online]
Available at: <http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html>
[Accessed 09 04 2015].

Oracle, n.d. *Package javax.swing*. [Online]
Available at: <http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
[Accessed 09 04 2015].

Oracle, n.d. *VirtualBox*. [Online]
Available at: <https://www.virtualbox.org/>
[Accessed 30 03 2015].

Plattner, C., n.d. *Ganymed SSH-2 for Java*. [Online]
Available at: <http://www.ganymed.ethz.ch/ssh2/>
[Accessed 12 03 2015].

Postel, J. & Reynolds, J., 1983. *Telnet Protocol Specification*. [Online]
Available at: <https://tools.ietf.org/html/rfc854>
[Accessed 11 03 2015].

SSHTOOLS, n.d. *Java SSH Client*. [Online]
Available at: <https://www.sshtools.com/en/products/j2ssh>
[Accessed 14 04 2015].

Study CCNA, n.d. *Telnet & SSH*. [Online]
Available at: <http://study-ccna.com/telnet-ssh>
[Accessed 11 03 2015].

The Apache Software Foundation, n.d. *Apache Commons*. [Online]
Available at: <http://commons.apache.org/>
[Accessed 12 03 2015].

The Apache Software Foundation, n.d. *Apache Log4j 2*. [Online]
Available at: <http://logging.apache.org/log4j/2.x/>
[Accessed 21 04 2015].

Verges, C. & Ryan, J., n.d. *Expect4j*. [Online]
Available at: <https://github.com/cverges/expect4j>
[Accessed 09 04 2015].

Wimberger, D., 2007. *telnetd*. [Online]
Available at: <http://telnetd.sourceforge.net/>
[Accessed 14 04 2015].

Ylonen, T. & Lonvick, C., 2006. *The Secure Shell (SSH) Transport Layer Protocol*.
[Online]
Available at: <https://www.ietf.org/rfc/rfc4253.txt>
[Accessed 10 04 2015].

Appendices

Appendix A – Code snippets

Appendix A.1

```
module example-system {
    namespace "http://example.com/system";
    prefix "example";
    contact "person@example.com";
    description "The example system.";
    revision 2015-03-05 {
        description "Initial revision.";
    }
    container system {
        leaf hostname {
            type string;
        }
        list interface {
            key "name";
            leaf name {
                type string;
            }
            leaf id {
                type int;
            }
            leaf type {
                type enumeration {
                    enum ethernet;
                    enum atm;
                }
            }
        }
        leaf-list domain {
            type string;
        }
        ...
    }
}
```

Example of a YANG model.

Appendix A.2

```
{
  "columns" : [ "c" ],
  "data" : [ [ {
    "outgoing_relationships" :
"http://localhost:7474/db/data/node/0/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/0/labels",
    "data" : {
      "name" : "Autowizard"
    },
    "traverse" :
"http://localhost:7474/db/data/node/0/traverse/{returnType}",
    "all_typed_relationships" :
"http://localhost:7474/db/data/node/0/relationships/all/{-
list|&|types}",
    "self" : "http://localhost:7474/db/data/node/0",
    "property" :
"http://localhost:7474/db/data/node/0/properties/{key}",
    "outgoing_typed_relationships" :
"http://localhost:7474/db/data/node/0/relationships/out/{-
list|&|types}",
    "properties" :
"http://localhost:7474/db/data/node/0/properties",
    "incoming_relationships" :
"http://localhost:7474/db/data/node/0/relationships/in",
    "extensions" : {
    },
    "create_relationship" :
"http://localhost:7474/db/data/node/0/relationships",
    "paged_traverse" :
"http://localhost:7474/db/data/node/0/paged/traverse/{returnType}{
?pageSize,leaseTime}",
    "all_relationships" :
"http://localhost:7474/db/data/node/0/relationships/all",
    "incoming_typed_relationships" :
"http://localhost:7474/db/data/node/0/relationships/in/{-
list|&|types}",
    "metadata" : {
      "id" : 0,
      "labels" : [ "Command" ]
    }
  } ] ]
}
```

Example of a client response formatted in JSON.

Appendix A.3

Get the prompt

Send a question-mark to detect the top commands of the CLI
// Other operations such as detecting whether a restart is
currently run

For each top command

 Perform a discovery on that top command

Code snippet showing pseudo-code for the discovery() method.

Appendix A.4

```
Send the command with a question-mark at the end
If there was no output
    Handle the output and possibly end the current recursion
If it is resuming a previous discovery process
    Parse the output
If the output is the same as the previous output
    End the current recursion
If the command is an abbreviation (has been previously tested)
    End the current recursion
If the output had a timeout
    Handle the timeout
If the command is just a question-mark
    Count up the level
    If the mode has been previously discovered
        Set the output to empty
    Else
        Add the mode to the graph
        Set the current mode as this mode
        Add the mode to the mode list
Parse the output
If the output is not empty
    Create a new node, thisNode
    If the command is just a question-mark
        Add a label "mode" to the last node
        Append the mode ID to the tree
        Set thisNode to the last node.
    Else
        Add the command to the tree
        Add properties to thisNode
        Add the node to the current mode
        Add a relationship between the last node and thisNode
Further parse the output word by word
If the output is not empty and has no errors
    Split the output into a list of lines
    For each line in the output
        If the line contains the command
            If the line contains valid attributes or commands
                Perform a new discovery on these
If no attributes or commands were found
    Split the output into a list of words
    For each word in the output
        If the algorithm is in the root of the mode
            Perform discovery on word
        Else
            Perform a discovery on command + " " + word
Send the command
If the output had a timeout
    Handle the timeout
If there is a new prompt
    Set the prompt to the new prompt
    Perform a discovery on "" (no command - new mode)
If the command is just a question-mark
    Send an exit command
    If the output had a timeout
        Handle the timeout
```

```
    While the prompt is not the previous mode prompt
        Send an exit command
        If the output had a timeout
            Handle the timeout
        Set prompt to the new prompt
    If the parsed output is not empty
        Return this attribute or command
    Else
        Return an empty string
```

Code snippet showing pseudo-code for the `discovery(String, int, Command, ArrayList<String>)` method.

Appendix B – Screenshots

Appendix B.1

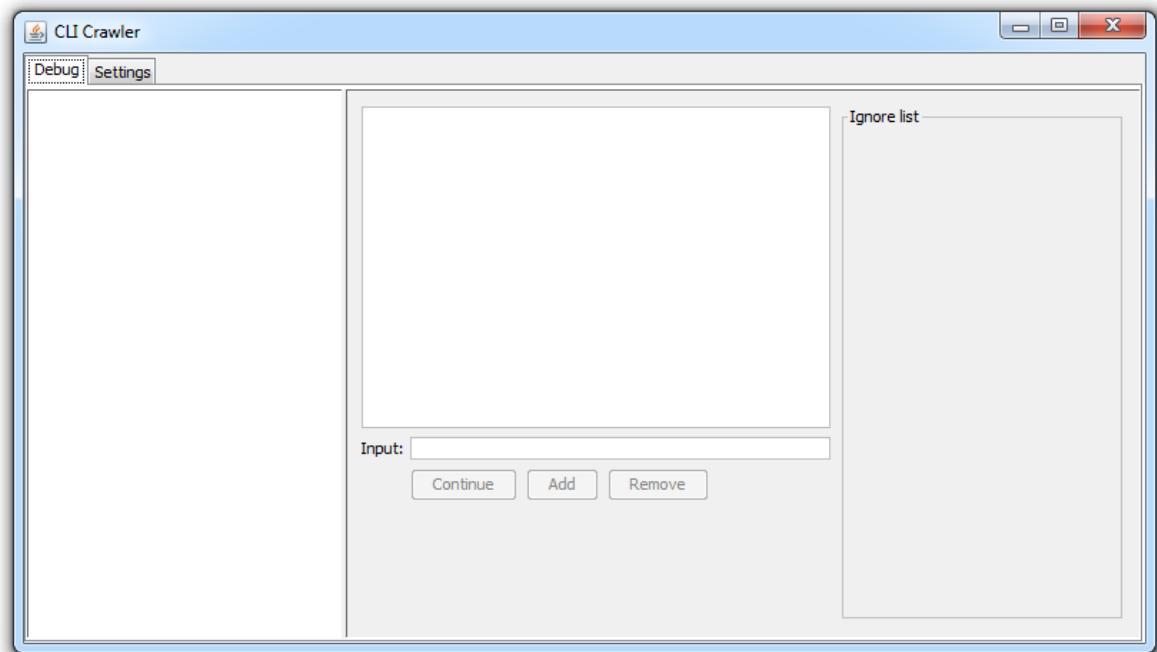


Figure showing the initial GUI of CLI Crawler using Java Swing.

Appendix B.2

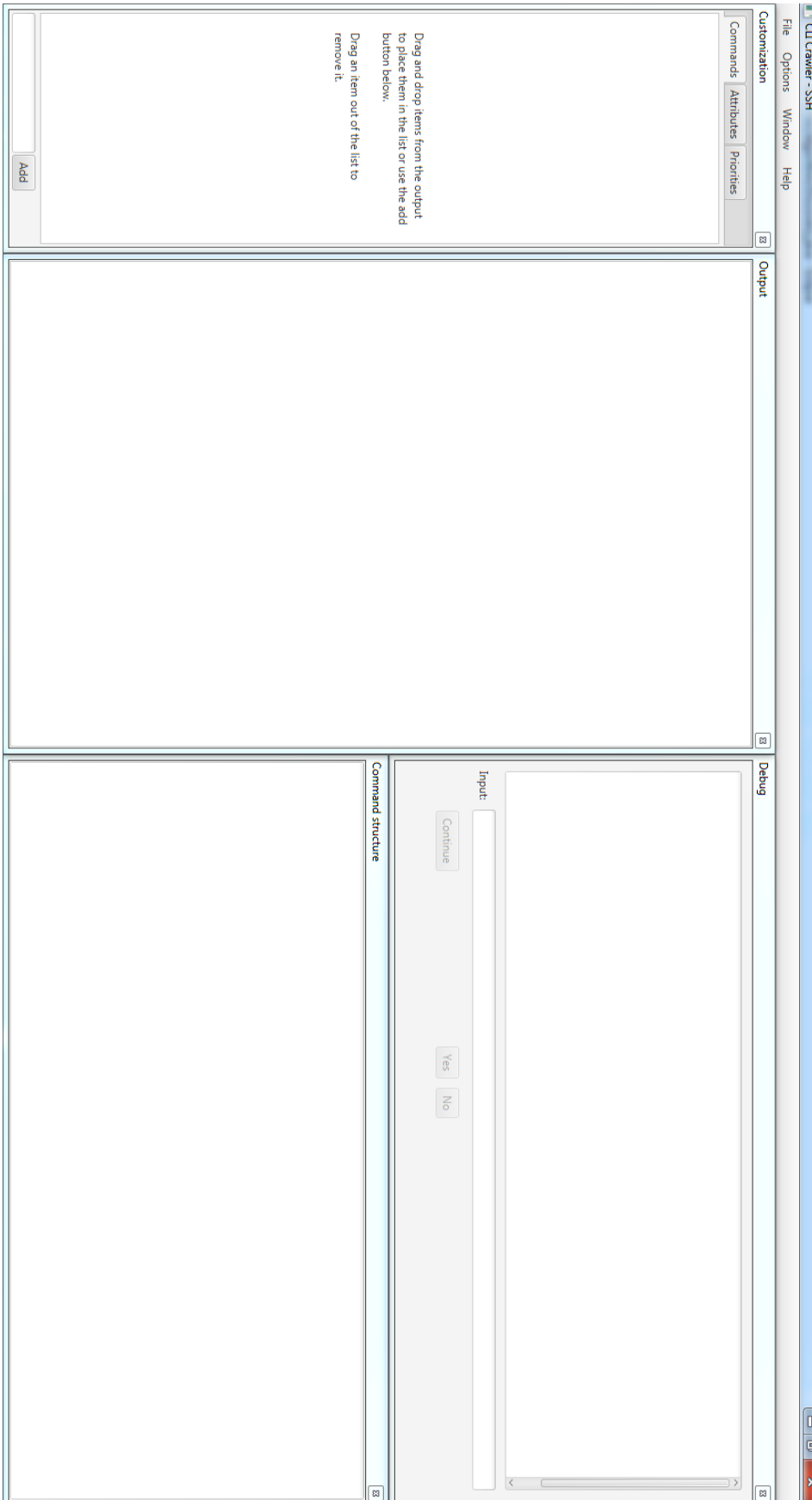
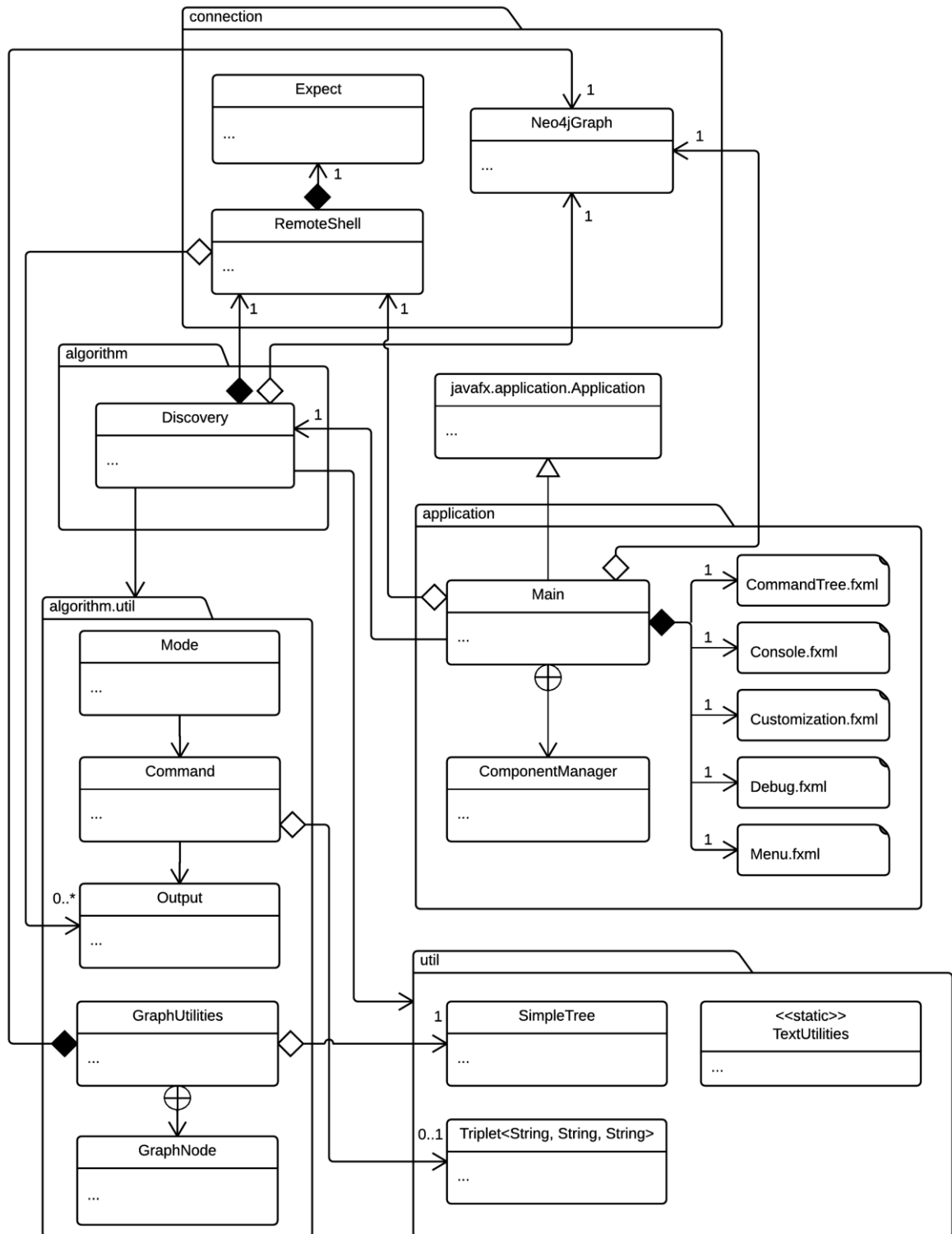


Figure showing the current GUI of CLI Crawler using JavaFX as GUI toolkit.

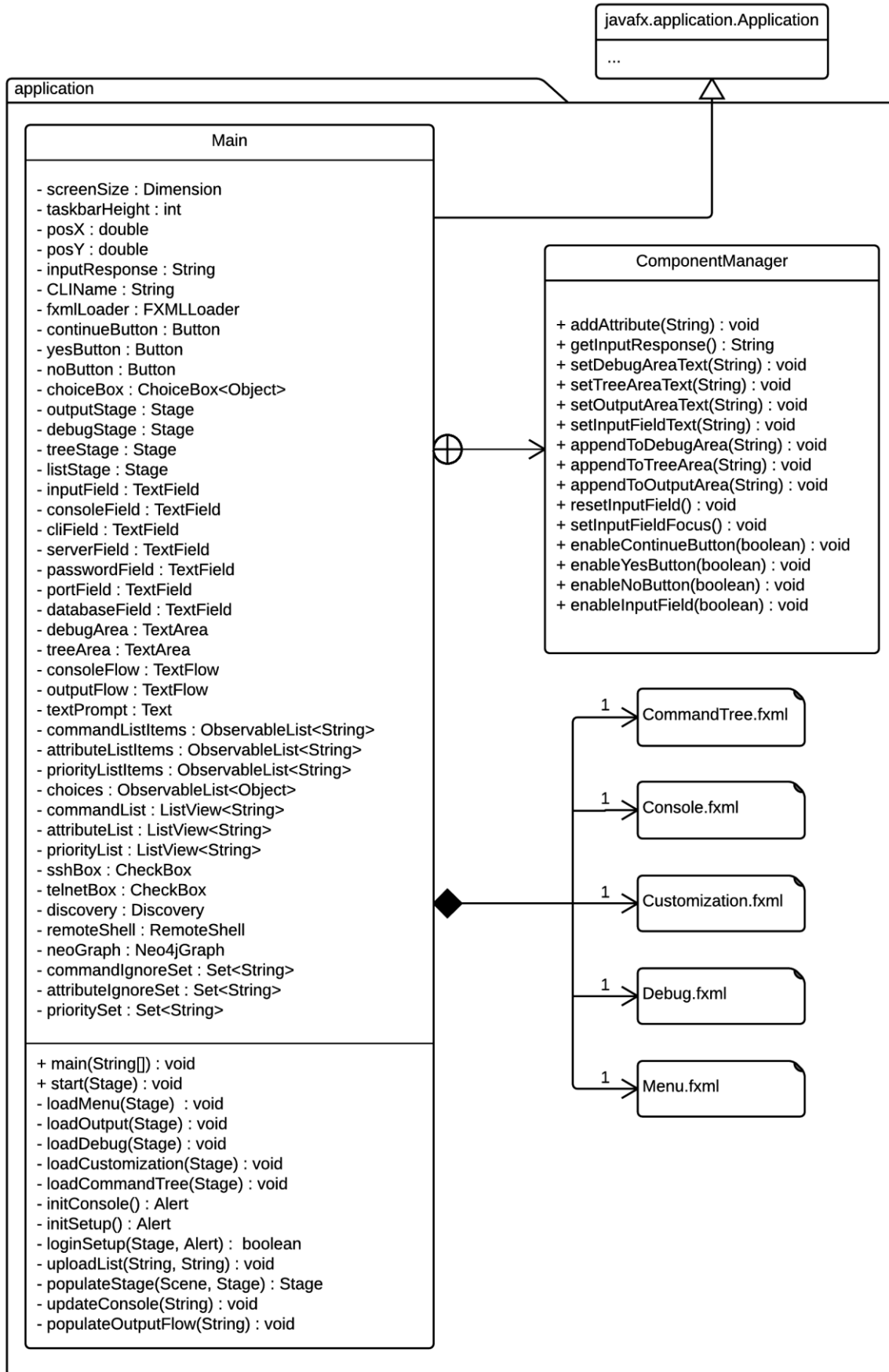
Appendix C – Diagrams, charts and figures

Appendix C.1



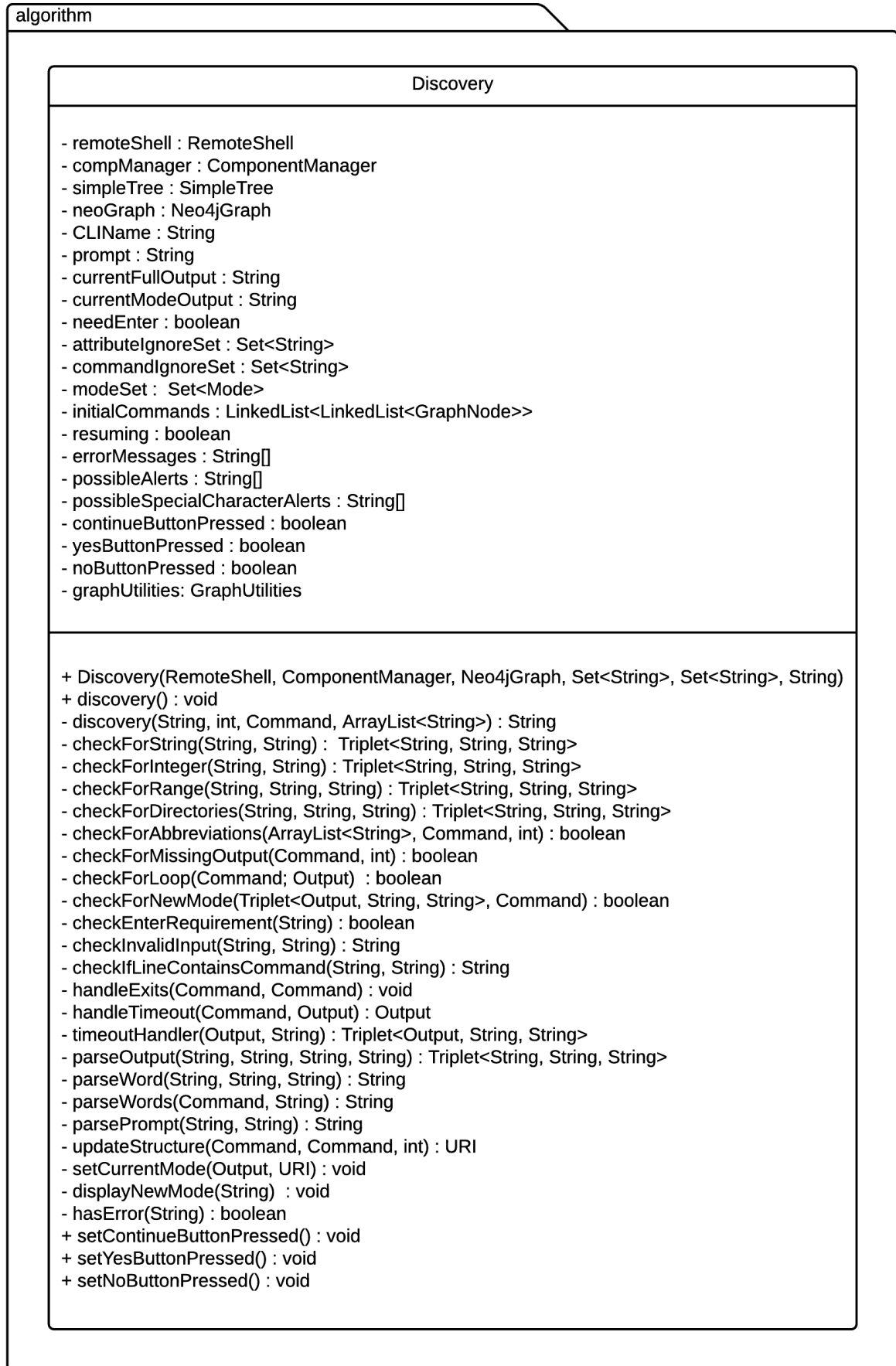
UML diagram representing CLI Crawler as an entity and showing the relationships between the different classes and packages.

Appendix C.2



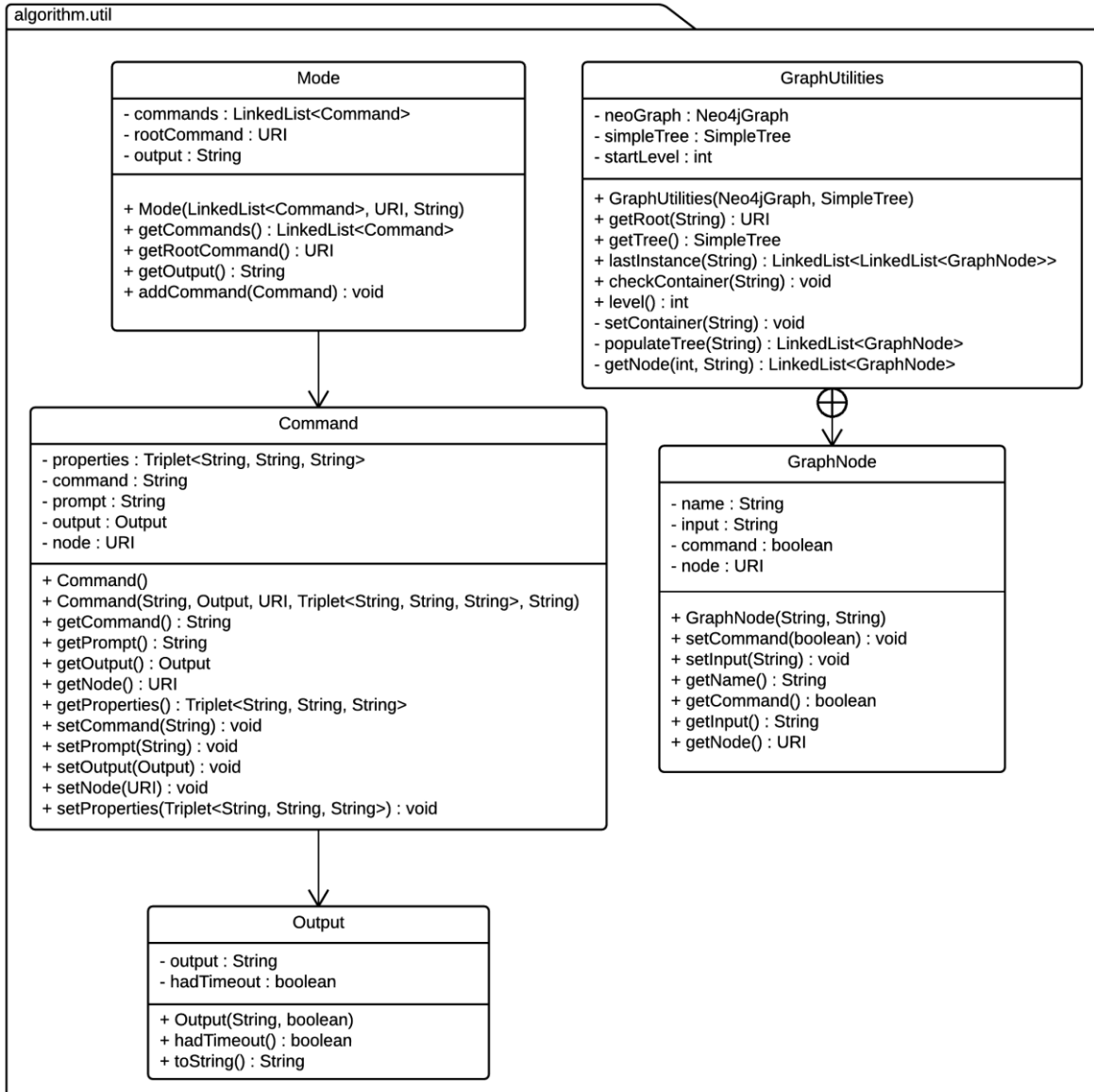
UML diagram representing the application package of CLI Crawler.

Appendix C.3



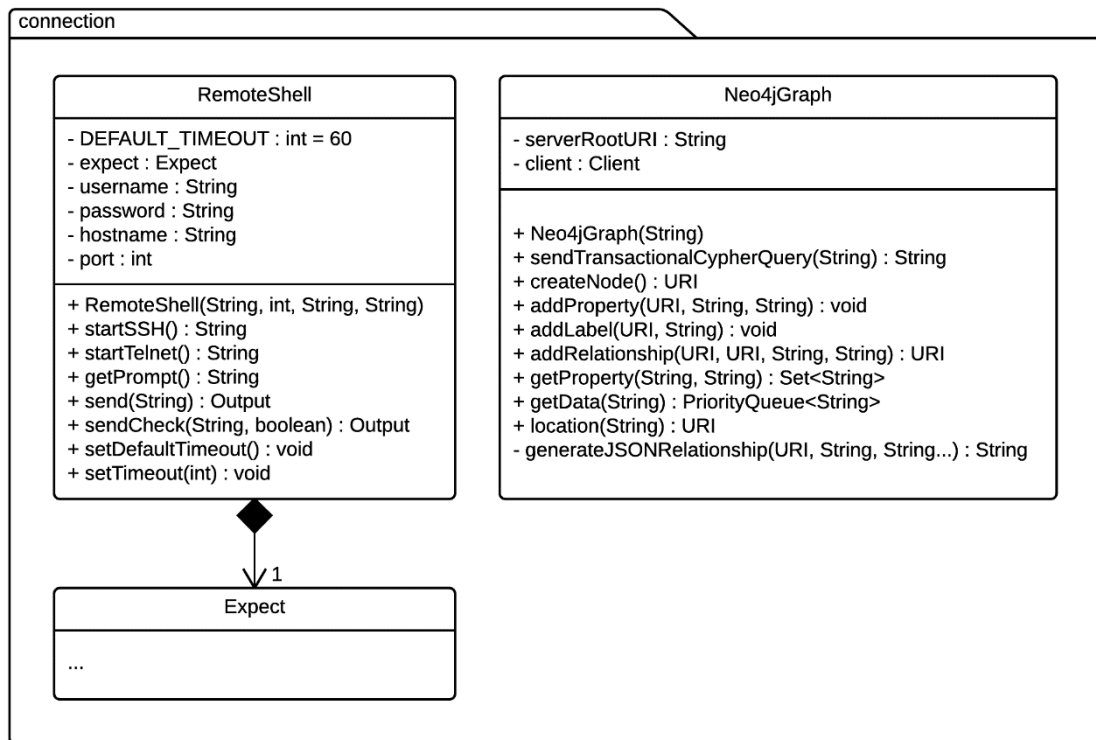
UML diagram representing the algorithm package of CLI Crawler.

Appendix C.4



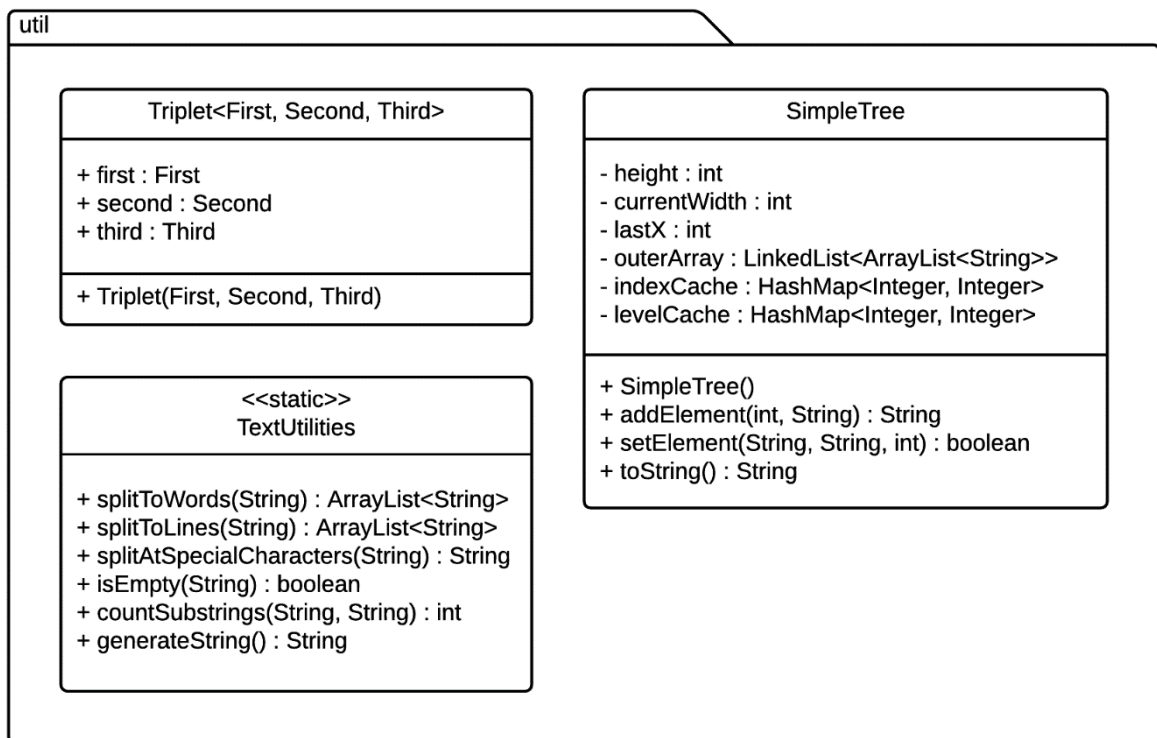
UML diagram representing the algorithm.util package of CLI Crawler.

Appendix C.5



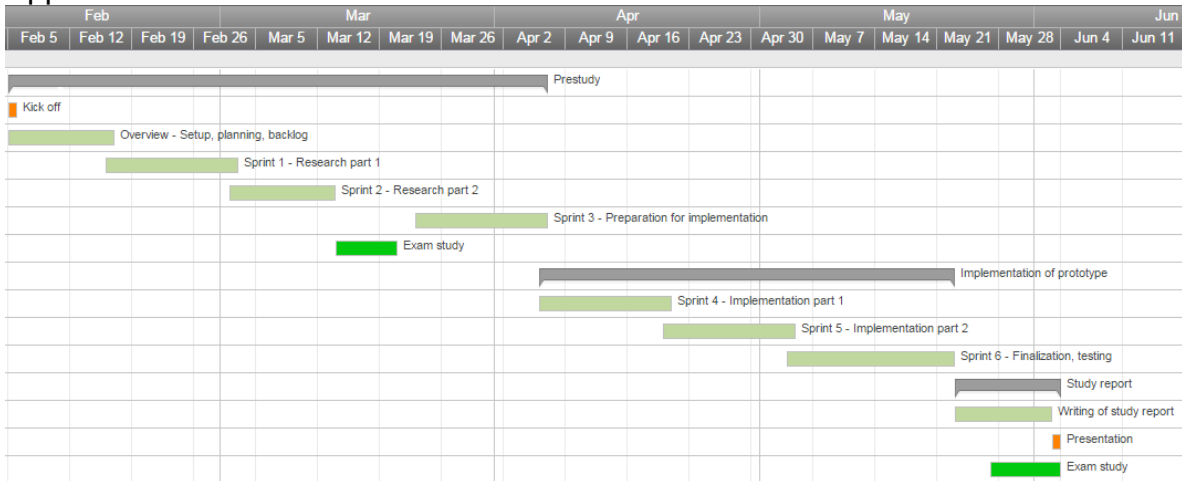
UML diagram representing the connection package of CLI Crawler.

Appendix C.6



UML diagram representing the util package of CLI Crawler.

Appendix C.7



Gantt chart showing the initial project plan.

Appendix D – User manual

User manual for CLI Crawler

Introduction

CLI Crawler is a very early build of a CLI discovery program. It will discover a certain CLI's command structure and store it in a Neo4j database while showing the hierarchical structure of the CLI in real-time during the discovery process.

This document will give an overview of how to use CLI Crawler.

Recommendations

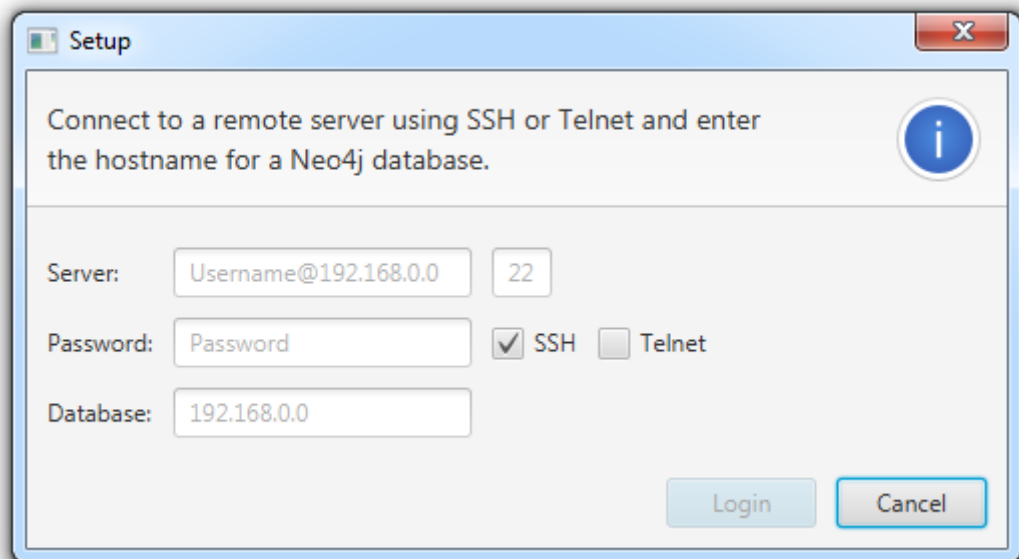
In order to use CLI Crawler properly, the following is recommended:

1. Having Java Runtime Environment 1.8.0_40 or higher installed.
2. Running CLI Crawler on a Windows or Linux platform.

CLI Crawler has not been tested with other Java Runtime Environments nor on any other operating systems and therefore may not work properly if the recommendations are not followed.

Starting a new CLI discovery process

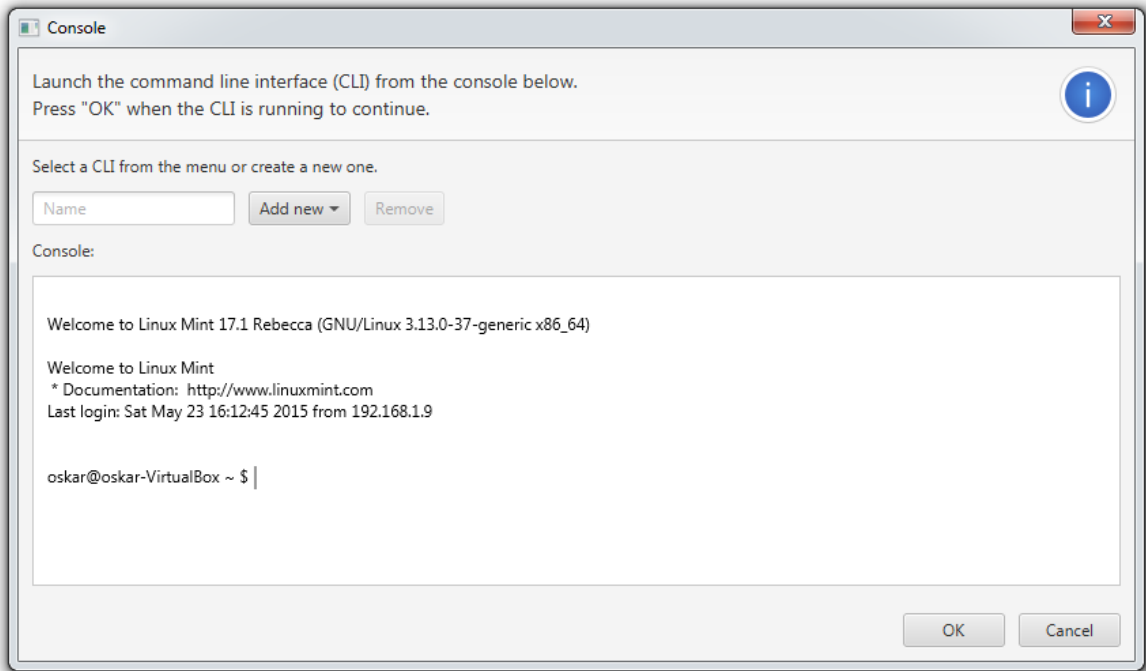
1. Run the CLI_Crawler.jar file.
2. A setup window will be shown as seen below.



- a. Enter the hostname, user, port and password of the remote system.
- b. Select whether the remote system is hosted by SSH or Telnet connection.
- c. Enter the hostname for the Neo4j database.
- d. Press Login to continue.

Note: If the connection is unsuccessful then a pop-up window will be shown and thereafter it will be possible to edit the connection information and reconnect.

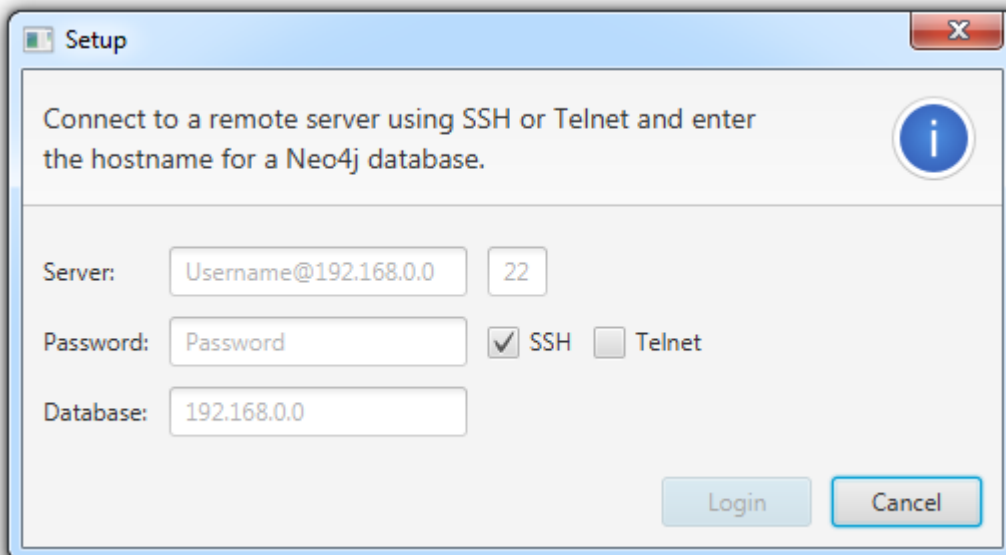
3. A console window will be shown as seen below.



- Enter the name of the CLI to be discovered in the input field.
- Enter the CLI in the console field by sending regular commands.
- Press OK to start the discovery process.

Resuming a CLI discovery process

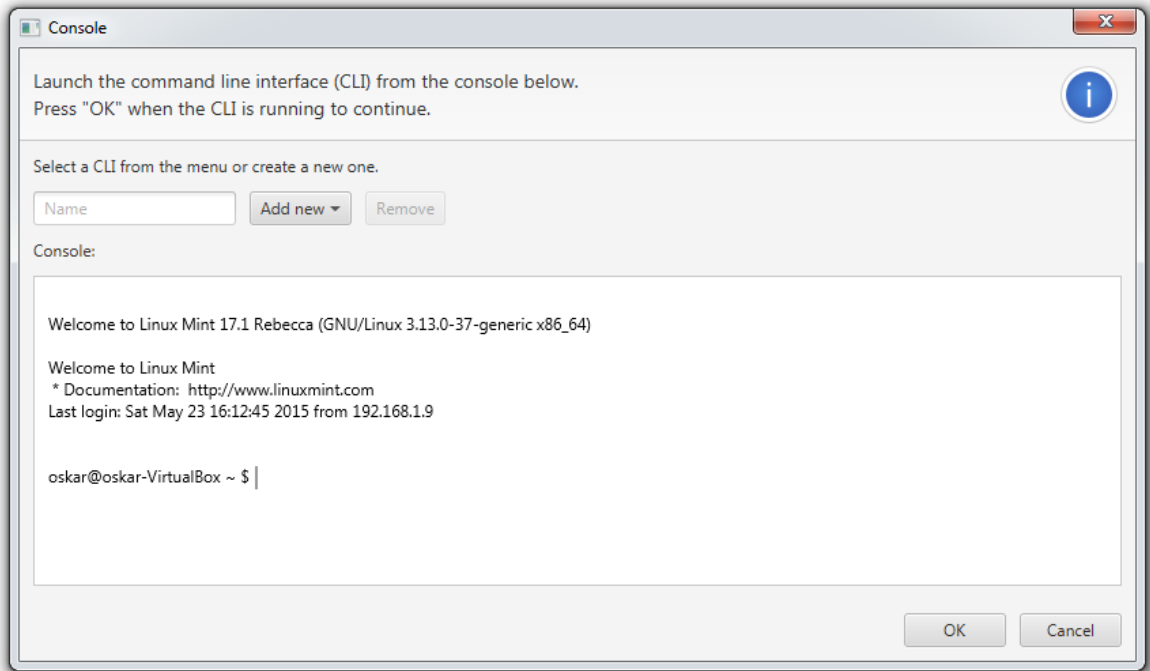
- Run the CLI_Crawler.jar file.
- A setup window will be shown as seen below.



- Enter the hostname, user, port and password of the remote system.
- Select whether the remote system is hosted by SSH or Telnet connection.
- Enter the hostname for the Neo4j database.
- Press Login to continue.

Note: If the connection is unsuccessful then a pop-up window will be shown and thereafter it will be possible to edit the connection information and reconnect.

3. A console window will be shown as seen below.

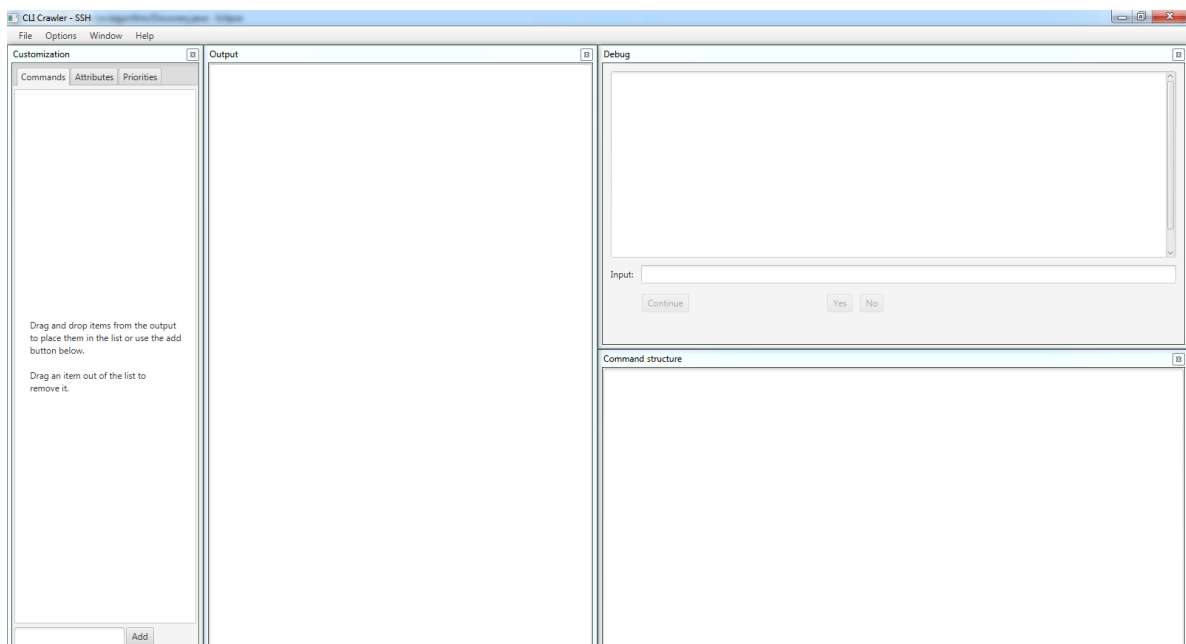


- Choose the CLI to be further discovered from the drop-down menu next to the name input field.
- Enter the CLI in the console field by sending regular commands.
- Press OK to restart the discovery process.

Note: Even when resuming a CLI process, the CLI has to be entered manually. This is due to the possibility of the CLI having a different path than in the previous discovery.

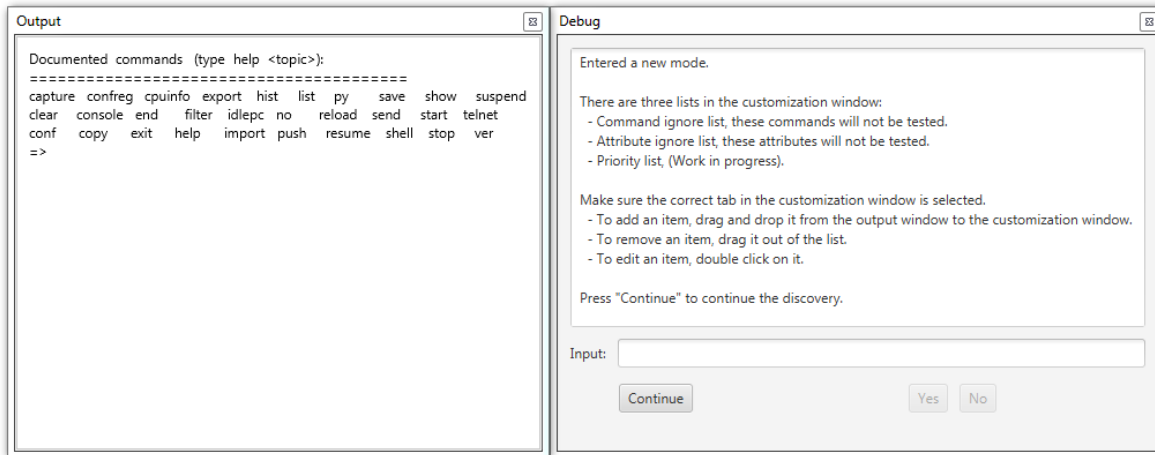
Managing CLI Crawler during a discovery process

The layout of the GUI is designed to be customizable so that each window can be resized, repositioned or even closed. The default layout and window positioning is shown below.



During a discovery process a tree representing the CLI's command structure will be printed in real-time in the command structure window.

It is also possible that user interaction will be needed during a discovery process. By default the output and debug windows will display a text saying "Running" if the discovery process is working by itself. These will however change if user interaction is needed. Below is an example of the output and debug windows when user interaction is needed – in this case when a new mode is discovered.



As seen in the screenshot above, when user interaction is needed instructions are shown in the debug window in order to know what to do since there may be different types of user interaction.

During a discovery process it is also possible to customize the lists in the customization window, how to do this will be explained in the next section of the user manual.

Customizing ignore lists

There are three lists in the customization window – one for each tab.

1. Command ignore list – commands in this list will not be tested during discovery.
2. Attribute ignore list – attributes in this list will not be tested during discovery.
3. Priority list – a list of execution order during discovery. (*Note: not yet implemented*)

The list may be edited in three ways;

1. To add an item – drag-and-drop it from the output window to the customization window or simply type in the item in the input field and press the Add button.
2. To remove an item – drag it out of the list.
3. To edit an item – double click on the item, edit it and then press enter.