BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

RPM PACKAGE QUERY RESOLVER

DOTAZOVÁNÍ NAD RPM BALÍKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ KORBAŘ**
AUTOR PRÁCE

SUPERVISOR **Ing. MIROSLAV HRONČOK,**
VEDOUCÍ PRÁCE

BRNO 2021

Abstract

The goal of this thesis is to create a tool allowing effective retrieval of data about RPM packages and perform queries both about their data and the relations which exist between them. The tool has to be able to outperform the speed of currently existing tools and allow easy extension for the preservation of more data or relations. Another required feature is a visualization of results according to user settings or providing of results in machine-readable format.

Abstrakt

Cílem této práce je vytvořít nástroj umožňující efektivně získávat data o RPM balících a dotazovat se jak na data jednotlivých balíků tak na vztahy, které mezi nimi existují. Nástroj musí být schopen předčít dosavadní rychlost dotazování existujících nástrojů a umožňovat snadné rozšíření o ukládání dalších dat nebo vztahů. Další z požadovaných funkcí je vizualizace výsledků podle nastavení uživatele nebo poskytnutí výsledku v strojově zpracovatelném formátu.

Keywords

Packages Queries Python RPM

Klíčová slova

Balíky Dotazování Python RPM

Reference

KORBAŘ, Tomáš. *RPM Package Query Resolver*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Miroslav Hrončok,

Rozšířený abstrakt

RPM packages are compressed archives containing software and metadata about their contents. When a user wants to install software with RPM package manager, DNF for example, DNF looks through its configured repositories and locates package which provides a function that the user requires. RPM package in its metadata contains information about what package provides, for example, library, and list of packages which are required on the host system for software to be working properly.

As the distribution grows bigger and more complex, so do its RPM repositories and more maintenance is required to keep distribution in a good state and useful to its users. This directly involves the cooperation of developers who maintain particular pieces of distribution and their ability to access information quickly.

While the most important information as dependencies and lists of files can be accessed in a matter of seconds through DNF alone, information that involves data that is not necessary for the package to be functional or is useful mainly to developers, often has to be searched for manually or through slow scripting. This fact leads us to the goal of this thesis and that is to invent a tool which will be able to quickly retrieve any data about packages, save this data in a cache and allow us to ask any question that we can think of.

An example of this is a situation that frequently occurs during the development and maintenance of every Fedora release. Fedora is maintained by the community and thus it is no surprise when some of the members have no longer time to maintain packages owned by them. When this happens, the maintainer can either pass the ownership to secondary maintainers or if there are none, orphan the package. An orphaned package is kept in official repositories for another 8 weeks and then is removed if no maintainer acquires it. Problem is that packages that depend on it, can not be installed. Normally maintainer of the dependent package steps up and takes ownership of the dependency but first, he must know that such an event occurred. Maintainers can be contacted but first, we must know who should we contact. This is the moment when the user has to individually query repositories about dependent packages and has to connect them with appropriate developers. An operation like this took tens of minutes with the current state of DNF API and scripting. With proper caching and optimization, it could be possible to cut this time down to seconds.

Another problem that this thesis is supposed to solve is resolving complicated queries. For example to assess how many frequently used packages will be affected by the removal of some library. This requires that packages are connected with counts of their downloads and we choose only those packages that meet the specified threshold. This way we can prioritize the maintenance of heavily used libraries and fulfill the true requirements of distribution users.

The first of the greatest challenges of this project is to find an efficient data structure to store package data in such a way that queries will be able to walk them quickly over and use algorithms that can quickly process relations. Another requirement is that this structure is well understandable and in the best case serializable to a file that we can later use as a cache. The cache should allow the user to not retrieve and build structure every time the project is run but increase the speed of query by avoiding this time-consuming operation.

The second is to choose how should the queries be specified. It has to be complex enough to be able to express complicated queries but easy to learn and read, so users do not have to spend a long time studying the tool before they can use it for something useful. At the same time, it is very important to assert that the queries will be optimized in such

a way that there will not be any unnecessary evaluations to accelerate the whole execution process.

Foundations on which any real-life usable tool stands is accurate documentation. The project has to be properly documented so users can find every information they need and are not slowed down by asking developers or studying how the project behaves in different situations.

Last but not least requirement is the need for the tool to be properly tested. Combination of unit and functional testing will be needed to ensure projects stability and to make long time maintenance easy. The developer has a lot of choices when it comes to choosing testing technology but since language of the project is Python it is only fitting that verified framework with a great community such as Pytest is used.

The project has a real-life use case scenario that came up during maintenance of packages in Fedora and RHEL distributions which will allow the author and lead to identify whether it has fulfilled its purpose or not. Solve all these problems is the final goal of this thesis which is meant to prove that the creation of such a powerful tool is possible and it can accelerate the work of package maintainers while making the dependent distribution more stable and secure.

RPM Package Query Resolver

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Miroslav Hrončok The supplementary information was provided by Mr. Adam Rogalewicz I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tomáš Korbař
April 7, 2022

Acknowledgements

I would like to thank Mr. Miroslav Hrončok for his help during making of this thesis and Mr. Adam Rogalewicz for watching that the project meets requirements for it to be usable.

Contents

Chapter 1

Introduction

Linux distributions are built upon packages that provide software to their users. It is these packages that make particular distribution different or similar to another one. All these packages require maintenance, in this case, the distribution is developed commercially, as well as is developed by the community. The fact that the whole system is stable and comfortable to use is a result of the good work of maintainers who work together to ensure that all packages are working and their requirements are met.

Efficient maintenance requires accurate information, which maintainers get from multiple sources and use many tools to retrieve. For example, which packages depend on their packages, and thus which maintainers should they communicate with when encountering disturbing changes? Often the information is not easy to acquire and slows down developers by forcing them to manually search through different places. To solve this problem, a new tool that can store and filter any information about packages is required as no other exists at the moment.

RPQR is an originally proposed tool that is supposed to make maintainers' life easier by allowing them to describe how to acquire the data only once and then be able to retrieve it on demand. It is flexible enough to store any new kind of data and to search packages based on a combination of any of them while also providing the option to accelerate queries by making specialized commands. RPQR also lets users build a cache which makes further queries faster and thus saves time while doing everyday work.

The tool can be directly used by other projects through provided API. It can also be used by a user as any other command-line project. Users are also able to visualize their results for a faster understanding of the results.

Chapter 2

Theory and current state

While crucial information about an RPM package is stored directly inside it within the header section, additional information as who maintains it or which bugs are currently known has to be searched in external sources of information. This chapter describes RPM packages and technologies which currently exist for working with them. Then we continue with a description of other subjects which are needed to successfully design and implement the RPQR project.

2.1 RPM package

RPM packages have their file format[5]. It is composed of four parts with their specific purposes. The parts are (i)lead, (ii)signature, (iii)header and (iv)archive. Here are described and explained all parts relevant to the RPQR project.

The lead

The lead is the first part of the RPM package. It contains the magic number and version of the RPM file format. It also contains whether the package type is binary or source and other information relevant to a system using it. The difference between binary and source packages is that the source package contains source code from which software can be built or the whole downloaded project, while the binary package contains the actual software. The lead is no longer used internally by RPM because of its inflexibility and is noted here only for completeness of file format description.

The signature

The signature is allowing package integrity and optionally authenticity to be verified. It holds little purpose for RPQR but it is important because DNF uses it and RPQR is using DNF API.

The header

The header is the most interesting part from the perspective of the RPQR project because it contains detailed crucial information about the package. It is composed of tags that describe different aspects of the bundled software. Examples of these tags can be *RPMTAG_VERSION* specifying a version of the package or *RPMTAG_RELEASE* which specifies what release of this version this is. The header is parsed by software that is making the metadata structure of RPM repositories and this structure is then used by the DNF package manager to find appropriate packages that the user needs.

```
00001198 8e ad e8 01 00 00 00 00 00 00 00 3e 00 00 0f dd |...........>....|
```

The first 16 bytes of the header part are describing the attributes of this header. Three bytes are the magic number identifying the header, one byte says that the header conforms to version 1 of the specification. Four other bytes are reserved. Then there is a count of entries stored in this header (00 00 00 3e to decimal is 62). The last four bytes mean how many bytes are stored in this structure (00 00 0f dd to decimal is 4061).

```
000011c8 00 00 03 e8 00 00 00 06 00 00 00 02 00 00 00 01 |................|
```

For the best example, we will describe the name tag. 00 00 03 e8 identifies the presence of a name tag and 00 00 00 06 says that value is a string. 00 00 00 02 means that the value is located 2 bytes after the start of the store and 00 00 00 01 indicates that there is just one value, which is the only allowed possibility for string value stored in the header.

The store is composed of values after each other that are distinguished only by their respective offsets.

The archive

The archive is a set of files and folders compressed with a compression algorithm. Its integrity can be verified with the signature specified.

2.2 RPM repository

RPM repositories are directory structures that contain RPM packages and metadata which are needed to quickly locate packages that the user needs. Metadata are created by *createrepo* [7] utility and accessed by the DNF package manager.

Repodata

Every RPM repository contains a folder *repodata* which contains data about the contents of the repository. There is a file *repomd.xml* containing XML structure indicating where should package manager look for databases with information about packages.

Important databases:

- **primary** database which specifies all crucial information about each package as of version, description, or file list.
- **other** database which contains other less important information e.g. changelog of a package

2.3 Package managers

Working with RPM packages can be done with multiple tools. Their general responsibility is to recognize package dependencies and being able to install or uninstall software contained in the package. Most frequently used are RPM package manager and DNF package manager (successor of the old YUM package manager).

RPM package manager

RPM package manager supports more low-level operations with packages than DNF does [6]. It allows to build the source of a project according to specfile and create distributable packages. More of the important operations are also reading the metadata and verification of installed software, in case it is not working properly. Installation of dependencies would have to be handled by a user manually so the rpm utility is not often used by end-users of the systems.

YUM package manager

Yum package manager is historically the first manager that allowed easy downloading of packages from remote repositories and handling their dependencies [11]. It is currently deprecated and has been replaced by DNF. Reasons for deprecation and replacement were that YUM was not properly documented, it was not ready for a switch to python3, and the algorithm for dependency resolution was not strong enough to handle all problems that withstand in modern RPM-based Linux distributions.

DNF package manager

DNF package manager is a successor of the older YUM package manager [1]. It allows users to install and remove software on the system comfortably by handling all of the operations that are needed to retrieve package dependencies and resolve any possible conflicts. A very often used feature is also system upgrades, when DNF can migrate a system from old versions of distribution to new ones. A very important fact that needs to be stated is that DNF provides python API which can be used by other projects to retrieve metadata from repositories and distinguish them.

DNF is the only tool that is currently able to query repositories for metadata that are specified in packages. An example of a frequently used query is:

$ dnf repoquery --whatprovides /usr/bin/bash

This command issues that DNF should execute command *repoquery* filtering by tag *provides* and find all packages that provide file */usr/bin/bash*. DNF can search for packages based on attributes that are supplied within the package, but it is not able to retrieve additional information or query based on complex relationships. It is not its job to resolve more difficult queries and it would be wrong to force it to by extending its capabilities.

2.4 DNF API

DNF provides python API through which developers can interact with repositories and retrieve information. At first, an instance of *Base* class has to be created and then specify repositories from which metadata should be retrieved. Call to *fill_sack* method after that will load the metadata and API can then execute queries that the DNF tool supports.

Example of how is dnf API used in RPQR project:

```
base = dnf.Base()
for (name, url) in self.repositories:
    base.repos.add_new_repo(name, base.conf, baseurl=[url])
base.fill_sack(load_system_repo=False, load_available_repos=True)
return base.sack.query().available()
```

2.5 Storing techniques and query language

As was stated before, it is crucial to choose the right technologies to store package metadata in such a format that they can be read by a human reader while also easily parsable and serializable. This section will describe possible formats. Another part will explain the existing query languages that could be used for RPQR queries and their pros and cons in the context of describing package metadata.

Data structures

While RPM repositories store metadata as a list in XML format or *sqlite* database, for use cases that are oriented about relationships between packages list does not have to be appropriate data structure for internal representation of package metadata.

- List

  Pros:

  Easy to work with Python

  Simple algorithms to process its members

  Cons:

  Bad handling of relationship representation

- Dictionary

  Pros:

  Faster accessing of members

  Cons:

  Forcing packages to be identified by the same attribute

- Graph

  Pros:

  Great representation of relationships between packages

  Fast algorithms for searching and filtration

  Cons:

  More complex algorithms for processing of nodes

  To filter packages according to attributes, dictionary or list is still needed since there is no package that we could consider as the proper root of the graph.

Data formats

An appropriate data format needs to be chosen for storing data. Currently, many massively used formats could be suitable for the RPQR use case. The data format should be chosen accordingly to how much readability it can provide for a human developer and whether it can be used within versioning repositories such as git or mercury.

XML

Extensible markup language[9] is used by *repocreate* utility which is parsing package metadata and creating their collections for package managers. It is natively supported by Python and relatively easy to read. XML uses tags to distinguish individual elements of serialized data. Its advantage is that it supports various encodings and even can contain comments so some things in serialized data could have additional explanations when needed.

Example of XML data:

```
<?xml version="1.0" encoding="UTF-8"?>
<element>
    <innerElement>
      Example text
    </innerElement>
    <!-- Explanation comment -->
</element>
```

JSON

JavaScript Object Notation[3] is a widely used format for data serialization which represents objects with pairs of named attributes and their values. One of the big advantages is that it also natively supports arrays and consists of minimal syntax which allows most data to be stored and transferred with less required space. Unfortunately, there is no support for comments but the readability of JSON data is generally good so they are not needed in most cases.

Example of JSON data:

```
{"Element":{
  "InnerElement": "Example text"
  }
}
```

YAML

YAML[10] Ain't Markup Language is a data format used by many applications for configuration and data transfer. YAML used JSON as a basis for its version 1.2 and it is accepting JSON as its subset. The interesting about this format is that unlike JSON it supports comments and extensible data types. Strings in YAML can be also specified without the starting and ending quotation marks. Individual attributes of objects are distinguished by name and indentation by a style that is similar to Python. While YAML data sets are generally smaller than JSON, the number of additional syntax features makes their parsers more complex and thus it inevitably takes more time to load them.

Example of YAML data:

```
element:
  InnerElement: Example text
```

Pickle

For completeness here is mentioned even Python pickle format[4] for object serialization. Because it is binary it can be parsed more quickly and support the additional acceleration of RPQR execution. There is an issue with the execution of arbitrary code when parsing pickle structures which does not occur in previously mentioned formats. Unlike the previous formats, it is not human-readable and thus unfortunately not appropriate to be used for package data structures that should be accessible by different tools. There is no example because it would not make sense to show binary data.

Query languages

For purposes of the RPQR project, there needs to be a specification on how to describe queries. Currently, there are many approaches. In this section, there will be a description of some of them and their features which could prove useful for selecting packages and their attributes.

SQL

Structured Query Language[8] is a domain-specific language that is widely used to interact with relational databases. SQL can select records based upon their attributes and relations but is not capable of describing complex recursive queries about graphs. Another caveat is that for a Python application, using standard Python libraries, to be able to use SQL, it would need to hold an instance of *sqlite* database in memory and that could prove to be an unnecessary overhead that could slow execution down.

Example of SQL query: *SELECT * FROM table WHERE id = 3*

GraphQL

GraphQL[2] is an open-source query language that allows developers to implement their interpretation of individual query parameters. It is used in REST APIs to allow client applications to retrieve data effectively from a server without it having to transfer any unnecessary data. Its flexibility is a great advantage, but queries are not as readable as they would be in SQL language.

Cypher query language

Cypher query language is an implementation of *opencypher* specification. It is meant to work with the neo4j graph database and is developed for such a purpose. For an application to be able to get the advantage of cypher, it needs to use the neo4j database which can result in too big an overhead for utilities designed with one specific purpose in mind.

Example of Cypher query language:

MATCH (peter: employee name: 'Peter Parker') RETURN peter

Example of GraphQL query:

```
{
  table(where: {id: {_eq: 3}}) {
    id
    name
    age
  }
}
```

Domain specific language on Python

Creating query language is always an option and it holds enormous power in the ability to bend the language to the specific purpose that the RPQR project needs. Problem is that developing and maintaining language takes time and energy. For a language to be functional, RPQR would need to implement its components like scanner, parser, and interpreter. In the essence, a domain-specific language for RPQR would need to be relatively simple. There is a requirement to interpret statements which result is always a set of nodes that represent packages. These statements consist of commands that take values and other statements as arguments and operators which realize basic set operations as is union or intersection.

Example of how RPQR language could look:

WHATDEPENDSON('libyang', 3) & WHATDEPENDSON('libgcc', 3)

Components that are needed for interpretation of RPQR language:

- **Scanner**

  The scanner is used to convert the source text of the language to tokens for further processing by the parser. Its crucial part is a finite state machine that reacts to characters in input and recognizes lexical tokens. The scanner is also able to tell a user when some lexical error occurs and the query needs to be changed for it to be valid.

- **Parser**

  Parser consumes tokens from the scanner and handles the creation of abstract syntactic tree or some other internal representation of source text on input. The parser can recognize syntactic errors which occur during parsing and optionally inform the user about them. There are multiple techniques for syntactic analysis as Top-Down Parsing or Bottom-Up Parsing, which are algorithms how to recognize language units on input. Both of these techniques use models for context-less languages such as formal grammatic. Formal grammatic is a list of rules which are used to check whether the input is written in a language or not.

- **Interpreter**

  RPQR does not need to translate the query into some other form, it needs to perform it. That is the reason why the last part of the RPQR language would be its interpreter. The interpreter inside of RPQR would need to be flexible enough for it to be able to accept new commands for searching for packages. Another important thing is using optimizations such as short evaluation to make searching for packages as fast as possible.

2.6 Caching

Since one of the most time-consuming actions of the current approach to queries about RPM packages is network communication and transfer of metadata, it is crucial to download all metadata at once at the start of execution to not need any further downloads. This can be ensured through DNF API by executing a query to list all packages that are available to install from specified repositories. DNF package manager uses a very similar approach by downloading all metadata to local storage and updating it only when the user forces it to or the cache expires.

The question of metadata expiration needs to be handled by the RPQR itself too.

Approach, when metadata is not updated unless the user wants to do so, could save time for checking of the repository, but the user would be responsible for the consistency of metadata and repository state, which could prove problematic.

RPQR could stick with the same approach as the DNF one. Rebuild metadata when they expire. The problem is that building internal structure and rebuilding cache could be a very time-consuming the operation, maybe even in a matter of minutes.

The third and maybe the most proper approach is to set the expiration time of metadata to some longer period. After such a period the time for the rebuilding of the cache will not be so important. Also if no change occurred then it is pointless to rebuild the data and it would be highly useful to rebuild only parts of internal data structures which do no longer correspond with the actual state of RPM repositories supplied in configuration.

2.7 Configuration

The RPQR tool will need multiple options for it to work properly and accordingly to users' notions. There are multiple ways how to supply such configuration to the tool. One of the most common ones is to supply configuration by command-line options. While this is easy to implement and Python offers native support for it, this approach could prove to be painful for the user when overused. For example, six or more command-line options would be difficult to track. That can be solved by providing the user with the means to set mostly static options through the configuration file.

With configuration file withstands more choices.

- Format of the configuration file. Multiple formats can be used. JSON or YAML is probably the most appropriate ones. Their description was stated in previous sections.
- What options should configuration include? Configuration should include only options which do not change often and thus do not force the user to change the file frequently.

Chapter 3

Research

With good knowledge about the current state of utilities and technologies, this chapter can explore the best approach to resolving complex queries about RPM packages. In each section, there is described a particular approach was chosen as the best solution to an individual problem.

3.1 Project structure

Python project is most often divided into folders that contain logically related classes. Classes that have fewer dependencies are located deeper in a directory structure. So entire implementation of the logic of the project has one root folder. Another folder is meant for executable binaries or scripts that are supposed to be installed in the path of the users' system. The last important folder is the folder containing tests. There are multiple ways to store projects' tests but a dedicated folder seems to be the cleanest and tests seeking utilities have an easier time finding tests organized like that.

Illustration of proposed structure:

```
bin
bin/script
project
project/example_module
test
test/example_module
```

3.2 Retrieval of information from repository

While there are approaches that would allow individual retrieval of metadata from repositories, such as custom downloading of XMLs and database archives, there is no reason for that. DNF provides API that allows an application to use its already implemented downloading of metadata. The best way to use it for this purpose is to create a query that matches every package accessible through configured repositories.

3.3 Customization and modification of functionalities

RPQR project needs to be able to adapt to changing demands on queries and the most simple way to achieve that is to create a system of loading plugins in a form of python modules. The Python script cannot dynamically load another module but because of the very high level of introspection that Python provides, it is possible to achieve something very similar. When the plugin upholds certain defined rules, such as that class for load is named the same as the file, then it is possible to easily create an efficient algorithm for searching and importing accessible plugins.

Illustration of plugin importing:

- Gather all directories for inspection from configuration or use hardcoded paths
- For each folder, walk over files and check if they fulfill naming rules
- Try to import classes by names devised from file names

Naming conventions for files containing plugins:

- Filename can not start with an underscore. Python uses ___init___ files in directories and we need to omit them. There also has to be a possibility to add supplementary files without importing them
- Class that should be imported has to have the same name as the file. This way we can avoid implementing unnecessary overhead by looking through the module and searching for classes by some more rules

A plugin will contain one main class that can define how to retrieve data that it needs and commands that can be used to filter packages by this attribute or relationship. Enforcing good structure will be done by providing base classes that plugins need to extend.

3.4 Internal structure of data representation

For RPQR to be able to effectively walk through packages and filter them by attributes and relationships, there has to be an appropriate way to access them as quickly as possible. That is why by nature, a graph is the best way. Using graphs will allow the RPQR project to use graph algorithms such as breadth-first search or depth-first search. Python itself does not have built-in graph support, so the RPQR project can either contain its implementation or use a library.

Networkx seems to be a very quick and easy-to-use implementation of graph abstract data type which is also capable of rendering a graph with multiple algorithms when needed. Another very useful feature is that networkx can save the graph to JSON formatted string and load it again from this string.

Example of building graph with Networkx:

```
import networkx as netx
graph = netx.Graph()
graph.add_node(1)
graph.add_node(2)
graph.add_edge(1,2)
```

Configuration

Some options are uncomfortable to enter through the command line repeatedly and because of that should be stored in a persistent file. The structure of the configuration file can have many forms but Python has a built-in module named configparser which defines a human-readable format appropriate for the RPQR project. Configparser uses a section to divide configuration into logically related blocks, there will be the main section for global options like URLs of repositories. Each plugin will have its section where it will be possible to disable it or provide individual information necessary for its proper function.

Example of configparser configuration file:

```
[first_section]
option1 = 1
option3 = filepath
[second_section]
option2 = 2
```

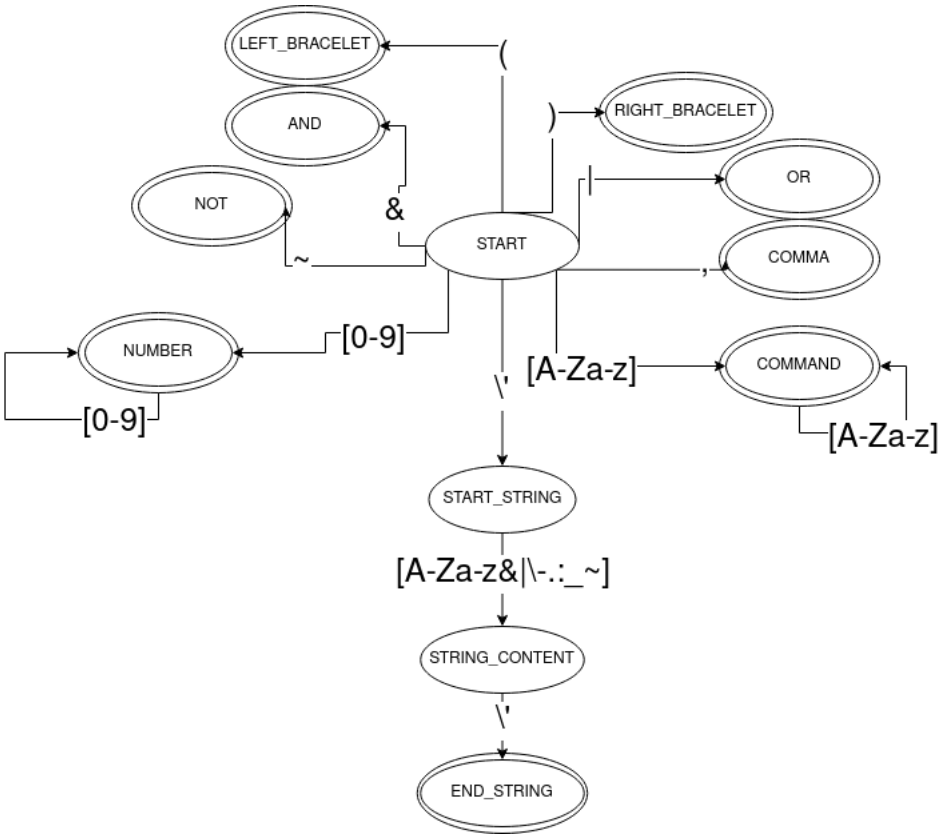
3.5 Query language design

RQPR language is by nature of its use oriented on filtering sets and thus will be constructed from statements and operations between them. This section will thoroughly describe the language and its formal description from the view of formal language theory.

Lexical analysis

RPQR will use a finite state machine for scanning tokens present in entered queries and putting them in a list that can be further processed. Each of the lexical tokens is defined by regular expression and when not recognized can be marked as invalid.

Finite state machine graph:



The lexical tokens that occur in RPQR language are:

- left bracelet (
- right bracelet)
- and operator &
- or operator |
- negation operator
- number (consisting only of numeric characters e.g 123)
- string (hyphen separated string of alphanumeric and special characters e.g 'hello')
- command (command contains only alpha characters and has to be described by a plugin e.g NAMELIKE)
- comma used mainly as a separator for command arguments,

Syntactic analysis

RPQR language syntactic analysis will be mainly precedent syntactic analysis because the language is statement-oriented. The precedent syntactic analysis uses an algorithm with a symbol stack and acts accordingly to the precedent syntactic table. This table defines what operators can be used at particular places and their respective priorities. This solves the problem of evaluation of statements but there is still the matter of command recognition and validation of argument types. Every command has to define what arguments it needs to work properly. The initial configuration of RPQR will load commands and create context less grammatic for them. Because every command has a different name and there is no need for dynamic arguments, a distinction should be straightforward and effective.

Another more problematic matter is that for RPQR language to be able to handle all necessary use cases, commands need to be able to accept the results of other commands as arguments. This is problematic since it requires a new instance of precedent syntactic analysis to parse this statement. Fortunately, this can be solved by cutting substatement out of the original statement and putting it queue of statements that have to be yet parsed.

After all these problems are solved, RPQR will have an abstract syntactic tree containing all the information that is necessary for the execution of statements e.g. commands that need to be executed first and operators located in depth accordingly to their precedence. This tree will be later processed by semantic analysis e.g. interpreter.

Precedent syntactic table used for RPQR language:

	(	)	&	|	~	$
(	<	=	<	<	<	#
)	#	>	>	>	>	>
&	<	>	>	<	<	>
|	<	>	<	>	<	>
~	<	>	>	>	<	>
$	<	#	<	<	<	X

Explanation of symbols

The algorithm which handles syntactic analysis is driven by the precedent syntactic table. It always looks at the first terminal symbol at the top of the stack and performs an operation that is specified in the table by what symbol is in input.

- < means that a special symbol marking the start of a particular statement needs to be put onto the stack and a new symbol loaded from the input
- = means only load new symbol
- > means that particular sub statements should be collapsed into one parent statement
- # means that syntactic error occurred and provided input is not a valid RQPR language statement

Semantic analysis

Semantic analysis e.g. interpreter will be an implementation of a depth-first search algorithm for processing of abstract syntactic tree provided by syntactic analysis. It is walking through the tree and putting found nodes in a stack until it finds a command or statement that can be already resolved. When the command that is defined by the accessible plugin is encountered, then the interpreter will filter loaded packages and either provide them as the final result or use them as an operand to one of the operators.

When a command is executed or a statement can be evaluated accordingly to the type of operator that it contains, part of the abstract syntactic tree related to it is marked as resolved and the temporary result is saved into the appropriate node. This means that the final result will be present as the root of the tree.

As in syntactic analysis, there is a problem with subsets used as an argument. These subsets have to be executed similarly as they were processed into the abstract syntactic tree. When encountered interpreter will stop command processing and proceed to resolve the substatement with higher priority.

3.6 RPQR language and its use

This section should provide usage examples of RPQR language and results that should be expected. RPQR statement always consists of at least one command.

Simple command

COMMAND()

This command will receive the entire graph of packages as input and will be responsible for providing a set of packages that conform to its filter. Since this command does not accept any arguments, as there are no arguments supplied, the filter is static and can not be affected by the user.

Command with arguments

ADVANCEDCOMMAND('package', 3)

Command used like this accepts two arguments which alter his behavior. The first one is a string and the second is a number. RPQR language does not consider whitespace characters, so there is no difference or problem with their presence in the query. Commands like this can have more advanced behavior and are generally more useful.

Command accepting subset as an argument

SUBSETCOMMAND(NAMELIKE('cups'), 3)

This is the most complex command that the RPQR language supports. This query will at first filter the entire graph with the command *NAMELIKE('cups')* and then supply its result to the *SUBSETCOMMAND* command. The second command will have the possibility to work with a subset and thus does not have to work with the entire graph which results in an ability to work more efficiently or perform operations with a specific context. For a better explanation of how the query could work. The first command could gather packages that contain string *cups* in their name. The second could then filter only three first by their name in alphabetical order.

Operators

Intersection

FIRSTCOMMAND() & SECONDCOMMAND()

Operator & can be used as an intersection between sets provided as outputs of two commands or sets. Packages returned by query specified like this have to be present in both the left and right set.

Join

FIRSTCOMMAND() | SECONDCOMMAND()

Operator | can be used to join sets provided by two commands or statements. It has a lower priority than intersection and means that resulting sets will contain packages that are present in either left or right set of this statement.

Negation

~FIRSTCOMMAND()

Operator ~can be used to specify that the result set of packages can contain only such packages that do not conform to conditions specified by *FIRSTCOMMAND*. It is important to keep in mind that the input of the command is always the entire graph.

Complex queries and explanation of their semantics

When in need of complex conditions and a combination of commands, it is very useful to use bracelets to force priority of evaluation and to avoid confusion between what the user expects as a result and what is the real result?

Example of bracelet use:

FIRSTCOMMAND() & SECONDCOMMAND() | THIRDCOMMAND()

Explanation of this query is: return packages that conform to conditions of *FIRSTCOMMAND* and *SECONDCOMMAND* or packages that conform to *THIRDCOMMAND*. This is caused by the priority of operators.

FIRSTCOMMAND() & (SECONDCOMMAND() | THIRDCOMMAND())

This query looks very similar but some bracelets change their meaning very significantly. Packages in result set has to conform to *FIRSTCOMMAND* and to *SECONDCOMMAND* or *THIRDCOMMAND* in the same time. Bracelets allow us to form more complex descriptions of packages that we are looking for.

3.7 Plugin architecture and its interface

Since the whole project will be written in Python which is an object-oriented language, all plugins will be inherited from a class that will provide the standard interface expected by the RPQR project. There should be an initialization method allowing the plugin to prepare helper structures and then a method responsible for inserting information into the plugins. Packages have attributes and relationships, so there will have to be two types of plugins, one inserting proper attributes to nodes and another one that will construct relationships between them. RPQR will work with attribute plugins as if they had a higher priority so relationship plugins will be able to work with already prepared attributes and there will be no unnecessary overhead.

Commands supplied by the plugin will also conform to the interface specified by their base class. Each will have to list types of arguments that they need and implement a function that contains the logic of their filtering operations. Since many commands will be working with similar graph algorithms such as depth-first search or breadth-first search, the base class will provide an optimized implementation that just needs specification of filter in a form of function.

3.8 Caching

As was stated in the previous chapter, RPQR will need to use cache to save time consumed by building information about packages contained in configured repositories. After careful consideration, the approach of using JSON as a format of the cache was chosen. This way, even third-party tools will be able to manipulate it and users will be able to read it if necessary. The cache will be invalidated only by users' request to do so and when a new plugin is found. The presence of plugins will be tested by a special record in the cache. Fortunately, all this is supported by the networkX library.

3.9 API design

RPQR needs to have a Python application programming interface so external developers can use its plugins and features even more efficiently than through a command-line interface and create their solutions. API will be using the same RPQR language as the command line interface and queries will be returning networkx graphs. This way, applications using the API can walk through nodes that represent packages and perform any transformation of the graph that they deem useful.

3.10 Command line interface design

RPQR tool will be using one main positional option that has to be provided and that is a query written in RPQR language. By default, the output will go into standard output and log messages to standard error output.

The first two options that can, but do not have to be specified will be whether the user wants to see a visualization of result created by his query and what attributes or relations should be included in the output. Filtering of attributes and relations is helpful because, with multiple plugins, there can be a lot of unnecessary data in the output that takes a lot of space. Another important option is the location of the configuration file if the user does not want to use the default location which is */etc/rpqr.conf*. The last option is whether the tool should invalidate the cache and build it again.

Chapter 4

Implementation and evaluation

With research and design complete, the RPQR project can now be implemented in the best way possible. This chapter will cover a deep description of the project's code and the algorithms it uses to fulfill the assigned task. Each section will cover the limitations of the presented solution and what could be done in the future to overcome them. Usage of the RPQR project will also be described in a form of a user manual and a description of the way how new plugins are supposed to be developed using the RPQR interface.

4.1 Important parts of implementation

This section will show the most important parts of the implementation and structure of Python code.

RPQRConfiguration

RPQRConfiguration class serves for purposes of loading plugins and creating RPQR language structures necessary for its successful parsing and interpreting. An instance of this class has to be created for every use of the RPQR project and is used by all following components and their diverse operations.

Initialization of plugins

```
def _initializePlugins(self):
  """ Load plugins from supplied directories
  """
  for dir in self.pluginDirectories:
      sys.path.append(dir)
      pluginModules = os.listdir(dir)
      for file in pluginModules:
          moduleName = file[:-3]
          # if file name starts with _ then it is most likely not a plugin
          if moduleName.startswith("_"):
              continue
          cfg = None
          if moduleName in self.userConfiguration.keys():
              cfg = self.userConfiguration[moduleName]

          if (cfg != None and cfg.get("disabled") == "1"):
              self._logger.info(
                  "%s plugin was disabled in configuration" % moduleName)
              continue
          module = importlib.import_module(moduleName)
          pluginClass = getattr(module, moduleName)

          pluginInstance = pluginClass(rootLogger=self.rootLogger,
                                       config=cfg)
          self.plugins.append(pluginInstance)
```

This is the centerpiece of plugin loading. This method walks through all configured directories which according to configuration should contain plugin modules and if its file does not start with an underscore then attempts to import them and create their instance. Two conditionals relate to the plugin configuration. The first one is checking whether a configuration related to this plugin exists and thus should be provided to it and the second one is there for a case when a user does not want to use the plugin at all, to save space for example, or to speed up processing.

RPQRLoader

RPQRLoader is a class responsible for the loading of data about packages through plugins and constructing graph structures out of them. It is taking advantage of DNF API to retrieve the data as efficiently as possible and access them in the same way as the package manager would.

Construction of graph

```
def createDatabase(self, cache: str = None) -> networkx.MultiDiGraph:
  """ Get graph of packages with data and relations described by plugins

  :param cache: path to cache file, defaults to None
  :type cache: str, optional
  :return: Graph of packages
  :rtype: networkx.MultiDiGraph
  """
  graph = networkx.MultiDiGraph()
  dataPlugins = [plugin for plugin in self.plugins if isinstance(
      plugin, RPQRDataPlugin)]
  relationPlugins = [plugin for plugin in self.plugins if isinstance(
      plugin, RPQRRelationPlugin)]

  pluginRecords = []
  for plugin in dataPlugins + relationPlugins:
      pluginRecords.append((plugin, plugin.__class__.__name__))

  ...

  av_query = self._getAvailableQuery()
  q_avail = av_query.run()

  for id, pkg in enumerate(q_avail):
      graph.add_node(id)
      for pluginInstance in dataPlugins:
          pluginInstance: RPQRDataPlugin
          pluginInstance.fillData(id, pkg, graph)

  for id, pkg in enumerate(q_avail):
      for pluginInstance in relationPlugins:
          pluginInstance: RPQRRelationPlugin
          pluginInstance.fillData(id, pkg, graph, av_query)

  graph.graph["plugins"] = [name for (_, name) in pluginRecords]
  ...
  return graph
```

This method distinguishes between plugins that are supposed to add attributes to package nodes and plugins that create relations between individual packages. The first kind is executed first so relationship plugins can depend on them later. Each instance of the plugin has its fillData method which is called for each package that was retrieved by DNF API. After the graph is built, a list of plugins that were present during the creation of this structure is saved into the plugins list for easier detection of invalid cache.

RPQRScanner

RPQRScanner is a class responsible for scanning the query entered in RPQR language format. It is able to recognize when there is a lexical error in the query and is used to parse a string into tokens.

Implementation of finite state machine

```
def getTokens(self, input: str) -> Optional[List[RPQRToken]]:
  ...
  while curInputIndex < len(input) + 1:
      if curInputIndex < len(input):
          c = input[curInputIndex]
      else:
          c = ''
      if curState == States.START:
          if c == '':
              break
          elif c == '(':
              curToken = RPQRToken(self.tokenTypes["leftBracelet"], c)
              curState = States.LEFTBRACELET
          elif c == ')':
              curToken = RPQRToken(self.tokenTypes["rightBracelet"], c)
              curState = States.RIGHTBRACELET
          elif c == '&':
              curToken = RPQRToken(self.tokenTypes["and"], c)
              curState = States.AND
          ...
          curInputIndex += 1
      elif curState == States.AND:
          tokens.append(curToken)
          curState = States.START
      elif curState == States.OR:
          tokens.append(curToken)
          curState = States.START
      elif curState == States.NUMBER:
          if c.isnumeric():
              curToken.appendToContent(c)
              curInputIndex += 1
          else:
              tokens.append(curToken)
              curState = States.START
```

Method getTokens is responsible for creating a list of tokens out of input. It is composed of a while cycle that parses characters and switches the state of the machine accordingly. It is strictly implemented accordingly to the graph of FSM which was mentioned earlier.

RPQRParser

RPQRParser is a class responsible for the processing of lexical tokens and the construction of abstract syntactic trees that can be interpreted in a strictly defined way. The class contains helper methods for easier manipulation with a list of tokens and methods considered as callbacks to certain operations encountered in the source query. These operations are uses of operators like *&* or ~ which needs the abstract syntactic tree to be constructed in a certain way.

parsing algorithm

```
while True:
  while True:
      if curInput.type in self.config.commandTypes.values():
          commandRule = None
          for rule in self.rules[4:]:
              if rule[0] == curInput.type:
                  commandRule = rule
          childList = [curInput]
          for indexMember, member in enumerate(commandRule[1:]):
              if member == self.nonTerminalTypes["statement"]:
                ...
              if argToken.type != member:
                ...
              if (argToken.type in
                [self.config.tokenTypes["number"], self.config.tokenTypes["
                    ↪ string"]]):
                  childList.append(argToken)
          newStatement = RPQRStackSymbol(
              self.nonTerminalTypes["statement"], childList)
          self.stack.append(newStatement)
          curInput = tokens.pop(0)
          continue

      lastTerminalIndex = 0
      ...

      requiredAction = precedencTable[self.stack[lastTerminalIndex].type][
          ↪ curInput.type]
      ...
  # decide whether we need to keep parsing or everything is already done
  if len(substatementQueue) == 0:
      return rootStatement
  else:
      curStatement = substatementQueue.pop(0)
      self.stack = [RPQRStackSymbol(self.config.tokenTypes["end"])]
      tokens = curStatement.children
      curInput = tokens.pop(0)
```

The parsing algorithm is based on the processing of a statement queue that contains all individual statements that need to be parsed. The first cycle is going through a queue of statements and the inner one is performing precedent syntactic analysis and calling appropriate callbacks. An interesting operation is that when substatement is encountered (command accepts a statement as an argument) algorithm cuts this substatement out of the source and inserts it into the queue for further resolution. Because of the tree's structure, it is possible to resolve the rest of the statement even when the construction of the substatement is not yet known.

The current parsing algorithm is not able to handle commands that take a dynamic number of arguments. This is a known limitation but because this feature was not needed in any relevant testing scenario, RPQR will not at the time of this thesis contain such an option.

RPQRInterpreter

RPQRInterpreter is a class responsible for the interpretation of RPQR language. It is mainly composed out of an algorithm that performs a depth-first search of the abstract syntactic tree and resolves nodes from bottom to up direction.

Interpretation algorithm

```
while len(stack) > 0:
    curNode = stack[-1]
    curResult = resultStack[-1]
    if curNode.operator is not None:
        if len(curResult.childResults) < 1:
            ...
        elif curNode.operator != '~' and len(curResult.childResults) < 2:
            ...
        else:
            # now we have all operands, we can begin resolution
            validNodes = []
            ...
            stack.pop()
            resultStack.pop()
    else:
        ...
        notResolvedStatementFound = False
        for argIndex, argType in enumerate(commandClass.args):
            if argType == str or argType == int:
                # literals can be resolved right away
                if (argIndex > len(curResult.childResults)-1):
                    curResult.childResults.append(RPQRResultTree(
                        curNode.children[1:][argIndex].content, []))
                else:
                    continue
            elif argType == list:
                if (argIndex > len(curResult.childResults)-1):
                    ...
                    break
                else:
                    continue
        if notResolvedStatementFound:
            continue
        arguments = []
        for partResult in curResult.childResults:
            arguments.append(partResult.result)
        curResult.result = commandClass.execute(graph, arguments)
        stack.pop()
        resultStack.pop()
```

The algorithm distinguishes between nodes that represent statements composed out of operator and operands and commands that filter packages. Processing of such nodes differs because operators are built-in and have a fixed number of arguments while commands are defined by plugins and every command can have a different number of arguments. The algorithm walks through the abstract syntactic tree and performs partial operations by calling execute method of plugins with already loaded arguments. When the root node is reached by resolution and its result is known, then the result can be returned by *performCommands* method and formated by users' requirements.

RPQR script

RPQR script is a command-line utility that allows users to use the RPQR project comfortably. It is designed to take advantage of the whole project and its features while providing the user with the ability to control for example when a cache file should be invalidated and overwritten.

RPQR script implemenation

```
rpqrcfg = RPQRConfiguration(pluginDirectories, namexrepository, cfgParser
    ↪ )
loader = RPQRLoader(rpqrcfg)
graph = loader.createDatabase(cacheFile, args.clearcache)
# we will not be performing empty query
if len(args.query) == 0:
    sys.exit(0)
result = RPQRQuery.performQuery(args.query, graph, rpqrcfg)
if result is None:
    sys.exit(1)
# we will filter result attributes according to supplied parameters
if len(args.filterattributes) != 0 or len(args.filterrelations) != 0:
    ...
    if len(args.filterattributes.split(";")[0]) != 0:
        for node in result.nodes:
            for key in list(result.nodes[node].keys()):
                if not key in args.filterattributes.split(";"):
                    del result.nodes[node][key]
    if len(args.filterrelations.split(";")[0]) != 0:
        for node in result.nodes:
            for u, v, edge_key in graph.out_edges([node], keys=True):
                if not edge_key in args.filterrelations.split(";"):
                    graph.remove_edge(u, v, key=edge_key)
# if result should not be visualized then just print it
# in JSON format to stdout
if not args.visualize:
    print(json.dumps(json_graph.node_link_data(
        result), indent=4, sort_keys=True))
    sys.exit(0)

# labeling requires some more processing
...
pos = networkx.spring_layout(result)
networkx.draw_networkx(result, pos=pos, with_labels=True, labels=
    ↪ labelDict)
edgeLabels = dict([((n1, n2), key) for n1, n2, key in result.edges])
networkx.draw_networkx_edge_labels(result, pos=pos, edge_labels=
    ↪ edgeLabels)
plt.show()
```

Implementation of RPQR script uses RPQR API and allows users to specify whether the result should be visualized or printed out through the standard output. Another very useful feature is the filtering of attributes and relations that should be included in the output. The script is using matplotlib python library to render graphs of packages if required.

4.2 User manual

This section contains a manual of the RPQR tool and a detailed description of the way how it is intended to be used. Another part provides information about the development of plugins and scripts that are using RPQR API to retrieve and filter package metadata.

NAME

RPQR - RPM package query resolver

SYNOPSIS

RPQR [-h] [--cfgpath CFGPATH] [--filterattributes FILTERATTRIBUTES] [--filterrelations FILTERRELATIONS] [--visualize] [--clearcache] query

DESCRIPTION

RPQR utility is supposed to make querying RPM repositories about package metadata easy by providing the user with the means to filter them by such metadata and individual types of relations that occur between them. Utility is configurable through configuration file which is located by default in */etc/rpqr.conf*.

OPTIONS

- -h, --help

  Show help message and exit

- --cfgpath CFGPATH

  Path to configuration file

- --filterattributes FILTERATTRIBUTES

  Specify list of attributes which interest you in the result. If left empty, then all attributes will be present in result

- --filterrelations FILTERRELATIONS

  Specify list of relations which interest you in the result. If left empty, then all relations will be present in result

- --visualize

  Visualize result

- --clearcache

  Clear cache

CONFIGURATION FILE

The following configuration file should illustrate general principles of how the RPQR utility behavior can be changed with it.

```
[RPQR]
pluginDirectories=["./rpqr/loader/plugins/implementations"]
cache=/var/tmp/rpqr.json

[RPQRRepo_f34-repo]
url=http://ftp.fi.muni.cz/pub/linux/fedora/linux/releases/34/Everything/
    ↪ x86_64/os/

[RPQRMaintainerPlugin]
url=https://src.fedoraproject.org/extras/pagure_owner_alias.json
```

The first section named *RPQR* is the main configuration section that contains the most important setting. *pluginDirectories* is an array of directories that contain python modules with RPQR plugins. *cache* is the path to the cache file, when this path is not supplied then RPQR utility will not use cache.

The second section named *RPQRRepo_f34-repo* is meant to set up the repository that which user wants to query. There can be one to n number of repositories and they all have to be configured in their section with prefix *RPQRRepo_* and member URL which specifies the base URL of the repository.

The third section is required for *RPQRMaintainerPlugin*. Each plugin can have its section of configuration and member disabled, which when set to *1* will prevent this plugin from working. Plugin configuration is described by plugins individually and is mentioned here only for the clarification of examples.

Example of use

RPQR „ONWHATDEPENDS('libyang-1.0.225-1.fc34.x86_64', 1)“ --filterattributes „name“ --filterrelations „depends“ --visualize

RPQR language

RPQR language serves as a means to specify what packages the user wants to see in the result. Take advantage of operators to create an appropriate combination of commands to get the results that you want.

Operators

- & - package has to conform to both right and left statements

- | - package has to conform to either left or right statements
- ~- package must not conform to statement located on the right

Parenthesis

RPQR also supports parenthesis to provide further means to set the priority of statements that are specified. Use parenthesis to make your query more readable and to make sure that the result is what you expect. *statement1 & (statement2 | statement3)* This statement is not equal to the version without parenthesis specified like this *statement1 & statement2 | statement3*. The semantic of the first statement is **Find packages that conform to statement1 and at the same time conform to either statement2 or statement3**. On the other hand, the second statement meaning is: **Find packages that conform to both statement1 and statement2 but if the package conforms to statement3 then it does not have to conform either to statement1 or statement2**

Official distributed plugins documentation

This section of the manual contains documentation about the behavior of plugins that are officially distributed with the RPQR tool and supported by the maintainers.

RPQRNamePlugin

RPQRNamePlugin is one of the most important plugins for RPQR utility. It gathers the complete name of the package, meaning its name, version, release, and architecture. It is an attribute plugin and inserts attribute *name* into the package.

- Added attribute: 'name'
- Added commands: 'NAME', 'NAMELIKE', 'SUBSETNAMELIKE'
- Depends on plugins: None

Commands provided by RPQRNamePlugin

NAME

Required arguments: name (string literal)

NAME command filters out only package that has the same name attribute as was specified with the name argument.

Example of use: *NAME('libyang-1.0.225-1.fc34.x86_64')*

NAMELIKE

Required arguments: name (string literal)

NAMELIKE command filters out packages that contain substring specified with the argument *name*.

Example of use: *NAMELIKE('libyang')*

SUBSETNAMELIKE

Required arguments: name (string literal), statement (RPQRLanguage statement)

SUBSETNAMELIKE command filters out packages returned by argument *statement* that contain substring specified with the argument *name*.

Example of use: *SUBSETNAMELIKE('x86_64', NAMELIKE('libyang'))*

Explanation of the example semantics: This query returns packages that contain *libyang* in their name and at the same time *x86_64* substring. The difference between this statement and *NAMELIKE('x86_64') & NAMELIKE('libyang')* is that the first query will be faster because it has to go through an only subset of packages.

RPQRDependencyPlugin

RPQRDependencyPlugin is a relation plugin that gathers information about package dependencies and creates dependency relations between nodes that represent them in the RPQR graph of packages.

- Added relation: 'depends'
- Added commands: 'ONWHATDEPENDS', 'WHATDEPENDSON'
- Depends on plugins: RPQRNamePlugin

Commands provided by RPQRDependencyPlugin

ONWHATDEPENDS

Required arguments: name (string literal), depth (numeric literal)

ONWHATDEPENDS command filters out packages on which package, with name attribute matching *name* argument, depends. *depth* argument is controlling the depth to which RPQR should go when gathering dependencies from the graph. Depth zero means that only the package specified by name will be present in the output, value one causes that only direct dependencies will be present, and so on.

Example of use: *ONWHATDEPENDS('libyang-1.0.225-1.fc34.x86_64', 1)*

WHATDEPENDSON

Required arguments: name (string literal), depth (numeric literal)

WHATDEPENDSON command filters out packages that depend on the package, with name attribute matching *name* argument. *depth* argument is controlling the depth to which RPQR should go when gathering dependent packages from the graph. Depth zero means that only the package specified by name will be present in the output, value one causes that only directly dependent packages will be present, and so on.

Example of use: *WHATDEPENDSON('libyang-1.0.225-1.fc34.x86_64', 1)*

RPQRMaintainerPlugin

RPQRMaintainerPlugin is an attribute plugin that gathers information about maintainers who work on packages. It inserts attribute *maintainer* into packages. Plugin, unfortunately, depends on the format of the list of maintainers which has to be in JSON.

- Added attribute: 'maintainer'
- Added commands: 'MAINTAINER', 'DEPENDSONUSER'
- Depends on plugins: RPQRDependencyPlugin

Commands provided by RPQRMaintainerPlugin

MAINTAINER

Required arguments: maintainers name (string literal)

MAINTAINER command filters out packages that have a maintainer specified with the argument maintainers name in the list of their maintainers.

Example of use: *MAINTAINER('tkorbar')*

DEPENDSONUSER

Required arguments: maintainers name (string literal), depth (numeric literal)

DEPENDSONUSER command filters out packages that depend on the work of the maintainer specified with argument *maintainers name*. That means that depth zero will retrieve packages that have specified maintainer in the list of its maintainers as *MAINTAINER* command would. Values higher than zero will retrieve packages that depend on those retrieved with depth zero.

Example of use: *DEPENDSONUSER('tkorbar', 1)*

RPQRMaintainerPlugin configuration

RPQRMaintainerPlugin has one additional variable for configuration not included in the default setting for all plugins. It is a variable URL that specifies the location of the maintainer list.

Example:

```
[RPQRMaintainerPlugin]
url=https://src.fedoraproject.org/extras/pagure_owner_alias.json
```

LICENSE

4.3 API documentation and example of scripting

RPQR project allows developers to create their plugins which can further extend its ability to recognize attributes and relations between packages. This section will show already existing plugins on which it will describe plugin development and the importance of individual parts.

RPQRMaintainerPlugin

```
class RPQRMaintainerPlugin(RPQRDataPlugin):
  """ Plugin allowing us to store information about package maintainers
      and ask Queries about them
  """
```

Plugin which wants to add a new attribute to packages needs to inherit from RPQRDataPlugin class. Please pay attention to the file name of your python module. The filename has to be the same as the name of the class.

```
desiredName = "maintainer"

implementedCommands = [MaintainerFilter, DependsOnUserFilter]
```

desiredName is the name of the attribute that this plugin wants to add. *implementedCommands* is a list of classes of commands that this plugin wants to add to the RPQR project. Commands will be described later.

```
packageToMaintainer = None

def __init__(self, rootLogger: Logger = None, config: dict = None):
    self.listUrl = None
    if config == None and rootLogger != None:
        lgr = rootLogger.getChild("RPQRDataPlugin")
        lgr.warning("url for retrieval of maintainers was not supplied")
        return

    self.logger = rootLogger.getChild(
        "RPQRDataPlugin") if rootLogger != None else None

    self.listUrl = config.get("url")
```

packageToMaintainer is a helper class variable. A plugin can declare its variables as it sees fit. The plugin needs to be ready for use-cases when it does not has access to logger or configuration and has to act accordingly. Because *RPQRMaintainerPlugin* needs to know the URL for retrieval of the maintainer list, it has to access configuration. All such use cases have to be documented.

```
def _downloadJson(self):
    if self.listUrl == None:
        return {}
    receivedResponse = requests.get(self.listUrl)

    if receivedResponse.status_code != 200:
        self.logger.error(
            "RPQR was unable to retrieve maintainer list from supplied
                ↪ url %s" % self.listUrl)

    return receivedResponse.json().get("rpms", {})
```

This is a helper method for retrieval of the maintainer list. Plugins can access any source of information that they want but this also has to be documented.

```
def prepareData(self, pkg: hawkey.Package) -> List[str]:
    """Get maintainers of package

    :param pkg: hawkey package information
    :type pkg: hawkey.Package
    :return: list of maintainers
    :rtype: List[str]
    """
    # download package maintainer list and build dictionary from it
    if RPQRMaintainerPlugin.packageToMaintainer is None:
        RPQRMaintainerPlugin.packageToMaintainer = {}
        data = self._downloadJson()
        data: dict
        for name, value in data.items():
            RPQRMaintainerPlugin.packageToMaintainer[name] = value
    # owner alias json uses source names as keys
    return RPQRMaintainerPlugin.packageToMaintainer[pkg.name if pkg.
        ↪ source_name == None else pkg.source_name]
```

prepareData is called for each package and has to return a value that should be saved to the implemented attribute. If block is there because the first run of this method will retrieve a list of maintainers and create a helper dictionary to accelerate loading. Please use such approaches to keep the project fast. After such initialization, every package is just returned a list of maintainers from *packageToMaintainer* dictionary. The ternary operator is there because source packages have different naming conventions in DNF API than the binary ones and RPQR has to be ready for both.

DependsOnUserFilter

```
class DependsOnUserFilter(RPQRFilteringCommand):
  """Command allowing filtering packages which depend on certain
     ↪ maintainer.
  That means the person is either maintaining them or package depends on
  package which they maintain recursively.
  """
  args = [str, int]
  name = "DEPENDSONUSER"

  def execute(graph: networkx.MultiDiGraph, args: list) -> List[int]:
      """ Get list of ids of packages which depend on user specified in
         ↪ args[0]
      to max depth args[1]

      :param graph: built graph of packages
      :type graph: MultiDiGraph
      :param args: arguments supplied to command
      :type args: list
      :return: node ids of packages which depend on specified maintainer
      :rtype: List[int]
      """
      targetUser = args[0]
      depth = int(args[1])
      nodes = [a for a in list(graph.nodes)
               if targetUser in graph.nodes[a]["maintainer"]]

      return RPQRFilteringCommand._BFS(graph, nodes, depth, "depends")
```

This is a more complex command out of *RPQRMaintainerPlugins* two commands so it is more worthy of explanation. All commands have to inherit from *RPQRFilteringCommand* class and declare *args* array and *name* class variable. *args* is an array that tells the RPQR project what arguments this command needs. Allowed types are str, int and list. Str means string literal, int means numeric literal and list is a result of substatement. A list is handled as a list of integers. *name* is the name of the command used to invocate it.

All commands have to implement *execute* method. *execute* method receives a built graph of packages and their arguments. As you can see, the first command gathers the ids of nodes that have specified maintainers in their maintainer list. Ids are then passed to static method _BFS which is a prepared implementation of Breadth-First Search. Breadth-First Search then finds dependent nodes to the specified depth.

RPQRDependencyPlugin

```
class RPQRDependencyPlugin(RPQRRelationPlugin):
  """Plugin for gathering dependencies of packages and allowing filtering
  by them
  """
  desiredName = "depends"
  implementedCommands = [OnWhatDependsFilter, WhatDepensOnFilter]

  def __init__(self, rootLogger: Logger = None, config: dict = None) ->
      ↪ None:
      self.optionalDataStructure = None

  def prepareData(self, pkg: hawkey.Package, graph: MultiDiGraph, query:
      ↪ hawkey.Query) -> List[int]:
      """ Get list of nodes to which we want to form this relation

      :param pkg: hawkey package object. Holds information supplied by dnf
          ↪  api
      :type pkg: hawkey.Package
      :param graph: graph of packages
      :type graph: MultiDiGraph
      :param query: hawkey query object. Allows further queries through
          ↪ dnf api
      :type query: hawkey.Query
      :return: list of target nodes for this node
      :rtype: List[int]
      """
      # we will use dictionary for optimalization of this process
      if self.optionalDataStructure == None:
          self.optionalDataStructure = {}
          for (node, attribs) in graph.nodes.items():
              self.optionalDataStructure[attribs["name"]] = node

      edges = list()
      requiredPackages = query.filter(provides=pkg.requires).run()
      for dependency in requiredPackages:
          target = self.optionalDataStructure[str(dependency)]
          edges.append(target)
      return edges
```

Relation plugins are very similar to those adding attributes with a few differences. Relation plugins can access a graph of packages that they can access and Hawkey query from DNF API. Relation plugins are returning a list of target nodes of oriented edges that they are creating.

4.4 Use of RPQR API to perform queries

Working of RPQR API will be described on the *RPQROrphaned* script which was used for testing of RPQR project.

```
if __name__ == "__main__":
  # set up configuration
  config = RPQRConfiguration([os.path.dirname(rpqr.loader.plugins.
      ↪ implementations.__file__)],
                             [("fedora-repo", "http://ftp.fi.muni.cz/pub/
                                 ↪ linux/fedora/linux/releases/33/
                                 ↪ Everything/x86_64/os/"),
                              ("source-repo", "http://ftp.fi.muni.cz/pub/
                                 ↪ linux/fedora/linux/releases/33/
                                 ↪ Everything/source/tree/")],
                             {"RPQRMaintainerPlugin":{"url": "https://src.
                                 ↪ fedoraproject.org/extras/
                                 ↪ pagure_owner_alias.json"}})
  query = RPQRQuery(config)
  result = query.performQuery("MAINTAINER('orphan')")
  for node in list(result.nodes):
      # now we will recursively gather packages which depend on every
          ↪ orphaned package
      dependentPackages = query.performQuery(
          "WHATDEPENDSON('%s', 20)" % result.nodes[node]["name"])
      # get maintainers affected by this change
      maintainerList = []
      for pkgId in list(dependentPackages.nodes):
          for maintainer in dependentPackages.nodes[pkgId]["maintainer"]:
              if maintainer not in maintainerList:
                  maintainerList.append(maintainer)
      maintainerList = list(dict.fromkeys(maintainerList))
      # orphan is just a placeholder so remove it
      maintainerList.remove('orphan')
      print("%s => " % dependentPackages.nodes[node]["name"], end="")
      for maintainer in maintainerList:
          print("%s " % maintainer, end="")
      print()
```

Every use of the API needs to start with creating an instance of *RPQRConfiguration* class. After that, the only other required thing is an instance of *RPQRQuery* class and then it is possible to create perform queries. Results are returned as a *networkx* graph and have to be worked with as. This script walks through orphaned packages and finds other maintainers affected by their orphaning.

4.5 Evaluation

RPQR projects' usefulness was evaluated with the *RPQROrphaned* script mentioned earlier in the API documentation. The previous solution needed tens of minutes to create a list of orphaned packages and maintainers that are affected, RPQROrphaned script needs one minute on a system with a casual processor.

Chapter 5

Conclusion

The RPQR project proved that the problem with query resolution about various metadata of packages can be solved properly. The script that now performs tasks that were present in the past is faster than the previous solutions and the project is ready to adapt to various new tasks that can occur in the future.

All initial goals of this thesis were fulfilled and the project is ready for further development because of well-documented code and set of tests that can always verify whether new changes did not create a regression.

Bibliography

[1] *DNF, the next-generation replacement for YUM* [online]. 2022 [cit. 2022-01-25]. Available at: https://dnf.readthedocs.io/en/latest/.

[2] *GraphQL* [online]. 2021 [cit. 2022-01-26]. Available at: https://spec.graphql.org/October2021/.

[3] Bray, T. *RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format* [online]. 2017 [cit. 2022-01-26]. Available at: https://datatracker.ietf.org/doc/html/rfc8259.

[4] *Pickle - Python object serialization* [online]. 2022 [cit. 2022-01-26]. Available at: https://docs.python.org/3/library/pickle.html.

[5] Bailey, E. C., Nasrat, P., Saou, M. and Skyttä, V. *Maximum RPM: Taking the RPM Package Manager to the Limit* [online]. 2000 [cit. 2022-01-25]. Available at: http://ftp.rpm.org/max-rpm/s1-rpm-file-format-rpm-file-format.html.

[6] *Rpm.org - RPM Reference Manual* [online]. 2021 [cit. 2022-01-25]. Available at: https://rpm-software-management.github.io/rpm/manual/.

[7] *Createrepo* [online]. 2013 [cit. 2022-01-25]. Available at: http://createrepo.baseurl.org/wiki.html.

[8] *Query Language Understood by SQLite* [online]. 2022 [cit. 2022-01-26]. Available at: https://www.sqlite.org/lang.html.

[9] Bray, T., Paoli, J., , Sperberg McQueen, C. M., Maler, E. et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. 2008 [cit. 2022-01-25]. Available at: https://www.w3.org/TR/xml/.

[10] Ben Kiki, O., Evans, C. and Net, I. döt. *YAML Ain't Markup Language (YAML™) revision 1.2.2* [online]. 2009 [cit. 2022-01-26]. Available at: https://yaml.org/spec/1.2.2/.

[11] *Yum* [online]. 2018 [cit. 2022-02-09]. Available at: http://yum.baseurl.org/.