

RPKG: RADE package management and deployment of LabVIEW code in a multitier environment at CERN

Vorgelegt von
Elena GALETTI

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Master of Science
- M.Sc. -
genehmigte Abschlussarbeit.

Gutachter : Prof. Dr.-Ing. SOHAN LAL
Prof. Dr.-Ing. SABINE GLESNER
Betreuer : Dr. NICOLAI STAWINOGA
M.Sc. JOSEPH TAGG



Abstract

Among many tools available on the market, LabVIEW is a programming environment widely used at CERN to monitor and control facilities around the accelerator complex. It allows fast programming, especially for GUI development, while interfacing with CERN's control infrastructure.

RADE is a "Rapid Application Development Environment" developed at CERN and integrated into LabVIEW to access the accelerator infrastructure from a user application. The goal of RADE is thus to help LabVIEW users at CERN focus on solving their problems and creating their applications, instead of re-developing already existing interfaces to the devices within CERN's control system.

Over time, RADE has grown larger than intended in its original design, and now incorporates many modules, libraries, templates, and dependencies. Moreover, RADE is built through Continuous Integration (CI), and a single change into the framework would trigger CI to re-build it again. Therefore, a framework decoupling is needed to make RADE lighter, less complex and nested, also to allow CI to be triggered at the library level in the future.

In addition, CERN's computing infrastructure features a multitier environment as well as different operating systems, which a single package manager cannot handle. As a result, different methods for installing and updating packages on a system were used in combination, that was hard to keep track of.

We propose RADE Package Manager (RPKG), a new Python-based package manager for LabVIEW, tailored for CERN's control infrastructure. RPKG helps package and distribute libraries and modules, including RADE, overcoming the shortcomings of existing solutions. This work proposes the core library `rpkglib`, a user-friendly GUI, through which users can install, create, and manage LabVIEW and RADE packages, and an evaluation of its functionalities. The newly developed package manager is compatible with legacy and future versions of LabVIEW, also when multiple versions are installed on a single system.

Keywords: Package Manager, LabVIEW, RADE framework, Python, CERN multitier infrastructure.

Zusammenfassung

Unter den vielen auf dem Markt erhältlichen Programmen ist LabVIEW eine am CERN weitverbreitete Programmierumgebung zur Überwachung und Steuerung des Strahls und des gesamten Beschleunigerkomplexes. Sie ermöglicht eine schnelle Programmierung, insbesondere für die Entwicklung von grafischen Benutzeroberflächen (GUI) und stellt gleichzeitig eine Schnittstelle zur Steuerungsinfrastruktur des CERN dar.

RADE ist ein “Rapid Application Development Environment”, das am CERN entwickelt und in LabVIEW integriert wurde, um von einer Benutzeranwendung aus auf die Beschleunigerinfrastruktur zuzugreifen. Ziel von RADE ist es, LabVIEW-Nutzern am CERN dabei zu helfen, sich auf die Lösung ihrer Probleme und die Erstellung ihrer Anwendungen zu konzentrieren, anstatt bereits vorhandene Schnittstellen zu den Geräten innerhalb des Steuersystems des CERN neu zu entwickeln.

Im Laufe der Zeit ist RADE jedoch größer geworden, als es im ursprünglichen Entwurf vorgesehen war, und umfasst nun viele Module, Bibliotheken, Vorlagen und Abhängigkeiten. Darüber hinaus wird RADE über Continuous Integration (CI) erstellt, und eine einzige Änderung am Framework würde den gesamten Erstellungsprozess auslösen. Daher ist eine Entkopplung des Frameworks erforderlich, um RADE leichter, weniger komplex und verschachtelt zu machen und um in Zukunft CI auf Bibliotheksebene auslösen zu können. Darüber hinaus besteht die Rechnerinfrastruktur des CERN aus einer mehrschichtigen Umgebung und verschiedenen Betriebssystemen, die ein einzelner Paketmanager nicht bewältigen kann. Infolgedessen wurden verschiedene Methoden zur Installation und Aktualisierung von Paketen in Kombination verwendet, was schwer zu überblicken war.

Wir stellen den RADE Package Manager vor, einen neuen, auf Python basierenden Paketmanager für LabVIEW, der auf die Steuerungs- und Regelungsinfrastruktur des CERN zugeschnitten ist. Dieser hilft beim Erstellen und Verteilen von Paketen, Bibliotheken und Modulen, einschließlich RADE, um die Unzulänglichkeiten der bestehenden Lösungen zu überwinden. In dieser Arbeit werden die Kernbibliothek `rpkglib`, ein benutzerfreundliches GUI, über die Anwender LabVIEW- und RADE-Pakete installieren, erstellen und verwalten können, sowie eine Bewertung der neuen RPKG-Funktionen, vorgestellt. Der neu entwickelte Paketmanager RPKG ist mit älteren und zukünftigen Versionen von LabVIEW kompatibel, auch wenn mehrere Versionen auf einem einzigen System installiert sind.

Keywords: Paket Manager, LabVIEW, RADE-framework, Python, CERNs Infrastruktur.

Eidesstattliche Versicherung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 15. August 2022

Elena Saletti
.....

Vorname NACHNAME

Acknowledgements

First of all, I would like to thank my colleagues in the MTA team at CERN. In particular, I would like to thank my CERN supervisor, Joseph Tagg, that guided me during this internship and master thesis work, while still giving me the freedom to choose many technical details.

Then, a thank goes to the EIT Digital organisation for giving me the opportunity to study and carry out my master degree at two different European universities, TU Eindhoven and TU Berlin.

Finally, I would like to thank my master thesis supervisor, Nicolai Stawinoga, from TU Berlin for his support and the time he dedicated to following my master thesis project.

Contents

1	Introduction	1
1.1	Thesis outline	1
1.2	Context of this work	1
1.3	Motivation and contributions of this work	4
2	Background knowledge	7
2.1	CERN, European Organisation for Nuclear Research	7
2.2	CERN's Control system	10
2.3	LabVIEW at CERN and RADE framework	12
2.4	Package managers	13
2.4.1	Dependency resolution	14
2.4.2	Dependency hell	15
2.4.3	LabVIEW package format	16
2.5	Repositories	17
3	Related work	19
3.1	<i>State-of-the-art</i> package managers	19
3.2	Package managers at the OS-level	20
3.2.1	dpkg and RPM, distributing binary packages for FOSS distributions	22
3.2.2	Dependency hell example on Debian	23
3.2.3	Portage, distributing source code for Gentoo	24
3.2.4	Pacloud, Cloud-based Linux package manger	24
3.2.5	Pure functional package managers	24
3.2.6	Homebrew, macOS package manager	25
3.2.7	Spack, Supercomputing Package manager	26
3.2.8	OPM, Ontology-based package manager	26
3.3	Package managers at the application-level	26
3.3.1	Existing package managers distributing LabVIEW code	27
3.4	Package manager security	27
4	RPKG: RADE Package Manager	31
4.1	The requirements	31
4.2	Installation of packages	32
4.3	Creation of packages	34
4.3.1	Source project in GitLab	34
4.3.2	From source project to binary package using build file	36
4.4	Decoupling of RADE	38
4.5	Supporting CERN's environment for CERN's users	39
4.6	Software architecture	39

4.6.1	rpkglib	40
4.6.2	oslib	47
4.6.3	GUI	47
4.7	Chapter conclusions	50
5	Evaluation	53
5.1	Dependency solver algorithm validation	53
5.2	Installation of packages validation	54
5.3	Creation of RADE packages validation	60
5.4	Cross-platform validation	62
6	Conclusions and future work	65
6.1	Conclusions	65
6.2	Future works	66
A	Example of a SPEC file	67
B	Build file created for LabVIEW JSON Library	71
C	SPEC file generated by RPKG for LabVIEW JSON Library	73
D	Calling LabVIEW from Python	77
E	LabVIEW code	79
F	Dependency solver algorithm	83
G	RPKG's Graphical User Interface	85
	Bibliography	91

List of Figures

1.1	Integration of RADE in LabVIEW	2
1.2	Nested libraries in the RADE framework	3
2.1	CERN's accelerator complex [1]	8
2.2	CERN's contributions to society [2]	10
2.3	The control system's three-tier architecture [3]	11
2.4	Control expert monitoring experiments at CCC [4] through GUIs . .	11
2.5	RADE framework, written in LabVIEW, allowing CERN users to connect to the accelerator infrastructure	13
2.6	Installing a package	15
2.7	Uninstalling a package	15
2.8	Content of the package <code>oglib_file 4.0.1.22</code>	16
2.9	Icons from RADE palette	17
2.10	Nexus repository [5] for RPKG	18
3.1	Positive features for package managers, taken from [6]	21
3.2	<i>rpm</i> SPEC file, taken from [7]	23
3.3	<i>deb</i> SPEC file, taken from [7]	23
3.4	Installing package <code>python-simpy</code> with <code>apt-get</code> , <code>aptitude</code> , and <code>smart</code> [8] .	23
3.5	VIPM GUI layout	28
3.6	NIPM GUI layout	28
4.1	Installation of packages	33
4.2	Creation of packages	35
4.3	Example of a source project organisation in GitLab: LabVIEW JSON Library	36
4.4	LabVIEW JSON Library	37
4.5	Dependencies within RADE framework	38
4.6	RPKG software architecture	40
4.7	Dependency tree example for explaining database functionalities . .	42
4.8	<code>packages</code> table from SQLite database	43
4.9	<code>dependencies</code> table from SQLite database	43
4.10	<code>install_files</code> table from SQLite database	44
4.11	Repository content filtered for <code>OpenG</code> library	44
4.12	Dependency solver algorithm example	46
4.13	GUI for RADE Package Manager, included in Appendix G in a bigger format	47
4.14	GUI Package information, included in Appendix G in a bigger format	48
4.15	GUI Installation page, included in Appendix G in a bigger format . .	49
4.16	Menu bar of the GUI for creation of packages	49

4.17	Menu bar of the GUI for local installation of packages	50
4.18	GUI CERN SSO, included in Appendix G in a bigger format	50
5.1	Certification authority 1.0.0.5	54
5.2	MGI File 1.1.0.4	55
5.3	OpenG Toolkit 4.0.1.9	55
5.4	OpenG File Library 4.0.1.22	55
5.5	Pane Relief 2.0.3.1	56
5.6	Package information page for <code>oglib_error-4.2.0.23</code>	56
5.7	Installing <code>oglib_error-4.2.0.23</code>	56
5.8	Installation completed for <code>oglib_error-4.2.0.23</code>	57
5.9	Database entry for the table “packages” for <code>oglib_error-4.2.0.23</code> .	57
5.10	Database entry for the table “install_files” for <code>oglib_error-4.2.0.23</code>	57
5.11	Installing <code>oglib_file-4.0.0.22</code>	58
5.12	<code>oglib_file-4.0.0.22</code> and its dependencies installed	58
5.13	Database entry for the table “packages” for <code>oglib_file-4.0.0.22</code> and its dependencies	58
5.14	Open G palette in LabVIEW 2020	59
5.15	RPKG page detecting that the package is badly formatted	59
5.16	Package <code>lj1-1.0</code> created	60
5.17	Package <code>lj1-1.0</code> inside RADE framework in LabVIEW 2020	60
5.18	Dependencies for <code>japc</code> to be installed	61
5.19	<code>japc-1.0</code>	61
5.20	RPKG application running on Linux in the GPN	63
5.21	RPKG application running on Linux exposed to the TN	63
5.22	RPKG application running on macOS Monterey	64
E.1	Mass compile VI	80
E.2	Generation menu VI	81
G.1	GUI for RADE Package Manager	86
G.2	GUI Package information	87
G.3	GUI Installation page	88
G.4	GUI CERN SSO	89

List of Tables

1.1	Comparison between existent package managers and RPKG, partially taken from [9]	4
4.1	Dependency solver algorithm example	46

Acronyms

AD Antiproton Decelerator. 9

APT Advanced Package Tool. 22, 29

BFS Breadth-first search. 15, 51

CC7 CERN CentOS 7. 62

CCC CERN Control Centre. 10

CERN Organisation européenne pour la recherche nucléaire. 1, 2, 4, 5, 7–10, 12, 13, 20, 26, 27, 29, 32, 44, 49, 62, 65, 66

CI Continuous Integration. 3, 24, 34, 65, 66

CMW Controls Middleware. 3, 13

COTS Commercial off-the-shelf. 12

DAG Direct Acyclic Graphs. 26

DAQ Data acquisition. 13

DFS Depth-first search. 15, 21, 46, 51

ELENA Extra Low ENergy Antiproton. 9

FEC Front-end computers. 10

FOSS Free and Open-Source Software. 20–22, 24

GPN General Purpose Ethernet Network. 10, 11, 31, 62, 65

GUI Graphical User Interface. 2, 5, 12, 16, 27, 31, 32, 40, 47, 51, 53, 61, 62, 66, 85

HPC High Performance Computing. 5, 26

LabVIEW Laboratory Virtual Instrument Engineering Workbench. 2–5, 7, 12–14, 16, 17, 20, 24, 27, 32, 34, 36–38, 43, 45, 47, 48, 53, 57, 59–62, 65, 66

LEIR Low Energy Ion Ring. 9

LHC Large Hadron Collider. 8–10, 12, 13, 39, 44

Linac 3 Linear Accelerator 3. 9

Linac 4 Linear Accelerator 4. 8

MTA Measurement Test Analysis. 2–4, 12, 32, 37, 59, 65, 66

NI National Instruments. 12, 13, 27, 32

NIPM NI Package Manager. 27

OPM Ontology-based Package Manager. 22, 26

PS Proton Synchrotron. 8, 9

PSB Proton Synchrotron Booster. 8, 9

RADE Rapid Application Development Environment. 2–5, 7, 12, 13, 26, 31, 32, 34, 36–39, 50, 51, 54, 60–62, 65

RBAC Role Based Access Control. 13

RPKG RADE Package Manager. 4, 5, 7, 11, 14–17, 19–22, 24–27, 29, 31, 32, 34, 37–42, 47, 49–51, 53, 54, 58, 62, 65, 66, 73

SPS Super Proton Synchrotron. 8, 9

TN Gigabit Ethernet Technical Network. 10–13, 27, 31, 39, 50, 62, 65

VIPM VI Package Manager. 27

YaST Yet another Setup Tool. 22, 29

YUM Yellowdog Updater, Modified. 22, 29

Introduction

Contents

1.1 Thesis outline	1
1.2 Context of this work	1
1.3 Motivation and contributions of this work	4

1.1 Thesis outline

In this work, a new package manager is proposed and discussed. The goal of the newly developed package manager, RADE Package Manager (RPKG), is to manage and deploy LabVIEW code in CERN environment, which is very specific, multitier environment.

Therefore, this thesis is organised as follows:

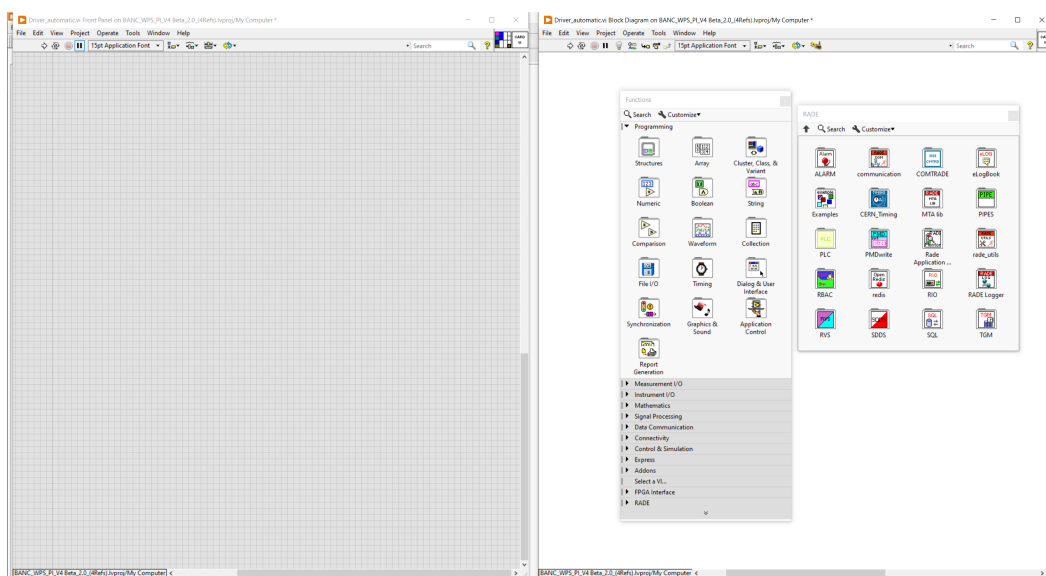
- [Chapter 1](#) provides the context, motivation and contributions of this work;
- [Chapter 2](#) illustrates the background knowledge, such as CERN accelerator complex, CERN's infrastructure and control system, the programming tool LabVIEW, the RADE framework, and the common aspects of package managers;
- [Chapter 3](#) analyses the *state-of-the-art* package managers and compares them with the newly developed RPKG;
- [Chapter 4](#) presents RPKG and the software, discusses technical implementation, and shows its user interface;
- [Chapter 5](#) provides four different test cases;
- finally, [Chapter 6](#) draws conclusions and proposes improving future works.

1.2 Context of this work

[Organisation européenne pour la recherche nucléaire \(CERN\)](#) is one of the largest and most respected centres for scientific research on particle physics and high-energy physics using particle accelerators, which collide particles with a velocity close to

From an engineering point of view, hardware and software integration into the CERN accelerator is a great challenge, because of its complexity. In this context, Graphical User Interface (GUI) applications are typically used allowing the control operators to easily monitor and control the beam and the whole CERN's accelerator complex [10]. Therefore, having tools that allow rapid development and fast prototyping for these applications is a crucial factor at CERN.

As **LabVIEW**, **RADE** is cross-platform. It can be installed in any **LabVIEW** version from 2015 on, on any of the three main operating systems (macOS, Linux, Windows) and it can be used from **LabVIEW**'s block diagram, as **Figure 1.1** shows. For each version of **LabVIEW**, there is a specific version of **RADE**. **Measurement Test**



Analysis (MTA) is the responsible team at CERN for having created, developing and maintaining RADE.

RADE started being developed around 2009, and since then it had lots of interactions with many different teams at CERN and became a tool used in both accelerator equipment and experiments. It grew larger than expected in terms of functionalities,

modules, libraries, and templates, but also in terms of dependencies and complexity [9], with nested libraries, as Figure 1.2 shows, where the user has to go 3 sub-folders down from RADE palette to use a specific function. It became so complex that it was necessary to bundle the whole framework into one application since it was hard to keep track of dependencies of every single library. RADE palette can be

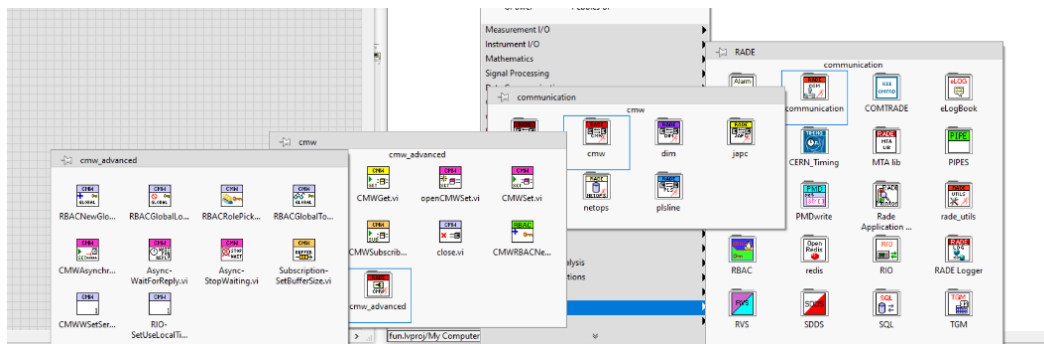


Figure 1.2: Nested libraries in the RADE framework

integrated into [LabVIEW](#) by downloading and running the [RADE](#) executable that corresponds to the [LabVIEW](#) version meant to be used ¹. Therefore, a software developer that wants to use [RADE](#) is required to install the whole framework, even when only few functions or libraries (e.g. [Controls Middleware \(CMW\)](#) library, introduced in [Section 2.3](#)) would have been used for a given project.

Since 2013, **RADE** is built through **Continuous Integration (CI)** with Jenkins [12]. By automating the build and unit testing stages thanks to **CI**, **RADE** release process time was reduced from one day to 53 minutes and the immediate integration of the last changes became possible [13]. This change in the build process brought advancements for **MTA** section. Nonetheless, it made it challenging for **RADE** end-users, especially if the new release was changing some of the dependent interfaces, that were not backwards compatible [13]. In addition to that, a single change in one of **RADE** libraries is still currently triggering **CI** to build a new version of the whole framework, also to the parts that were unchanged.

To trigger [CI](#) at the library level and to prevent [RADE](#) libraries to become more nested, a better organisation is needed, where framework decoupling is required. A smart way to decouple [RADE](#) into smaller bundles is by creating packages out of it. Therefore, using a package manager for creating, installing and updating packages on a system would definitely help in distributing [RADE](#) in a lighter and more organised way, where the end users install only the packages required and not the whole framework.

Name	Package supported	Local server	Cross-platform	Mixed language	License	Suitable for RADE
RPKG	LabVIEW, extensible	Yes	Yes	Yes	-	Yes
VIPM [14]	LabVIEW	Paid	Yes	No	Paid	No
NIPM [15]	LabVIEW	Yes	No	Yes	NI	No
dpkg [16]	Linux, Debian and derivatives	Yes	No	Yes	GPLv2	No
opkg [17]	Linux, embedded systems	Yes	No	Yes	GPLv2	No
RPM [18]	Linux, Red Hat/Fedora/Suse and derivatives (e.g. CentOS)	Yes	No	Yes	GPL	No
homebrew [19]	macOS, Linux	Yes	No	Yes	MIT	No
conda [20]	Python, R	Yes	Yes	Yes	BSD	No
pip [21]	Python, PyPi library	Yes	Yes	No	MIT	No
Spack [22]	Python	Yes	No	No	MIT	No

Table 1.1: Comparison between existent package managers and RPKG, partially taken from [9]

1.3 Motivation and contributions of this work

This work proposes **RADE Package Manager (RPKG)**, a new Python-based package management system for **LabVIEW** to help initially package and distribute libraries, templates, and modules, including **RADE**, for **MTA** team and other **LabVIEW** users at **CERN**, to overcome the shortcomings of existing solutions used at **CERN** or available on the market.

In fact, on the market there are many tools that automatically update software (e.g. package managers) and manage version control. Even the **LabVIEW** community

¹The installer can be found in this webpage <https://readthedocs.web.cern.ch/display/RADE/Download>

has two tools, described in [Subsection 3.3.1](#), which are responsible for distributing [LabVIEW](#) code. Nonetheless, [CERN](#)'s infrastructure, explained in [Section 2.2](#), and the platform requirements are limiting factors, that led to the development of a new package manager, [RPKG](#).

In summary, in this thesis we provide the following contributions:

1. we analyse the *State-of-the-Art* for package managers ([Chapter 3](#)), highlighting their limitations with reference to the context of [CERN](#) and [RADE](#), motivating our decision to develop [RPKG](#);
2. we present [RPKG](#) ([Chapter 4](#)): [RPKG](#) is a novel package manager, cross-platform and built for [CERN](#)'s multitier environment, which creates, installs, manages packages, that decouples the [RADE](#) framework, while providing a user-friendly [GUI](#);
3. we evaluate [RPKG](#)'s main functionalities, such as installation and creation of packages ([Chapter 5](#));

In [Table 1.1](#), a comparison among some of the package managers available on the market and [RPKG](#) is provided. From this table, many of the available package managers do not satisfy the cross-platform requirement, so they are not considered to be suitable. However, also the cross-platform tools (e.g. [VIPM](#), [conda](#), or [pip](#)) are far away to give the same contributions that [RPKG](#) offers, either because they are specific to their own environment (e.g. [pip](#) distributing Python libraries only) or because they would add a higher cost than added benefit (e.g. [VIPM](#)) [\[9\]](#).

As [RPKG](#), also [Spack](#) [\[22\]](#) was developed for a specific environment, the [High Performance Computing \(HPC\)](#) environment, aimed at compensating the shortcomings that other package managers initially had in the [HPC](#) field.

[RPKG](#) has been developed for [CERN](#) environment and infrastructure, it is able to run on all the three main operating systems, designed in a way that it will not be restricted for distributing [LabVIEW](#) packages only.

[RPKG](#) is written in Python3, using PyQt5 for the graphical user interface, executing [LabVIEW](#) VIs² as well. It interfaces with web APIs (e.g. [Nexus](#), [CERN SSO](#)), databases ([SQLite](#)), files and other applications (e.g. [LabVIEW](#)). Technical details and implementation will be explained in [Chapter 4](#).

²[LabVIEW](#) programs are called virtual instruments, or VIs. This will be introduced in [Section 2.3](#).

Background knowledge

Contents

2.1	CERN, European Organisation for Nuclear Research . . .	7
2.2	CERN's Control system	10
2.3	LabVIEW at CERN and RADE framework	12
2.4	Package managers	13
2.4.1	Dependency resolution	14
2.4.2	Dependency hell	15
2.4.3	LabVIEW package format	16
2.5	Repositories	17

Chapter outline

In this chapter, the background knowledge necessary for this work is provided. In [Section 2.1](#), CERN's mission and the accelerator complex are explained. Then, in [Section 2.2](#), the CERN's control system is described. It is a fundamental part since RPKG is built according to CERN infrastructure. In [Section 2.3](#), the reasons why LabVIEW is used at CERN are provided, together with history and a more detailed description of the RADE framework functionalities. Since this work is about developing a new package manager, in [Section 2.4](#) and [Section 2.5](#), the background knowledge for package managers and repositories is given.

2.1 CERN, European Organisation for Nuclear Research

CERN is one of the largest particle physics laboratories in the world, established in 1954, in Geneva near the Franco-Swiss border, and run now by 23 Member States. CERN focuses its research on particle physics and high-energy physics, trying to answer fundamental questions about the universe by studying the tiniest particles in nature. In fact, studying these fundamentals allows scientists to understand what happened a few moments after the Big Bang.

Since its foundation, CERN has always aimed at improving the understanding of the universe, enriching human knowledge while pushing back the frontiers of technology, developing the most advanced scientific instruments to carry on the experiments.

To reach its goals, CERN is using particle accelerators, which are meant to accelerate (or decelerate) and collide particles.

The accelerator complex, shown in Figure 2.1, is a succession of machines with increasingly higher energies, and each machine is injecting the proton or ion beam in the next one, which is bringing the beam to higher energy levels [23]. The electron-volt (eV) is the most frequently unit of measurement used in particle physics.

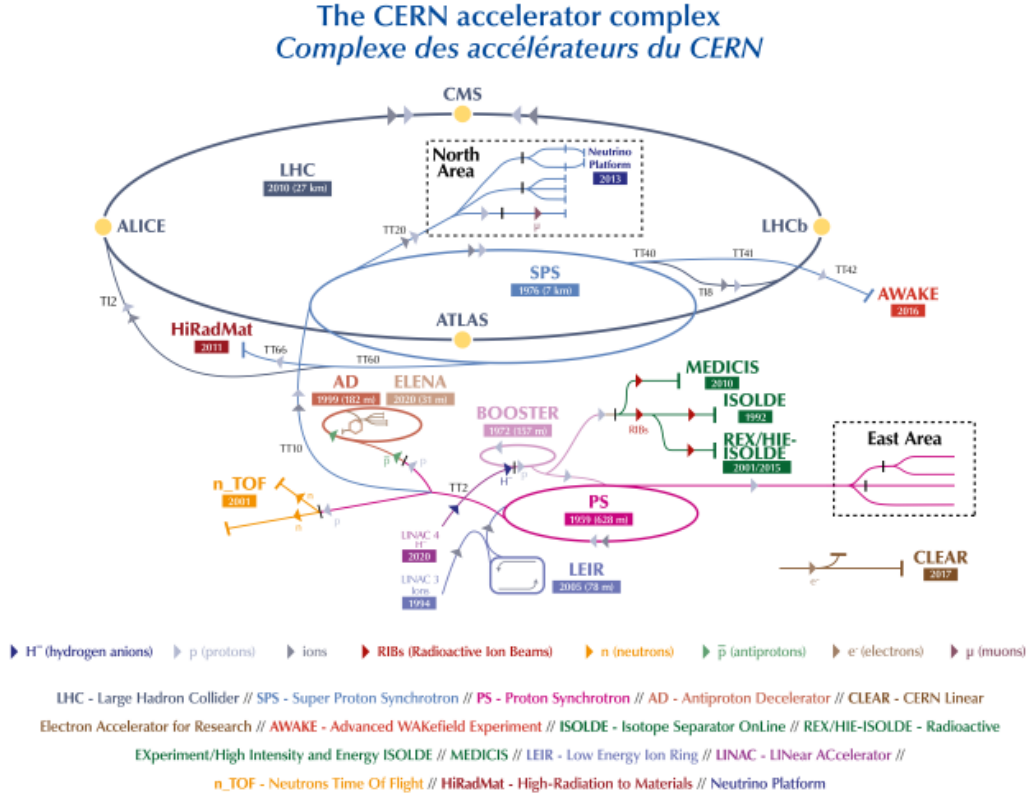


Figure 2.1: CERN's accelerator complex [1]

The Linear Accelerator 4 (Linac 4) is source of proton beams at CERN. It accelerates negative hydrogen ions ¹ to 160 MeV, leading them to the Proton Synchrotron Booster (PSB).

In PSB, the two electrons of H^- will abandon the atom, leaving the hydrogen atom with one proton only, since hydrogen does not have any neutrons. At this moment, the protons are accelerated to 2 GeV into the Proton Synchrotron (PS), which accelerates the protons up to 26 GeV. From the PS, protons are now ready to get injected into the Super Proton Synchrotron (SPS), where they are pushed up to 450 GeV. Only at this stage, the proton beam is finally ready to be injected into the Large Hadron Collider (LHC). In order to collide protons, the beam is split into two beam pipes: one beam goes into one pipe and it circulates clockwise, while the

¹ H^- , which is a hydrogen atom with one more electron added to its configuration.

other proton beam is injected into the other pipe and it circulates anticlockwise. The **LHC** is the world's most powerful accelerator, which consists of a ring of 27 km circumference and contains more than 1200 dipole magnets, more than 8000 other superconducting magnets operating at temperatures down to 1.9 K and with currents up to 12000 A [10]. In **LHC**, protons (or hadrons) circulate close to the speed of light and approximately one billion collisions are generated every second. It takes 4 minutes and 20 seconds to fill each **LHC** ring, and approximately 20 minutes for the proton beam to reach its maximum energy of 6.8 TeV [24]. These two beams generated are meant to collide in four different points of **LHC**, exactly where the detectors ALICE, ATLAS, CMS, and LHCb are situated. At the moment of the collision, the energy is around 13.6 TeV [25].

A network of computers filters, records, and processes the data flow from these collisions from the four detectors. The computing grid processes millions of gigabytes of data every year, allowing physicists around the world to analyse data. **LHC** experiments include around 150 million sensors delivering data 30 million times per second. Even after filtering, the amount of data produced is around 50PB per year [23].

In all of these years, several important achievements in particle physics have been made, such as “the landmark discovery of the Higgs boson” [26] in 2012 when CMS and ATLAS experiments announced they had each observed a new particle in the mass region around 125 GeV, which is consistent with the predicted mass of the Higgs particle.

From Figure 2.1, the other particles accelerated are ions. From **Linear Accelerator 3** (Linac 3), ions are generated, brought to **Low Energy Ion Ring (LEIR)** and then they follow the same route as the one of protons.

Around **LHC**, there are also other experiments, such as **MEDICIS** or **ISOLDE**, that use the protons or ions from **PSB**, **PS**, or **SPS**.

Moreover, the accelerator complex has also experiments such as **Antiproton Decelerator (AD)** and **Extra Low ENergy Antiproton (ELENA)**, where protons are decelerated, instead of accelerated, producing low-energy antiprotons for studying antimatter.

Besides **CERN**'s primary mission, the technology and expertise developed for particle physics research can be applied to fields other than high-energy physics. **CERN** aims at having a positive impact on society globally, bringing benefits to the medical field (e.g. with **Medipix** [27] or at **MEDICIS** experiment [28]), to the area of computing, and to many other fields, as depicted in Figure 2.2. A remarkable example is the invention of the World Wide Web, conceived at **CERN** in 1989 [29], developed for sharing accelerator data among research institutes around the world, that only later on influenced deeply the modern society.

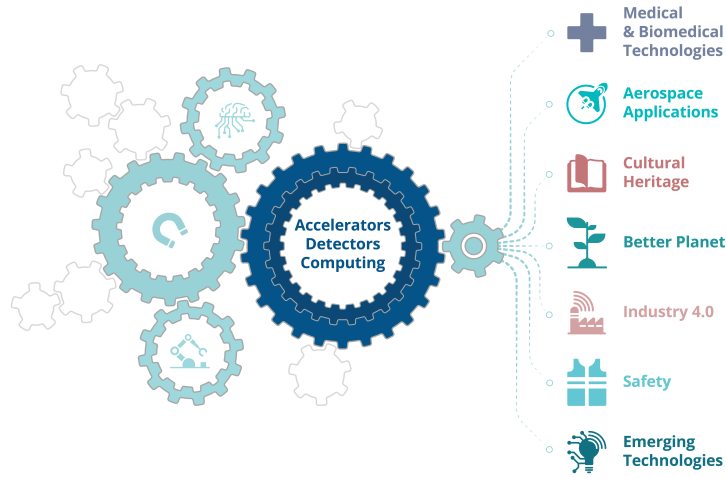


Figure 2.2: CERN's contributions to society [2]

2.2 CERN's Control system

Particle accelerators are made of many components meant to control and monitor mainly the beam and the equipment or facilities around it. Therefore, a control system was developed in order to let physicists and operators remotely control the LHC and the experiments around it. The CERN's control system, shown in Figure 2.3, is structured in a three tier architecture [3], which means there are three different and distinct physical layers.

There are two different networks as well: one is the General Purpose Ethernet Network (GPN), and the other one is the Gigabit Ethernet Technical Network (TN). The GPN reaches all the buildings at CERN and connects generic users and services, while the TN is the network for LHC control and it connects critical devices (e.g. application servers in Figure 2.3) for TN management and monitoring.

Regarding the three tiers, described in [3], the top (or client) tier is composed by Linux-based computers used in control rooms at CERN, such as the well-known CERN Control Centre (CCC), where the high-level graphical applications are running and the responsible operators are controlling experiment values through the monitors of control room consoles, as the example shows in Figure 2.4.

The middle (or business) tier is composed by back-end servers, databases, and file servers. They are rack-mountable and multi-CPU PCs.

The lower (or front-end) tier is composed by embedded computers or PLCs running real time applications, written in C++, interacting with electronic boards controlling and monitoring accelerator components. Those computers are also called Front-end computers (FEC) because they are strictly involved in the first sensor acquisition part of the experiments.

During the implementation of [RPKG](#), an important part is to make sure that the tool can run on the three main operating systems exposed to the [GPN](#), but also on the Linux-based computers exposed to the [TN](#). This requirement will be addressed in [Chapter 4](#) and in the tests, reported in [Chapter 5](#).

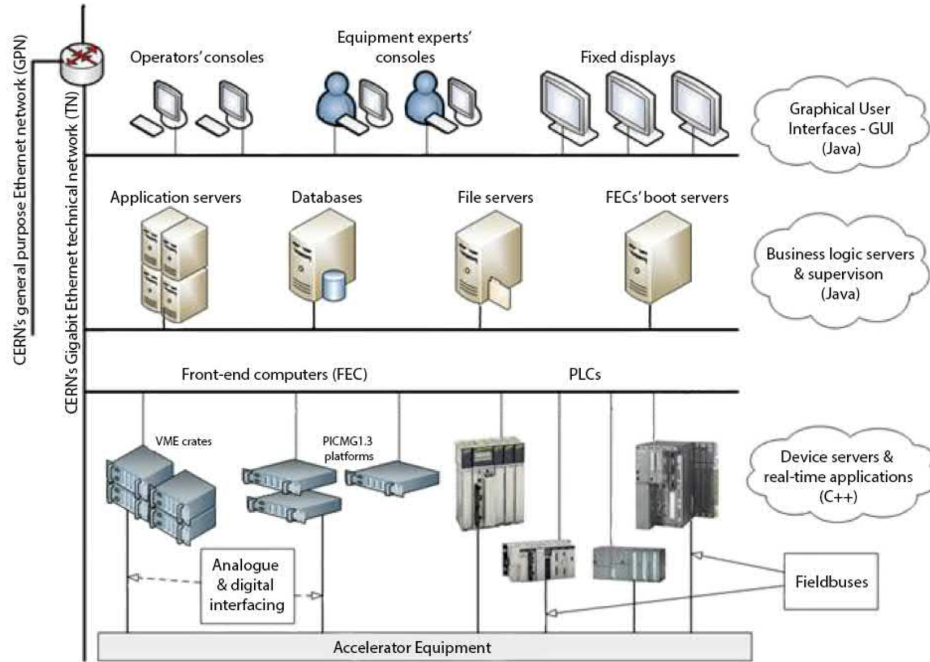


Figure 2.3: The control system's three-tier architecture [3]



Figure 2.4: Control expert monitoring experiments at CCC [4] through GUIs

2.3 LabVIEW at CERN and RADE framework

GUI applications are widely used in many facilities at CERN. Therefore, CERN's engineers were looking for tools to allow rapid development of these applications, where measurements, tests, and analysis can be programmed in a quick way, providing the user interface part as well.

Some tools, such as Matlab or Excel, are highly used and recognised in both industry and academia but their scope is quite limited in terms of data acquisition from CERN's facilities, and in terms of integration with accelerator equipment and CERN control infrastructure, described before in Section 2.2. In general, these tools have quite low flexibility in terms of hardware and software integration.

By using LabVIEW, developed by National Instruments (NI), with C++ and Java extensions, the limitations mentioned before can be overcome [10].

LabVIEW provides development, debug, and deployment all in the same environment. It is based on a graphical language, enabling drag and drop for both code and GUI, allowing very fast programming, particularly for GUI. LabVIEW programs are called virtual instruments, or VIs. When a new VI is created, there are two windows: the front panel window, which is the user interface, and the block diagram, which contains the graphical source code. LabVIEW has a wide range of analysis libraries and a light and independent environment. It runs on all the three main operating systems, and the user can choose the version to install from 2009 to 2021 [30]. In addition, for Windows only, the user can choose the application bitness (32 or 64 bits). Multiple versions of LabVIEW can be installed on the same system, commonly happening among the LabVIEW users.

LabVIEW allows developers to interface with a huge variety of hardware equipment. In addition to that, by using LabVIEW with NI Commercial off-the-shelf (COTS) equipment, such as PC-based systems or PXIs or compactRIOs, the HW-SW integration becomes extremely easy to be handled, developed and maintained.

For all of these reasons, LabVIEW is a suitable software candidate for CERN to develop expert interfaces to the accelerator equipment.

Within LabVIEW, RADE framework was developed by the MTA section at CERN to integrate LabVIEW in the accelerator domain, bridging the programming environment LabVIEW with the accelerator infrastructure, aiming at creating a rapid application development environment, for users where they could focus on solving problems rather than re-developing interfaces to interact with the devices under the TN (e.g. databases). Therefore, RADE covers all the three tiers, described in Section 2.2, and its functions can be used directly from LabVIEW through the LabVIEW palette, as Figure 1.1 showed.

RADE's main motivation was the hardware commissioning of the LHC, where more than 10.000 circuits needed to be analysed during the powering tests of the LHC. To do the analysis, engineers had to interface with all the LHC equipment (e.g. power supplies, beam position monitors, or quench-, cryogenics temperature-, strain-, and vibration detectors). Successfully integrating NI COTS equipment in the CERN accelerator domain relies on many factors, highlighted in violet in Figure 2.5, but

mainly on three components [31]. The first one is [CMW](#), that provides a common communication infrastructure and protocol for the accelerator controls under the [TN](#). Second, the timing: an elaborate timing is required to synchronise electronic and electrical equipment in [LHC](#) at the sub nanosecond precision. Third, the configuration management to keep the devices up-to-date . Therefore, these three aspects are fully developed in [RADE](#), together with others, such as interfacing databases, PLCs, [Data acquisition \(DAQ\)](#) devices, and [Role Based Access Control \(RBAC\)](#), which is authorisation method designed to protect from accidental and unauthorised access to the LHC and injector equipment. Main functionalities of [RADE](#) are schematised in [Figure 2.5](#).

Summing up, [LabVIEW](#) is a widely used tool among others to develop expert tools, allow machine development analysis and test facilities integrated with the [CERN](#) control infrastructure, that do not have to be necessarily from [NI](#). The [RADE](#) environment is a framework, written in [LabVIEW](#) with Java and C++ extensions, aiming at accessing the accelerator infrastructures from a user application in a easier and faster way.

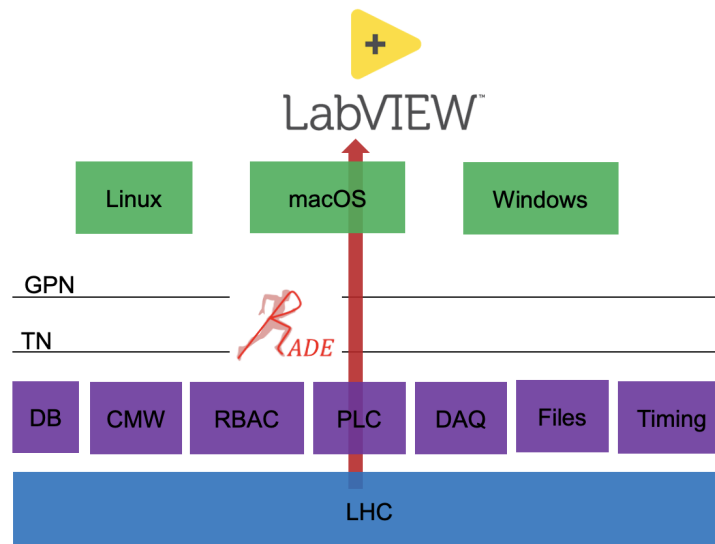


Figure 2.5: RADE framework, written in LabVIEW, allowing CERN users to connect to the accelerator infrastructure

2.4 Package managers

Nowadays, package managers are a popular and very straightforward way to distribute software, running on modern operating systems. Package managers distribute software bundled into archives, often called packages, providing a privileged, central mechanism for management of software on a computer system, gathering information on packages available on package repositories [32]. Package managers are used to install, update, configure or remove packages on a given system in a

consistent manner, and the package format consists of an archive containing installation files and metadata. For a given package, the package metadata contains information about the package itself (e.g. package's name, description of its purpose, version number, vendor) as well as the list of its dependencies and conflicts, necessary to be installed, or uninstalled in case of conflicts, in order to have the package running properly. Usually package metadata is in the format of a SPEC file, which can be thought of as the “recipe” that a package manager uses to actually install a package on a system, or that creates during the package creation process. An example is shown in [Appendix A](#). In [Subsection 2.4.3](#), the **LabVIEW** package format is described more in details.

For installation, package managers usually install the requested package as well as any additional package that is needed to let the former one to operate as it should. This process is called *dependency resolution*, and it is explained in [Subsection 2.4.1](#). Sometimes, dependency resolution can get more problematic than expected causing dependency issues, defined as *dependency hell* ([Subsection 2.4.2](#)).

2.4.1 Dependency resolution

Dependency resolution is a crucial concept and functionality for package managers, that is implemented for **RPKG** as well, explained in [Subsection 4.6.1.3](#). It consists in the automatic detection done by a package manager of packages that should be installed together with the desired package in order to let the latter work properly. The automatic dependency resolution is a feature of package managers lately developed that early package managers, from around 1994, did not have [\[33\]](#).

For example [\[32\]](#), a user requests a package **foo**, that depends on two other packages, **libc** and **bar**, as [Figure 2.6](#) suggests. Let's suppose that **libc** is already installed in the user's system, while **bar** is not. Then, a package manager will install package **foo** as requested and will suggest to install **bar** as a dependency since it is required to make the former one working correctly, and it will detect that **bar** is already installed.

A similar behaviour is also expected for uninstalling packages: a package manager will uninstall the package that a user selected and it will ask the user if also dependencies should be uninstalled as well. Hence, in [Figure 2.6](#) example, if the user wants to uninstall **foo**, then the package manager will suggest to install also **libc** and **bar** because it was installed previously as dependencies. However, the uninstalling process could be slightly more complicated, and it will be explained by extending the previous example to the one schematised in [Figure 2.7](#). Let's suppose that the user installed one more package, **cat**, after having already installed **foo** and its dependencies. The package manager will detect that **cat**'s dependency (i.e. **bar**) is already installed, so it will install only **cat**. Now, let's suppose the user wants to uninstall package **foo**. If the situation would still be the one depicted in [Figure 2.6](#), then package manager would suggest to uninstall dependencies **libc** and **bar**, as said before. But now the situation looks different: if a package manager would uninstall dependencies as well, then package **cat** would be broken since its dependency **bar** is

missing. In this case, either `cat` will be uninstalled as well, or it will stay together with `bar`. Therefore, when packages are sharing the same dependencies, package managers will usually tend to ask the user how to proceed.

Updating packages is also part of the dependency resolution algorithm. Either only the selected package will be updated, while old dependencies just checked if they can still satisfy the new updated selected package, or everything will be updated. Some

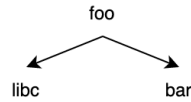


Figure 2.6: Installing a package

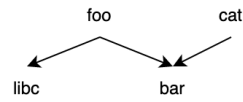


Figure 2.7: Uninstalling a package

of the algorithms used among package managers are [Breadth-first search \(BFS\)](#) [34], [Depth-first search \(DFS\)](#) [35], or SAT solvers (discussed in [Chapter 3](#)). The algorithm used for [RPKG](#) is [DFS](#) inspired.

2.4.2 Dependency hell

Dependency hell is the colloquial term that is used when more than one package has dependencies on the same shared packages or libraries, but they depend on different and incompatible versions of the shared packages. It comprehends many different situations, reported for example in [36], such as:

- Many dependencies: when a package depends on many others, causing lengthy downloads, or large amounts of disk space.
- Long chains of dependencies: when a package depends on another one, which depends on a different one as well, and so on. This can cause problems when two different versions of the same package are required (e.g. conflict dependencies, explained in the next point). In this particular situation, some package managers do not know what to do and leave the decision to the user, which has to solve dependencies manually.
- Conflicting dependencies: supposing a package `pkg-a` depends on package with a specific version `pkg-z 1.9`, and another package `pkg-b` depends on `pkg-z 1.8`. Then, `pkg-a` and `pkg-b` cannot be installed at the same time in the same environment.
- Circular dependencies: when `pkg-a` depends on a specific version `pkg-b` and vice versa. This can raise issues because upgrading one will break the other.
- Diamond dependency: when a package `pkg-a` depends on package `pkg-b` and package `pkg-c`, both `pkg-b` and `pkg-c` depend on `pkg-d`, but `pkg-b` requires `pkg-d 1.0` and `pkg-c` requires `pkg-d 2.0`. The build fails because only one version of `pkg-d` can exist.

Usually if a package ends up in one of the beforehand listed situations, then it can be agreed that the package is badly created.

2.4.3 LabVIEW package format

LabVIEW packages consists of installation files and metadata, like the typical package format, described in Section 2.4. The content of `oglib_file 4.0.1.22` package is shown in Figure 2.8. A LabVIEW package includes:

1. the installation files, which are all the files ending with the `.vi` extension for what concerns LabVIEW and `.mnu` files, which are binary files responsible to show library icons in the block diagram of LabVIEW environment;
2. the SPEC file, an example is included in Appendix A.;
3. the `.bmp` icon, useful to show the icon of the package in the GUI of the package manager;
4. the license file.

In Figure 2.8, installation files are contained inside the folders “Folder Group 0”, “Folder Group 1”, and “Folder Group 2”. In Figure 2.9, some icons from RADE

Name	Date Modified	Size	Kind
File Group 0	Today at 17:25	--	Folder
user.lib	Today at 17:25	--	Folder
_OpenG.lib	Today at 17:25	--	Folder
file	Today at 17:25	--	Folder
file.llb	Today at 17:25	--	Folder
File Group 1	Today at 17:25	--	Folder
_functions_oglib_file_1.mnu	5 September 2013 at 18:34	3 KB	LabVIEW...ocument
_functions_oglib_file_2.mnu	5 September 2013 at 18:34	4 KB	LabVIEW...ocument
functions_oglib_file.mnu	5 September 2013 at 18:34	4 KB	LabVIEW...ocument
File Group 2	Today at 17:25	--	Folder
dir.mnu	5 September 2013 at 18:34	4 KB	LabVIEW...ocument
icon.bmp	5 September 2013 at 18:34	4 KB	Window...P image
license	5 September 2013 at 17:33	2 KB	Document
spec	5 September 2013 at 18:34	6 KB	Document

Figure 2.8: Content of the package `oglib_file 4.0.1.22`

framework are shown, such `alarm` or `communication`. As said, LabVIEW is a graphical language, hence it is using `.mnu` binary files to display icons that users can drag and drop in the block diagram. In the package creation phase, `RPKG` is also responsible for the generation of these files.

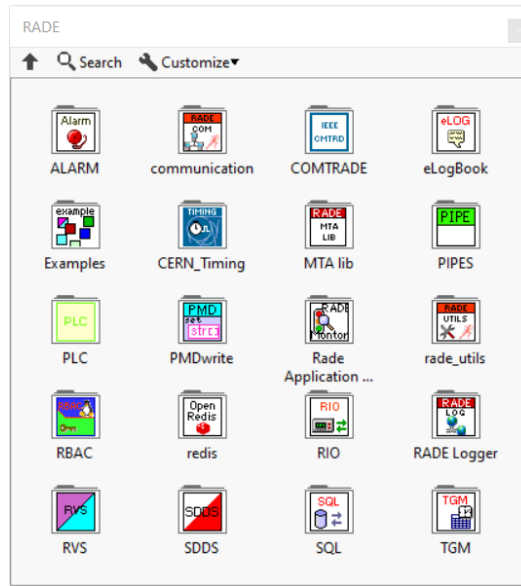


Figure 2.9: Icons from RADE palette

2.5 Repositories

Package managers interact with software repositories, binary repository managers and app stores, where usually bundles are stored. As package managers, many different repositories are available on the market. A package repository, or *repo* for short, is a data structure, usually a HTTP, HTTPS or FTP server [32], where software packages and their metadata can be stored. Often a table of contents is stored as well, which represents the whole updated content of a package repository. The package metadata is usually just a copy of the embedded package metadata for all packages on the repository. Package managers download the package metadata from a repository so that they know which packages are available from that repository. Nexus [37], showed in Figure 2.10, is one of many repositories available on the market, and it is the repository chosen for RPKG [5]. In Figure 2.10, it can be seen that the RPKG repository is storing LabVIEW packages and their SPEC file.

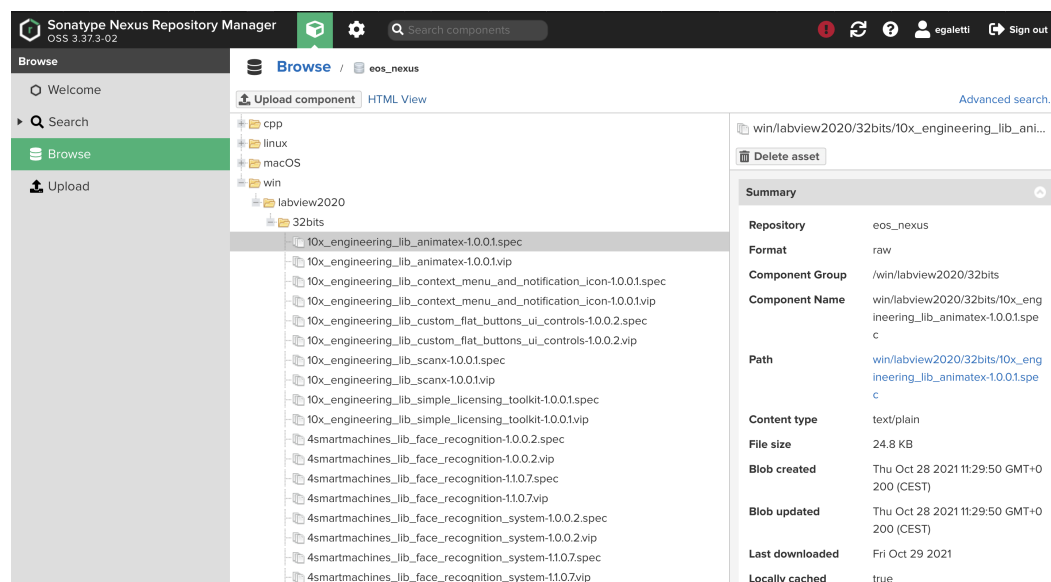


Figure 2.10: Nexus repository [5] for RPKG

Related work

Contents

3.1	<i>State-of-the-art</i> package managers	19
3.2	Package managers at the OS-level	20
3.2.1	dpkg and RPM, distributing binary packages for FOSS distributions	22
3.2.2	Dependency hell example on Debian	23
3.2.3	Portage, distributing source code for Gentoo	24
3.2.4	Pacloud, Cloud-based Linux package manger	24
3.2.5	Pure functional package managers	24
3.2.6	Homebrew, macOS package manager	25
3.2.7	Spack, Supercomputing Package manager	26
3.2.8	OPM, Ontology-based package manager	26
3.3	Package managers at the application-level	26
3.3.1	Existing package managers distributing LabVIEW code	27
3.4	Package manager security	27

Chapter outline

In this chapter, the *State of the Art* for package managers is provided. Package managers distributing software at the OS level and at the application level are discussed and compared with [RPKG](#). The last part of this chapter is dedicated to package manager security systems, which is an important feature developed and embedded in some of them.

3.1 *State-of-the-art* package managers

Package managers can usually be classified under different aspects [\[38\]](#), such as:

- the type of package format they distribute,
- the operating system (OS) they are developed for,
- the level at which they operate: if they are either handling packages at the OS-level or at the application level.

A package manager can distribute binary package, source code, or hybrid forms. For a binary form, all the packages are compiled and ready to be installed and use; whereas for source code format, either users know how to compile the packages, or they are provided with a script that automates the compilation process. [RPKG](#) distributes binary packages, already compiled, each bundle specific for a given [LabVIEW](#) version and operating system.

Most of the package managers are developed for Unix operating systems (e.g. Linux, macOS, Android, iOS) and fewer for Windows, and a minority of them is cross-platform, as it can be seen in [Table 1.1](#). This is one of the reasons why [RPKG](#) was developed, since [CERN](#)'s users work with all the three main operating systems.

The majority of the package managers are implemented using an imperative programming paradigm, while fewer of them are developed adopting pure functional language approach. [RPKG](#) is developed according to the imperative logic. Positive and negative aspects of these two paradigms are presented in [Subsection 3.2.5](#).

In [Section 3.4](#) package managers have been evaluated also in terms of security attacks, and vulnerabilities have been found in many of them.

Surprisingly, one of the few communities with a low adoption of package managers is the C++ Open-Source community [\[6\]](#). [Miranda et al. \[6\]](#) investigated into the obstacles that are preventing C++ Open-Source community to use package managers. The biggest concern is about the license, as sometimes package managers are commercial software. Moving to open-source package managers, other reasons were related to complicated configurations, insufficient documentation from these tools, possibility of bugs, the low availability of libraries, and many others listed in the paper. In the same paper, the authors also listed positive features that the C++ community expects being developed in package managers and the result is shown in [Figure 3.1](#). The most requested feature is again related to the license, being open source. Not far from that, there is the good availability of libraries. Of course, the more libraries are available in a repository, the more the package manager will be used from developers. Another big requirement concerns how a package manager can handle dependencies and conflicts. Therefore, an efficient dependency resolution algorithm must be developed. Other positive features listed involve a good documentation available, the ease of both installation and updating of packages, and the simplicity for the set up.

3.2 Package managers at the OS-level

[Free and Open-Source Software \(FOSS\)](#) distributions for Linux operating system have always been organised on the basis of packages, allowing applications to run correctly. Linux-based package managers are popular and very well supported with respect to others. However, when using some of them in combination, users might experience bugs or problems, because of incompatibility between them.

As introduced in [Subsection 2.4.1](#), dependency solvers are key element for package managers. It was demonstrated that determining whether a component can be in-

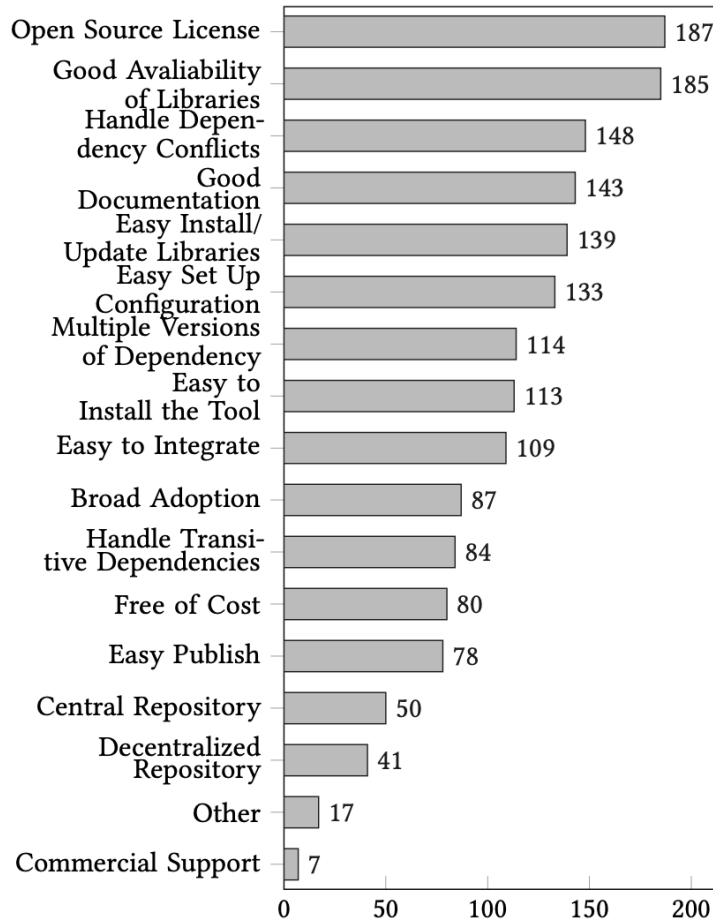


Figure 3.1: Positive features for package managers, taken from [6]

stalled for FOSS distributions is considered an NP-complete problem [39]. This led to incorporating SAT solvers into package managers for FOSS distributions aimed at resolving package dependencies using Boolean satisfiability, instead of implementing dependency solvers algorithms *ad-hoc*. A SAT solver is a computer program which aims to solve the Boolean satisfiability problem. In the case of dependency graphs, a SAT formula expresses the dependencies with a declarative approach using a propositional formula¹, then the computer will make the computations to understand if the formula is satisfiable or not, eventually providing a solution.

RPKG's dependency solver algorithm is implemented *ad-hoc*, following the DFS algorithm principles [40].

Even though dependency solvers are advancing, some of the most popular package managers on the most stable operating system version can still end up in a *dependency hell* situation, where the users get lost in dependencies and conflicts that they have to solve on their own manually. An example will be given in Subsection 3.2.2.

¹A propositional formula is a formula composed with Boolean variables and Boolean connectives.

Besides the FOSS distributions, package managers were also used in specific areas, for example in the high-performance computing [41], such as Spack, described in Subsection 3.2.7, or in the energy sector, such as Ontology-based Package Manager (OPM), described in Subsection 3.2.8. As Spack and OPM, RPKG is targeted to a specific environment as well.

3.2.1 dpkg and RPM, distributing binary packages for FOSS distributions

FOSS distributions are mainly divided into RPM and Debian based distributions. Based on this, two different package formats are used, either *rpm* or *deb* package formats, both distribute binary packages. Consequently, two different low-level package managers are involved: RPM [18], developed by Red Hat and mainly used for Red Hat, Fedora, Suse and derived distributions such as CentOS; dpkg, package manager for Debian and its derived OS (e.g. Ubuntu) [16]. A similar package manager to dpkg is opkg, which is used for Linux embedded systems [17].

The most used package managers based on RPM are Yet another Setup Tool (YaST) [42], Yellowdog Updater, Modified (YUM) [43], urpmi [44], and apt-pbo [45]. Urpmi and YUM are essentially higher level interfaces of RPM, with a dependency solver algorithm developed *ad-hoc*. As mentioned in Section 3.2, some other package managers uses SAT solvers, and this is the case for YaST. Apt-pbo is a fork of apt-get and uses a pseudo-Boolean solver [45].

For dpkg, the main ones used are Advanced Package Tool (APT) and aptitude [46], and the less frequently but still used are cupt [47] and smart [48]. Both APT and aptitude have an *ad-hoc* dependency solver.

Both *rpm* and *deb* packages are compressed bundles that include all the installation files and the metadata. The main differences between these two package managers are: the way they produce archive files and the format of their metadata specification. An *rpm* package has a specific format, it can be produced by the rpm-build mechanism only, while *deb* packages can be created by utilising standard tools such as tar. Moreover, *rpm* packages encode the specification from SPEC file in a binary form and constitute it as a part of its archive files; while *deb* packages make use of a textual representation [7]. Therefore, *deb* packages are easier to be processed. Both SPEC files are quite similar as Figure 3.2 and Figure 3.3 show. Both RPM and dpkg package managers handle dependencies in a semi-automatic way. In fact, these two management systems only provide a dependency list of packages that are required to be installed and the user should install the them manually. Their higher levels (e.g. APT, YUM) can handle dependencies in an automatic way: both of them install the specific package selected and its dependencies without requiring user's action. It is very difficult to find some criteria to choose the optimal installation candidate, because all of them present their pro and cons, such as dependency-hell situations could happen from time to time. Krafft et al. [49] provided some of the criteria on how FOSS developers pick their tools.

```

Name: ao_redpacket
Version: 1
Release: 5
Requires: platform.framework public_scripts

```

Figure 3.2: *rpm* SPEC file, taken from [7]

```

Package: vim
Version: 2.7.4
Depends: libacl1 (>= 2.2.51-8),libc6(>=2.15),libgpm2(>=1.20.4),libpython2.7(>=2.7),
libselinux1(>=1.32),libtinfo5,vim-common(=2.7.4.052),vim-runtime(=2.7.4.052)

```

Figure 3.3: *deb* SPEC file, taken from [7]

3.2.2 Dependency hell example on Debian

Abate et al. [8] demonstrated that even on stable Debian releases² there were some difficulties when trying to install the package `python-simpy` 1.8.1, which is a process-based discrete-event simulation framework written in Python, with the latest version available.

The package managers used were: `apt-get` v0.9.7, `aptitude` v0.6.7, `smart` v1.4, and `cupt` v2.5.6. In Figure 3.4, the results are shown. `apt-get` v0.9.7 proposes a solution where 119 packages have to be updated, 203 have to be installed, and 414 must be removed. Of course, this is a quite unfeasible solution since removing 414 packages is unacceptable. With `aptitude` v0.6.7 the situation is slightly improved, with 25 packages to be upgraded, 16 new packages to be installed and “only” 18 packages to be uninstalled. With `smart` or `cupt`, a solution was not found: the former entered in a long computation phase that was aborted after 10 minutes, while the latter fails immediately.

This example is meant to show how easy it is to end up in a *dependency hell* situation, even using a stable OS Debian release with the most used package managers, such as `aptitude`, mentioned in Subsection 3.2.1. Also, it is clear that some *dependency solvers* work better than others, as it is the case in this example where `aptitude` and `apt-get` work better than `smart` and `cupt`.

```

apt-get --without-recommends install python-simpy
The following extra packages will be installed:
  binutils ca-certificates coreutils cpp cpp-4.3
...
tk8.4 tottem-common whois xinit xkb-data zlib1g
The following packages will be REMOVED:
  abiword-common abiword-gnome amor apt apt-utils
...
xserver-common xserver-xfree86 yelp zenity
The following NEW packages will be installed:
  ca-certificates cpp-4.3 dbus
...
system-tools-backends tottem-common whois
The following packages will be upgraded:
  binutils coreutils cpp debconf dia-libs
...
ocaml-interp perl perl-base perl-modules
119 upgraded, 203 newly installed, 414 to remove
and 338 not upgraded.

aptitude --without-recommends install python-simpy
The following NEW packages will be installed:
  debian-archive-keyring [2010.08.28 lenny]
... python-simpy [1.8-1] ...
python2.5-minimala [2.5.2-15+lenny]
The following packages will be REMOVED:
  aptitudea [0.2.15.9-2] dia-commona
...
synaptica [0.55-cvs20050503-4] taskseta
The following packages will be upgraded:
  apt [0.5.28.6 → 0.7.20.2+lenny2]
...
zlib1g-dev [1:1.2.3.3.dfsg-12]
25 packages upgraded, 16 newly installed,
18 to remove and 784 not upgraded.

smart install python-simpy
Loading cache...
Updating cache... ***** [100%]
Computing transaction...
^C

```

Figure 3.4: Installing package `python-simpy` with `apt-get`, `aptitude`, and `smart` [8]

²They tried with `sarge`, `etchy`, and `lenny`. All of them are stable releases which passed all the tests before deployment.

3.2.3 Portage, distributing source code for Gentoo

Gentoo is a Linux distribution built using the Portage package management system [50]. For Gentoo, unlike binary software distributions, the source code is compiled on the local system according to the user's preferences and it is often optimised for the specific type of computer. Therefore, Portage is required to compile the packages that the user wishes to install from source code. In Portage's repository, *ebuild* files are stored. *Ebuild* files are bash files, that contain information about packages like SPEC files, but also functions to execute for custom installation depending on the system. Bloemen et al. [51] [52] discuss how Portage's dependencies management is handled. They parse *ebuild* files using Paludis [53], and they use a custom tool to build the dependency graph. Therefore, for Portage an *ad-hoc* dependency solver was developed, instead of using the available SAT algorithms. The drawback of distributing source code stands in the compilation time that occurs on the target system every time users want to install a package.

3.2.4 Pacloud, Cloud-based Linux package manger

Courtes et al. presented a cloud-based Linux package manager, called Pacloud, that uses Amazon Web Services (AWS) cloud computing features to compile and distribute Linux packages [54]. In the paper, they argue that Linux distributions have two main weaknesses: either packages are binary packages (e.g. *deb* packages), hence pre-build and therefore not optimised for specific hardware and often heavy for their target needs; or they are source code packages (e.g. Portage's packages), and therefore it takes plenty of time and system resource for compilation. To improve computing performance, they shifted the mass compilation part from the users' systems to cloud services. The biggest advantage is in the cloud itself: having a cloud is more reliable than a server and it can support a lot of users by scaling depending on the servers' load. A missing information in the paper are costs related to AWS, or any other cloud service involved. Therefore, it would be unrealistic to consider this package manager as suitable for FOSS distributions, since AWS is a paid service. Therefore, it is very likely that Pacloud might not satisfy the features requested by the C++ open source community [6].

RPKG is distributing compiled packages specific to the OS and targeted for a single LabVIEW version. The compilation part will be done upfront using CI and compiled packages are then stored in the repository, such that end users will not lose time and system resources in the compilation process, but they will just directly install packages by downloading them from the repository. On the other hand, more space on the repository will be required since each package will be stored multiple times for different LabVIEW versions and OS.

3.2.5 Pure functional package managers

In this subsection, the focus will be shifted from "traditional" package managers, to pure functional package managers.

Pure functional package managers are developed according to pure functional programming paradigm, where the build and installation processes are considered as pure function, without side effects [55]. This is in contrast with the imperative programming paradigm that other package managers put in place, such as dpkg, RPM or RPKG.

In imperative software deployment paradigm, package managers have access to the whole software installed on a machine and the installation modifies the files on the system. In other words, the installation, removal and upgrade of packages is mutating the operating system state.

Generally speaking, pure functional and imperative programming languages present both pro and cons. In a pure functional package manager, functions do not change the state of the program, so it is easy to reproduce the OS state. Operations (e.g. installation, removal, upgrading) are *transactional*, therefore an interruption will not lead to an undefined or unusable state of the system, where the developer does not know if files in the system have been altered or not, as it could be the case by using imperative languages. Rolling back to previous state of a system is feasible, while it is not for imperative languages since the formal mechanism to describe the system's configuration is not present.

On the other hand, the main disadvantage of having a pure functional package manager is that the code does not include loops, but recursive functions. Writing programs in a recursive style instead of using loops can be a difficult because it potentially reduces performance because complexity of a function is less controllable. Moreover, it leads to a reduction of the readability of the code. In addition, combining pure functions with the rest of the application and with the I/O can become tedious for a pure functional package manager. The advantages of the imperative approach led to develop RPKG using this paradigm, instead of using the pure functional approach.

In any case, it cannot be said if functional programming is better than the imperative, and it is out of the scope of this work. It is interesting as well as important to know that some of the package managers, such as Nix package manager [56] and GNU Guix [55], are developed accordingly to the pure functional paradigm for distributions. From Nix package manager, a new entire system configuration was build, called NixOS [57].

3.2.6 Homebrew, macOS package manager

Homebrew is a free package manager distributing source code with close Git integration, very popular and used for macOS, currently also available for Linux computers, called Linuxbrew [58]. It is written in Ruby and either targets the version of Ruby that comes installed with the macOS operating system or users need to install Ruby beforehand.

3.2.7 Spack, Supercomputing Package manager

Spack package managers was developed initially for the HPC environment [22]. The existing software management tools were not suitable for the complexity of HPC environment, therefore the Lawrence Livermore National Laboratory introduced Spack, a package manager suitable for HPC written in Python. With Spack, parameterised builds were introduced, where users could compose their own builds with version, compiler and dependencies rapidly. It introduced a new SPEC syntax, suitable for HPC environment to simplify the process of working with large software configuration files. Moreover, abstract Direct Acyclic Graphs (DAG), that most of package managers use, for dependencies are turned into concrete and “buildable” SPEC files. The authors of the paper [22] they provided four different use cases where Spack is increasing the operational efficiency at the Lawrence Livermore National Laboratory.

Spack is introduced and reported in this work because, RPKG, as Spack for HPC field, is developed for a targeted environment, CERN. Spack is not used only for supercomputers anymore, but for macOS and Linux distributions as well. Referring to Table 1.1, Spack cannot be used for RADE because it is lacking the cross-platform requirement.

3.2.8 OPM, Ontology-based package manager

Hviid et al. [59] introduced OPM, Ontology-based package manager for building operating systems in the energy sector moving to green energy. Building operating systems that can predict the demand-response of green energy was their goal. In the paper [59], authors developed an ontology-based package manager, that dynamically resolves dependencies and installs services. Ontology languages are formal languages used to construct ontologies, usually written with declarative languages. OPM uses ontology-based paradigm to model and resolve dependencies, and the unit of a package in the system, where OPM operates, is a container, such as Docker [60], used for deploying in a distributed environment. Once again, OPM, such as Spack and RPKG, was developed because of a specific target, but it was too tailored to its own environment to be used at CERN.

3.3 Package managers at the application-level

Besides package managers that operate at the OS-level, there are also application-level package managers. For example, Conda [20] distributes packages for Python and R, Maven [61] is for distributing Java code, or CPAN distributing Perl [62]. Conda is an open-source, cross-platform, language-agnostic package manager [63] and environment management system, originally developed to solve challenges that Python data scientists were having. Thanks to its virtual environments, users can switch between package versions, downloading and installing updates from a software repository. It manages projects written in any language, multi-language projects

as well. The main difference with pip [21] is that Python can be installed through Conda, while not through pip, which is only a package installer for python packages. Another big difference between Conda and pip, before version 20.3, was the way dependencies were handled. In all pip versions before 20.3, all the dependencies were installed whether or not those were in conflict with other packages previously installed. This was leading users to unpleasant situations where packages or the whole environment was broken or malfunctioning. On the other hand, Conda checks the current environment, making sure that the selected packages can work in the current configuration.

For Java projects, the primary tool used is Maven, that can also be used to build and manage projects written in C #, Ruby, Scala, and other languages. As for other packages with SPEC files, an XML file describes the software project being built, its dependencies, the build order, directories, and required plug-ins.

Conda, pip, and Maven are still widely used in both industry and academia.

3.3.1 Existing package managers distributing LabVIEW code

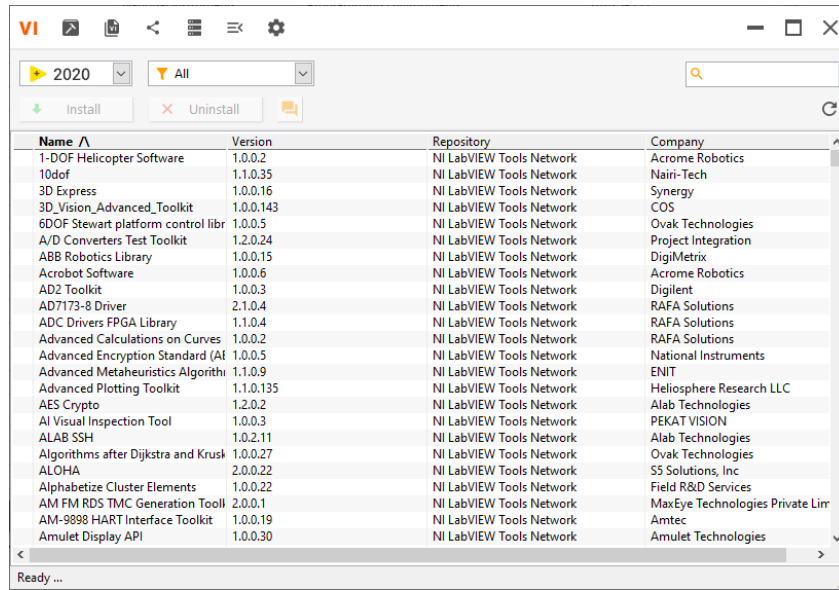
As introduced already in Section 1.3, two main package managers are currently used to distribute LabVIEW code around CERN: NI Package Manager (NIPM) [15], developed by NI, and VI Package Manager (VIPM) [14], developed by JKI [64]. As pointed in Table 1.1, the main shortcoming of NIPM consists of not being a cross-platform tool, being meant for Windows only. Not being cross-platform was clearly one of the main motivations to develop RPKG, since the main three operating systems are all widely used at CERN.

VIPM can run on the three main operating systems (Windows, macOS, and Linux), but one of the drawbacks consists of having a paid license for providing private and customised team package repositories and customised deployments. Therefore, every single users at CERN would need to pay the VIPM enterprise license [65] for making repositories exposed to the TN, which would add a higher cost rather than an added value.

In Figure 3.5 and Figure 3.6, the GUI of the two package managers are shown.

3.4 Package manager security

In the last 20 years, the number of package managers available increased significantly and now millions of developers rely on them for installation of packages on their machines. Therefore, given that package managers are responsible for the majority of software and libraries running on a system, it is important to make sure that the software they install is secure and without any malware. Hence lately, package managers started being evaluated also in terms of security, testing their behaviour under security attacks, analysing their potential weaknesses and vulnerabilities. In the literature [66][32], there are many papers that are raising awareness about malicious packages, which should be considered an important issue in the software ecosystem. Moreover, many of them show which anomalies are detected in package managers



Name \	Version	Repository	Company
1-DOF Helicopter Software	1.0.0.2	NI LabVIEW Tools Network	Acrome Robotics
10dof	1.1.0.35	NI LabVIEW Tools Network	Nairi-Tech
3D Express	1.0.0.16	NI LabVIEW Tools Network	Synergy
3D_Vision_Advanced_Toolkit	1.0.0.143	NI LabVIEW Tools Network	COS
6DOF Stewart platform control lib	1.0.0.5	NI LabVIEW Tools Network	Ovak Technologies
A/D Converters Test Toolkit	1.2.0.24	NI LabVIEW Tools Network	Project Integration
ABB Robotics Library	1.0.0.15	NI LabVIEW Tools Network	DigiMetric
Acrobot Software	1.0.0.6	NI LabVIEW Tools Network	Acrome Robotics
AD2 Toolkit	1.0.0.3	NI LabVIEW Tools Network	Digilent
AD7173-8 Driver	2.1.0.4	NI LabVIEW Tools Network	RAFA Solutions
ADC Drivers FPGA Library	1.1.0.4	NI LabVIEW Tools Network	RAFA Solutions
Advanced Calculations on Curves	1.0.0.2	NI LabVIEW Tools Network	RAFA Solutions
Advanced Encryption Standard (AI	1.0.0.5	NI LabVIEW Tools Network	National Instruments
Advanced Metaheuristics Algorithm	1.1.0.9	NI LabVIEW Tools Network	ENIT
Advanced Plotting Toolkit	1.1.0.135	NI LabVIEW Tools Network	Heliosphere Research LLC
AES Crypto	1.2.0.2	NI LabVIEW Tools Network	Alab Technologies
AI Visual Inspection Tool	1.0.0.3	NI LabVIEW Tools Network	PEKAT VISION
ALAB SSH	1.0.2.11	NI LabVIEW Tools Network	Alab Technologies
Algorithms after Dijkstra and Krusk	1.0.0.27	NI LabVIEW Tools Network	Ovak Technologies
ALPHA	2.0.0.22	NI LabVIEW Tools Network	SS Solutions, Inc
Alphabetize Cluster Elements	1.0.0.22	NI LabVIEW Tools Network	Field R&D Services
AM FM RDS TMC Generation Tool	2.0.0.1	NI LabVIEW Tools Network	MaxEye Technologies Private Lim
AM-9898 HART Interface Toolkit	1.0.0.19	NI LabVIEW Tools Network	Amtec
Amulet Display API	1.0.0.30	NI LabVIEW Tools Network	Amulet Technologies

Figure 3.5: VIPM GUI layout

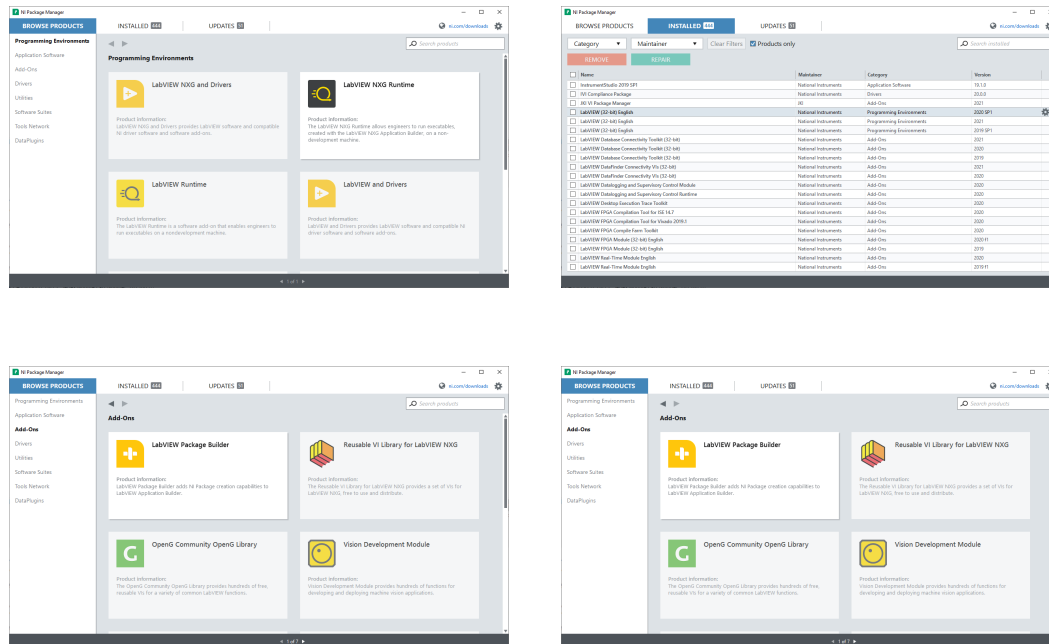


Figure 3.6: NIPM GUI layout

on the market, and propose changes in their security system. For example, authors of some papers suggest to change the communication protocol package from HTTP to its secure version HTTPS.

Package managers can be evaluated in terms of integrity checking, whether they support or not integrity checking by including checksum with each downloaded file.

A checksum is a small-sized block of data derived from another block of digital data, with the purpose of detecting errors that might happen during its transmission or storage. Checksums are reliable to verify data integrity, but not suitable for verifying data authenticity.

In terms of authenticity checking, a very common type of attack is the man-in-the-middle attack. In the man-in-the-middle model, an attacker can control the network between the user and the package repository.

Athalye et al. [66] conducted a security analysis on CPAN, Pearl package manager, which at that moment had around 140,000 modules³. The authors stated that they could have infected users' machines by uploading in the repository a malware they wrote, able to erase user's file system. Cappos et al. [32] identified security issues in eight different popular package managers [32], APT [46], apt-rpm [67], pacman [68], Portage [50], Slaktool [69], urpmi [44], YaST [42], and YUM [43]. These eight package managers present different security models, from the least secure to the most secure one:

- no security (e.g. pacman, slaktool),
- cryptographic signatures embedded with package (e.g. YUM and urpmi),
- signatures detached on package metadata (e.g. Portage),
- signatures on the root metadata (e.g. apt, apt-rpm, and YaST).

Even the most secure package managers presented their vulnerabilities, and ignoring security mechanisms could compromise a lot of clients.

A similar approach was done in [70], where the authors proposed an approach for detecting suspicious package updates in the Node.js/npm environment using an automated machine learning technique. This approach was able to confirm malicious packages and it could have potential to detect suspicious packages in real time, once they are available in the repository.

For RPKG, the security aspect will be addressed most likely in the future. At the moment, the repository is restricted to CERN's users only, therefore the problem of having suspicious packages is less significant, since CERN's users are signing a code of conduct. Nonetheless, having a security mechanism is a significant and valuable feature, that would allow RPKG to be used outside of the CERN network.

³it now has 209,245 modules in 43,467 distributions, written by 14,273 authors.

RPKG: RADE Package Manager

Contents

4.1	The requirements	31
4.2	Installation of packages	32
4.3	Creation of packages	34
4.3.1	Source project in GitLab	34
4.3.2	From source project to binary package using build file	36
4.4	Decoupling of RADE	38
4.5	Supporting CERN's environment for CERN's users	39
4.6	Software architecture	39
4.6.1	rpkglib	40
4.6.2	oslib	47
4.6.3	GUI	47
4.7	Chapter conclusions	50

Chapter outline

In this chapter, technical aspects and implementation details of **RPKG** are discussed. The requirements are listed in [Section 4.1](#), the installation and creation processes explained respectively in [Section 4.2](#) and [Section 4.3](#). The **RADE** decoupling process is analysed in [Section 4.4](#). [Section 4.5](#) explains how **RPKG** can run on systems under both **TN** and **GPN** and on the three main operating systems. The software architecture is described in [Section 4.6](#), where details for the core library **rpkglib**, the library **oslib** responsible for guaranteeing the cross-platform feature, and for the **GUI** are given. In [Subsection 4.6.1.3](#), the dependency solver algorithm is presented. Since the chapter is quite dense of content, a chapter conclusion is provided, in [Section 4.7](#).

4.1 The requirements

From the problem description stated in [Chapter 1](#), and having surveyed the *state-of-the-art* package managers, proposed in [Chapter 3](#), we decided to develop a new package manager, **RPKG**.

RPKG manages and deploys LabVIEW code in the CERN environment and infrastructure. RPKG is designed to meet the following requirements:

1. it handles the full life cycle and deployment of packages (install, update, remove, and create), explained in [Section 4.2](#) and [Section 4.3](#);
2. it has the flexibility to support different and future versions of LabVIEW as well as non-LabVIEW packages ([Section 4.6](#));
3. it partitions RADE into separate packages that are handled individually ([Section 4.4](#));
4. it supports CERN's multitier environment ([Section 4.5](#));
5. it handles database and package repository locally ([Subsection 4.6.1.2](#) and [Subsection 4.6.1.3](#));
6. it is easy to operate from a user-friendly GUI ([Subsection 4.6.3](#)).

Python3 was chosen as the programming language for RPKG because of its widespread adaption and the possibilities in combining the strengths of LabVIEW-based and Python-based platforms. RPKG, as many other package managers, is written using an imperative programming paradigm, mostly because the imperative approach is more human readable, and it is easier to calculate algorithms' complexity. To keep track of package manager actions on the system, an SQLite local database is used. SQLite is a popular open source SQL database [71]. It can store an entire database in a single file, and all of the data can be stored locally without having to connect the database to a server.

For the package repository, the decision was to use Nexus [37], as it was done previously in the MTA team [9]. In this repository LabVIEW packages from NI and JKI repository will be stored to test the installation.

Package installation and creation processes are fully analysed and discussed, respectively in [Section 4.2](#) and [Section 4.3](#).

The LabVIEW package structure, described in [Subsection 2.4.3](#), and the source project structure, described in [Subsection 4.3.1](#), must be kept in mind since installation and creation processes strongly rely on them.

4.2 Installation of packages

The installation phase of a package interacts mainly with the Nexus repository, the SQLite local database, and the file management system, as it is shown in [Figure 4.1](#). Inputs required from the user are: the name and version of the package that the user would like to install, and the LabVIEW version meant to be used with the package. Firstly, the content of the repository and the content of the local database are read. The selected package is downloaded and unzipped. Then, the SPEC file must be read because dependencies and conflicts have to be checked. If any, dependencies

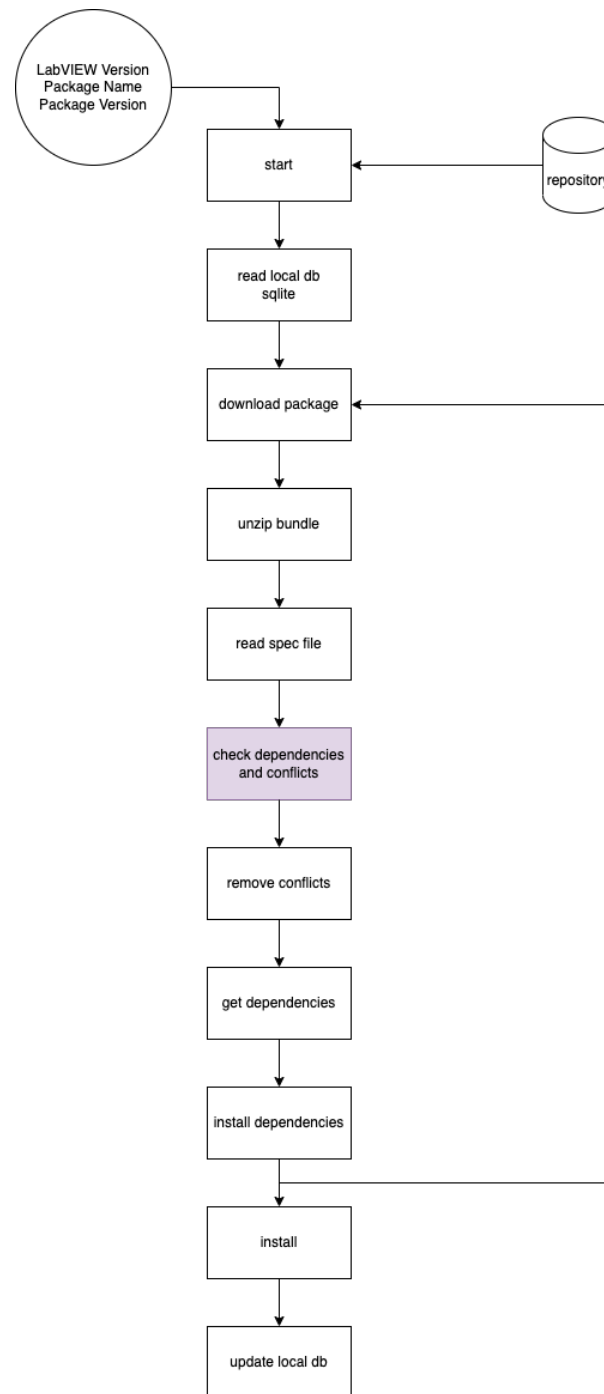


Figure 4.1: Installation of packages

must be installed and conflicts must be removed before installing the desired package. Once dependencies and conflicts have been solved, the package can be installed and the database must be updated, keeping track of dependencies and conflicts as well. The highlighted block in violet, in Figure 4.1, is responsible for the dependency

solver algorithm developed for RPKG, that will be discussed in Subsection 4.6.1.3. Installation paths are calculated based on the OS, where RPKG is running, on the LabVIEW application bitness and version where the package will be used, and also on the content of SPEC file specified under the “File Groups” section.

In Appendix A, a SPEC file is shown. Let’s suppose the user is using LabVIEW 2020 with 32 application bits on Windows. All the files under “File Group 0” will end up in the directory `C:\ProgramFiles(86x)\NationalInstruments\LabVIEW2020\user.lib\_OpenG.lib\file\file.llb`. The first part of the directory, `C:\ProgramFiles(86x)\NationalInstruments\LabVIEW2020`, is determined by the python file `oslib.py`, described in Subsection 4.6.2, responsible for calculating paths depending on the OS and LabVIEW version chosen, whereas the last part, `\user.lib\_OpenG.lib\file\file.llb`, is specified by the SPEC file content. Keeping the same OS and LabVIEW version, `.mnu` files in “File Group 1” and “File Group 2” will be copied in `C:\ProgramFiles(86x)\NationalInstruments\LabVIEW2020\menus\Categories\OpenG`, following the same logic explained before.

4.3 Creation of packages

Creating a package is slightly more complicated than installing one, since it interacts with LabVIEW and it must take how RADE is organised in terms of files structure. The decision is to distribute binary packages, such that the user will install packages already compiled, ready to be used. In this way, the compilation will be involved only once during package creation with CI and the user will not spend system resources in compiling. On the other hand, the repository storage will be bigger. In fact, depending on the LabVIEW version, LabVIEW application bitness, and operating system, different bundles will be created from the same LabVIEW source project stored in GitLab [72]. Supposing packages will be created from LabVIEW 2015 version, for a given source project, 28 different packages will be created: 7 for Windows 32bits, 7 for Windows 64 bits, 7 for Linux, and 7 for macOS. However, increasing the capacity of the repository is not a real issue since repositories nowadays can store a huge amount of data, and it is still preferable than distributing packages not compiled.

4.3.1 Source project in GitLab

In Figure 4.3, an example of how a project is stored in git is given. In the `src` folder, all the source files, that must be distributed, are stored. All the files will be distributed, but not all of them will be displayed in the palette of the block diagram. For example, the source project shown in Figure 4.3 contains around 60 VIs, but in the palette available for the end user from the block diagram, the functions available are only seven, as shown in Figure 4.4. This information is specified in the folder `palette`, that contains a `.rademnu` file, shown below.

```
1 [rade_utils.mnu]
2 name = "rade_utils"
```

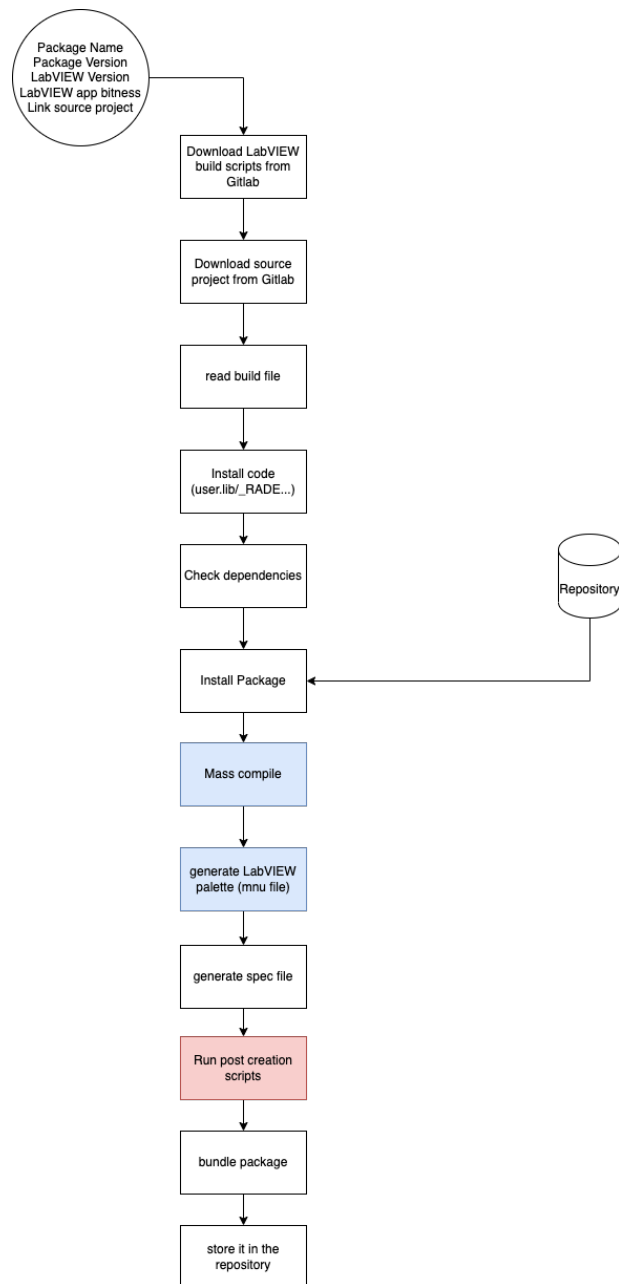



Figure 4.2: Creation of packages

```

3 icon = "<userlib>/_RADE/ljl/palette/rademnu/img/ utils .png"
4 dest = "<labview>/menus/ Categories/RADE"
5 synch = TRUE
6 synchpath = "<labview>/menus/ Categories/RADE/_rade_utils"
7 vis = ""
8
9 [ljl.mnu]
10 name = "JSON Parser"

```

< > ljl-master    

Name	Date Modified	Size	Kind
 ljl.build	10 August 2022 at 14:30	2 KB	NAnt Build File
▼  palette	Today at 19:17	--	Folder
 ljl.vi	10 August 2022 at 14:30	30 KB	LabVIEW...ocument
▼  rademnu	10 August 2022 at 14:30	--	Folder
>  img	10 August 2022 at 14:30	--	Folder
 ljl.rademnu	10 August 2022 at 14:30	599 bytes	Document
 utils.vi	10 August 2022 at 14:30	19 KB	LabVIEW...ocument
▼  src	Today at 19:16	--	Folder
▼  LabVIEW	10 August 2022 at 14:30	--	Folder
 Check value type.vi	10 August 2022 at 14:30	8 KB	LabVIEW...ocument
 Create.vi	10 August 2022 at 14:30	6 KB	LabVIEW...ocument
>  JSON array	10 August 2022 at 14:30	--	Folder
>  JSON boolean	10 August 2022 at 14:30	--	Folder
>  JSON null	10 August 2022 at 14:30	--	Folder
>  JSON number	10 August 2022 at 14:30	--	Folder
>  JSON object	10 August 2022 at 14:30	--	Folder
>  JSON string	10 August 2022 at 14:30	--	Folder
 JSON to map.vi	10 August 2022 at 14:30	19 KB	LabVIEW...ocument
>  JSON value	10 August 2022 at 14:30	--	Folder
 LV-JSON.lvproj	10 August 2022 at 14:30	3 KB	LabVIEW...object File
 lvjson.mnu	10 August 2022 at 14:30	2 KB	LabVIEW...ocument
 Modify.vi	10 August 2022 at 14:30	7 KB	LabVIEW...ocument
 Read.vi	10 August 2022 at 14:30	7 KB	LabVIEW...ocument
 test FESA to VIRTUAL.vi	10 August 2022 at 14:30	11 KB	LabVIEW...ocument
 Test modification.vi	10 August 2022 at 14:30	7 KB	LabVIEW...ocument

Figure 4.3: Example of a source project organisation in GitLab: LabVIEW JSON Library

```

11 icon = "<userlib>/_RADE/ljl/palette/rademnu/img/ljl.png"
12 dest = "<labview>/menus/Categories/RADE/_rade_utils"
13 synch = FALSE
14 vis = "ljl/JSON value/Import JSON.vi,ljl/JSON value/Export JSON.vi,ljl/
    JSON value/Modify value (poly).vi,ljl/JSON value/New value (poly).
    vi,ljl/JSON value/Read value (poly).vi,ljl/JSON value/is null.vi,
    ljl/JSON value/Read value type.vi"

```

In this file, in this case `ljl.rademnu`, it is specified that the library JSON Parser [73] (also called `ljl` library) will be contained inside the folder “`rade_utils`” in the LabVIEW palette and it specifies where “`rade_utils`” icon is located. Then, inside “`rade_utils`” folder, there will be `ljl` folder, from where the end user could use 7 different functions, or VIs, expressed in line 14, as it is also shown in Figure 4.4.

4.3.2 From source project to binary package using build file

Now, let’s suppose there is a LabVIEW developer writing a new library for the RADE framework. Once the developer has finished the library, the source project

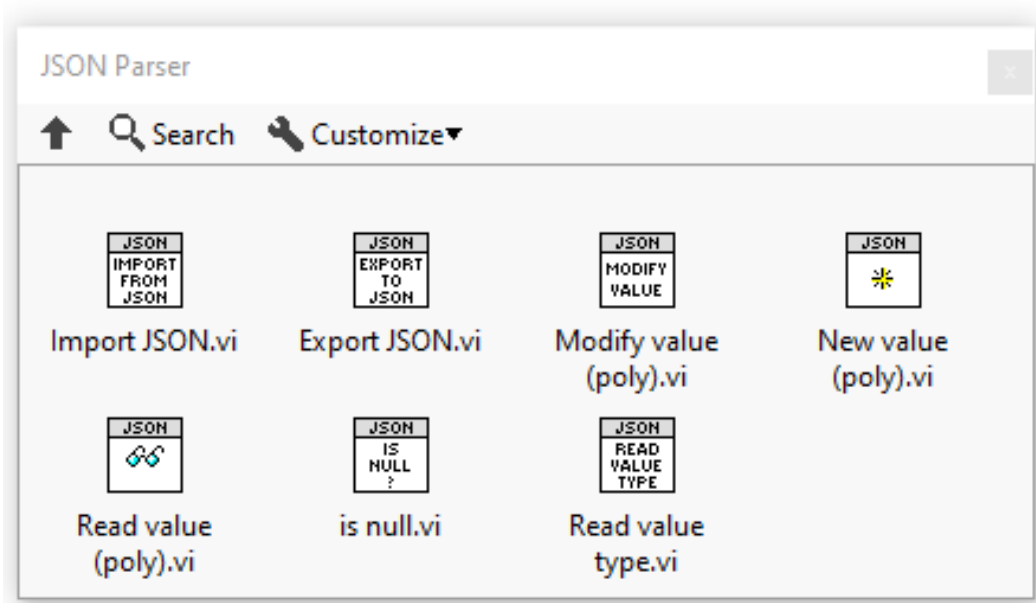


Figure 4.4: LabVIEW JSON Library

will be stored in GitLab [72]. A source project is stored with an “old” LabVIEW version (e.g. LabVIEW 2013) in GitLab, since LabVIEW is not forward compatible. In fact, a source project written in LabVIEW 2021 cannot be compiled by LabVIEW 2020, but can be the other way around. Therefore, MTA’s projects are usually written in LabVIEW 2015, sometimes even with an older version, such that the same source project can be compiled with newer LabVIEW versions or opened without any problem.

Instead of bundling this library within RADE, the developer would like to create a package out of it.

From a source project to a package, the key element is to use a build file. An example of a build file for the library “JSON Parser”, mentioned before in Subsection 4.3.1, is attached in Appendix B. By combining the build file with a source project, the package manager will know how to compile and bundle, creating a package out of the source project.

As can be seen in Appendix B, a build file contains information regarding the package and its functionalities, about the platform requirements (e.g. on which OS a particular package can be used and which LabVIEW version), and dependencies and conflicts (usually specified in a requirement.txt file). In the bottom part of the file, file groups are listed. The first file group usually concerns source files.

RPKG will make a copy of source files from GitLab to the real folder on the system. Then, it will do the same with files present in the palette folder, and, in case any, it will install dependency files as well. For the sake of completeness, .rademnu file, explained in ‘Subsection 4.3.1, is attached to the build file as well.

Once the build file is read, then RPKG will copy the source files in the right Lab-

VIEW directory. If there are any dependencies, RPKG will install them from the repository. Once source files are in the correct LabVIEW directory and dependencies are installed, it is time to mass compile. At this moment, RPKG will execute a LabVIEW VI responsible for that. The script to execute this function from Python is shown in Appendix D and the LabVIEW VI is attached in Appendix E. Once mass compilation and generation of .mnv files are completed, it is time to generate the SPEC file, which is necessary for RPKG, and in general for all package managers, to install the package later on. The SPEC file is very similar to the build file with the difference that all the installation files and .mnv files will be explicitly written. The SPEC file generated for the “JSON Parser” library is attached in Appendix C. Finally, all the main steps have been processed. The compiled code will be bundled together with the SPEC file, and stored in the repository.

4.4 Decoupling of RADE

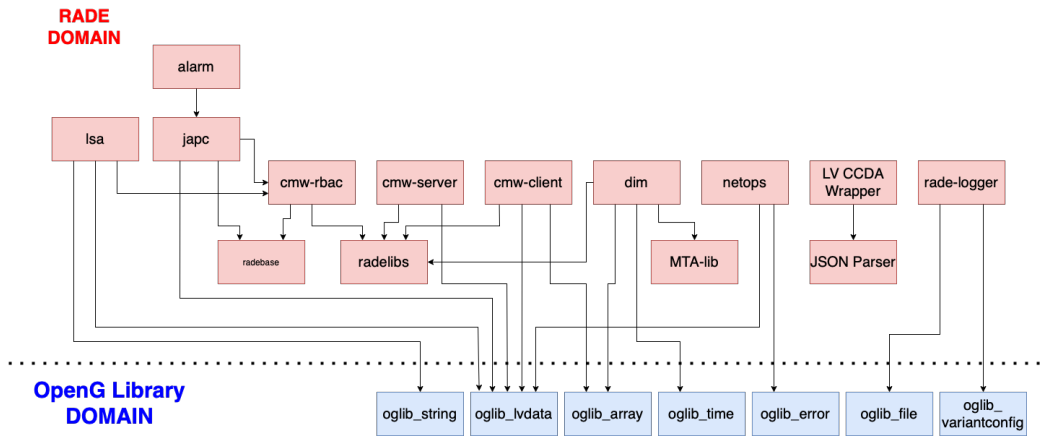


Figure 4.5: Dependencies within RADE framework

By following the package creation process and creating packages out of the RADE framework, the framework decoupling is achieved. Since the RADE framework is quite complex in terms of dependencies, the idea is to create a package out of RADE on a system where any other LabVIEW package is installed. Then, the package will be likely broken since its dependencies are not on the system. The next step is to go through the errors that LabVIEW is listing, and understand the dependencies of that particular library previously created. Once dependencies are figured out, they must be included and written in the build file stored on GitLab.

This process has to be done manual for the first time, because dependencies of every single libraries were not specified.

From all RADE libraries¹, we focus on the following ones: alarm, japc, CMW-rbac, CMW-client, CMW-server, isa, dim, LabVIEW CCDA Wrapper, radebase, radelibs,

¹They are stored in this repository: <https://gitlab.cern.ch/rade/rade-core/libs>

JSON Parser, netops, MTA-lib, and rade-logger. They take care mainly of communication between devices exposed to TN, handling files, LHC alarm system, and they are the most used libraries within the RADE framework.

The dependency graph within RADE libraries is shown in Figure 4.5. In this graph, RADE packages are highlighted in red, whereas packages from OpenG library [74] are in blue. Therefore, it can be seen that RADE packages rely on OpenG ones. In the graph, the OpenG dependencies are not specified, because the focus is on RADE's dependencies. For example, oglib_file depends on oglib_error and the dependency solver algorithm, explained in Subsection 4.6.1.3, knows how to handle it. The graph has to be interpreted in the following way: the library alarm depends on japc, which depends on cmw-rbac, openg_lvdata, and radebase.

By finding out dependencies within RADE framework and specifying them in the build file of each library, packages can be created and then stored in the Nexus repository. Once there, they will be ready to be installed from other users and the dependency solver algorithm will take care of all the dependencies to be installed, or conflicts to be uninstalled, from the system.

By having found dependencies within RADE, the users will not need to install the whole RADE framework, but only the package they want to use and the dependencies.

4.5 Supporting CERN's environment for CERN's users

CERN features a multitier environment and CERN's users work with all the three main operating systems. To make sure that RPKG can run on all the three main operating systems, a library (oslib in Subsection 4.6.2) is developed. Regarding the multitier environment, the application must support Linux, as the computers exposed to the TN are Linux based and the repository must be accessed under that network. In other words, users must be able to download packages from the Nexus repository when their system is running on the TN.

To do that, the RADE Nexus repository [5] is handled by an application on OpenShift [75], where a project to handle the repository is requested. After this, a request to the IT CERN Security department is made to allow the repository to be exposed to the TN, such that it is trusted and users can download packages from there.

4.6 Software architecture

The software structure of RPKG project is shown in Figure 4.6. The arrows represent the python imports that from one file are calling the others, for example main.py importing gui.py. The idea behind this software architecture was to develop a library that was representing the core functionalities of RPKG, called rpkglib, and to build on top of it a user-friendly GUI. In this way, the decoupling between the core functions of RPKG and the GUI was achieved. Having rpkglib decoupled from the GUI brings the following benefits:

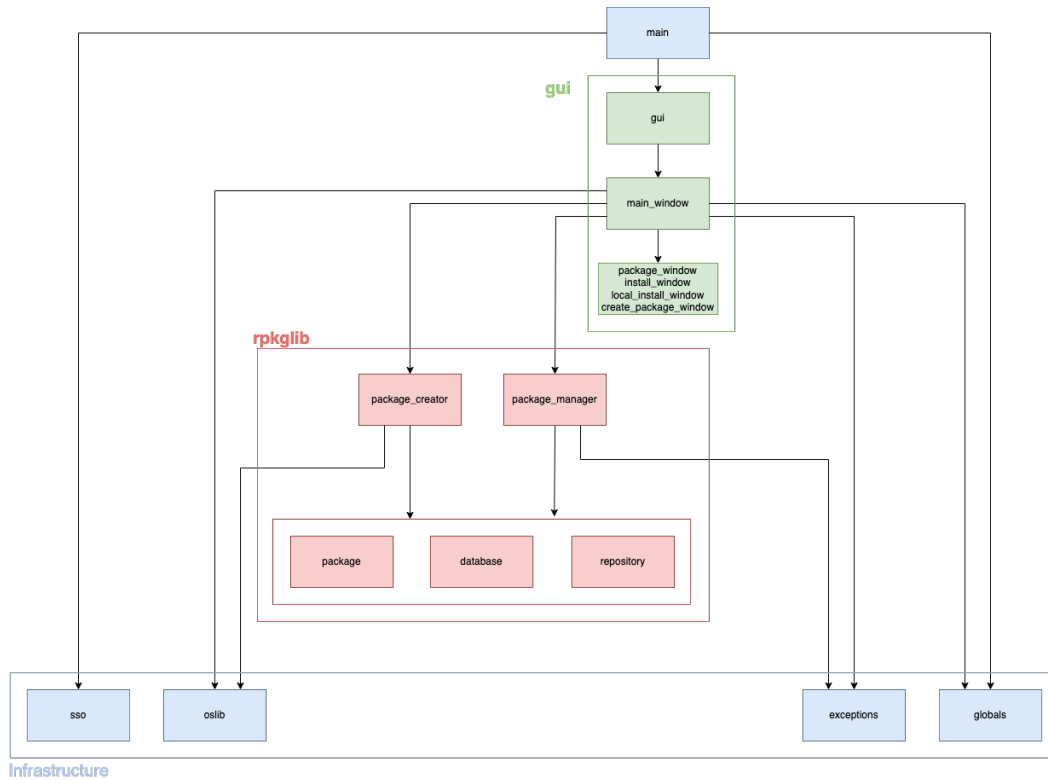


Figure 4.6: RPKG software architecture

- faster debugging, since it is easier understanding and localising in which part of the code there is the issue;
- the ease of changing the GUI without changing the core of the application, being `rpkglib`;
- having a compartmentalised and clear code, in terms of software responsibilities.

4.6.1 `rpkglib`

`rpkglib` is the Python library that represents the core of `RPKG`. It takes care of the main building blocks of the package management, and it makes use of the concept of class inheritance. The reason behind using class inheritance is to develop a package manager that could have been as flexible as possible.

For example, one of the parents classes is `Database` and its child is `SQLiteDatabase(Database)`. If, in the future, developers of the package manager will decide to move from SQLite to another type of database (e.g. Oracle), the parent class will still work and it will be only a matter of re-implementing the interfaces to the new type of database. The same principle is applied to `package` and `repository`, shown in Figure 4.6.

`rpkglib` is structured in five different files: `package.py`, `database.py`, `repository.py`, `package_manager.py`, and `package_creator.py`.

4.6.1.1 `package.py`

The class `Package` is the parent class that represents a general package and allows to perform common actions such as:

- extract the SPEC file from a package;
- read information from the SPEC file, such as the OS compatible with that package, or the description of the package in terms of functionalities;
- get dependency and conflict information based on the SPEC file content;
- install a package by copying its installation files to the correct path.

The class `VIPackage(Package)` is a child of `Package` and it handles the installation of LabVIEW packages. In the future, if `RPKG` will handle different packages, other child classes are required to be implemented according to the specifications coming from the parent class. It is important to mention that this class is responsible for getting direct dependencies from a package, so the dependency solver algorithm is not developed within this class.

4.6.1.2 `database.py`

The `Database()` class represents the database used to store all the information about what has been installed on the local client system. Looking at [Figure 4.7](#), where `package d` depends on `package e` and `package h`, the database provides the following functionalities:

- it initialises the database, in case it does not exist;
- it reads the packages which are already installed;
- it can add packages to the database which have been installed, and modify the ones that have been upgraded;
- it can remove packages;
- given `package d`, it gives as output all the direct packages that it depends on, which are `package e` and `package h` in [Figure 4.7](#), and this list will be called *children list*;
- given `package d`, it gives as output the dependency list of successors that package depends on, which are packages `e`, `h`, `f`, and `g` in [Figure 4.7](#);
- given `package d`, it gives as output the packages that directly depends on that package, in the example package `b` and `c`, and this list will be called *parent list*;

- given package **d**, it gives as output the ancestor tree, in the example packages **b**, **c**, and **a**.
- given a package name, it tells whether that package is installed or not;
- given a package name, it tells all the installation files and their directory on the system;
- given a package **foo**, it provides the list of packages that will be uninstalled together with **foo**.

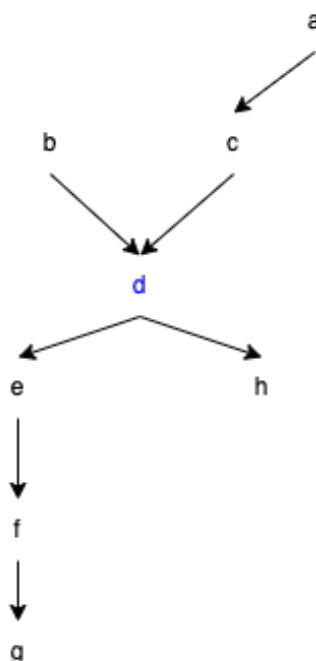


Figure 4.7: Dependency tree example for explaining database functionalities

As in most package managers, the database has a central role for RPKG. It is important that the database remains always synchronised with what is installed (or not) on the machine since most of the decisions are taken considering what it is stored in the database.

As the database used is SQLite, the child class `SQLite(Database)` was implemented. The database implemented consists of a relational database `.db` file with three different tables, called *packages*, *dependencies*, and *install_files*, shown in Figure 4.8, Figure 4.9, and Figure 4.10. The *packages* table keeps track of packages installed on a system. The entries registered are the name and version of the package, the SPEC file content converted as a dictionary, the repository from where the packages was taken, the installation mode, and the install path. Installation mode entry allows to distinguish whether a package was installed because of user selection or as

Table: packages						
	package_name	package_version	spec	install_from	install_mode	install_path
1	4smartmachines_lib_face_recognitio...	1.1.0.7	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
2	astronics_test_systems_inc_lib_astro...	1.2.0.9	("Package": {"name": ...	Nexus RADE Repository user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
3	4smartmachines_lib_face_recognitio...	1.1.0.7	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
4	atesystem_lib_claudie_xlsx	1.3.0.1	("Package": {"name": ...	Nexus RADE Repository user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
5	openg.org_lib_openg_toolkit	4.0.1.9	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
6	wireflow_ab_lib_wf_progressbar	1.0.2.56	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
7	oglib_appcontrol	4.0.0.5	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
8	oglib_array	4.0.0.9	("Package": {"name": "oglib_array", ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
9	oglib_boolean	4.0.0.7	("Package": {"name": "oglib_boolean...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
10	oglib_buttons	4.0.0.7	("Package": {"name": "oglib_buttons...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
11	oglib_comparison	4.0.0.3	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
12	oglib_dictionary	4.0.0.4	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
13	oglib_file	4.0.0.20	("Package": {"name": "oglib_file", ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
14	oglib_largefile	4.0.0.3	("Package": {"name": "oglib_largefile...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2019	
15	openg.org_lib_openg_toolkit	4.0.1.9	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
16	wireflow_ab_lib_wf_progressbar	1.0.2.56	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
17	4smartmachines_lib_hog_object_det...	1.0.0.10	("Package": {"name": ...	Nexus RADE Repository user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
18	4smartmachines_lib_poly_clip_system	1.0.1.2	("Package": {"name": ...	Nexus RADE Repository auto dependency	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
19	ai_visual_inspection_tool_by_pekat	1.0.0.3	("Package": {"name": ...	Nexus RADE Repository user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
20	autotestware_lib_human_readable_d...	1.0.0.1	("Package": {"name": ...	Nexus RADE Repository user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020	

Figure 4.8: packages table from SQLite database

Table: dependencies							
	package_name	package_version	install_path	dependency	dependency_name	dependency_version	dependency_eq
1	oglib_appcontrol	4.0.0.5	C:\Program Files (x86)\National ...	D	oglib_error	2.0	>=
2	oglib_appcontrol	4.0.0.5	C:\Program Files (x86)\National ...	D	oglib_lvdata	2.0	>=
3	oglib_array	4.0.0.9	C:\Program Files (x86)\National ...	D	oglib_error	2.0	>=
4	oglib_array	4.0.0.9	C:\Program Files (x86)\National ...	D	oglib_lvdata	2.0	>=
5	oglib_array	4.0.0.9	C:\Program Files (x86)\National ...	D	oglib_string	2.6	>=
6	oglib_file	4.0.0.20	C:\Program Files (x86)\National ...	D	oglib_appcontrol	2.10	>=
7	oglib_file	4.0.0.20	C:\Program Files (x86)\National ...	D	oglib_array	3.0.0	>=
8	oglib_file	4.0.0.20	C:\Program Files (x86)\National ...	D	oglib_error	2.3	>=
9	oglib_file	4.0.0.20	C:\Program Files (x86)\National ...	D	oglib_string	2.6	>=
10	oglib_largefile	4.0.0.3	C:\Program Files (x86)\National ...	D	oglib_error	2.0	>=
11	4smartmachines_lib_hog_object_det...	1.0.0.9	C:\Program Files (x86)\National ...	D	4SmartMachines_lib_HOG_Object_D...	1.0.0.9	=
12	4smartmachines_lib_face_recognition...	1.0.0.2	C:\Program Files (x86)\National ...	D	4SmartMachines_lib_Face_Recogniti...	1.0.0.2	=
13	scilogex_ms7_h550_pro_driver_system	1.0.0.26	C:\Program Files (x86)\National ...	D	ni_lib_nilm_api	2.0.0.3	>=
14	scilogex_ms7_h550_pro_driver	1.0.0.26	C:\Program Files (x86)\National ...	D	ni_lib_nilm_api	2.0.0.3	>=
15	scilogex_ms7_h550_pro_driver	1.0.0.26	C:\Program Files (x86)\National ...	D	SCIOGEX_MS7_H550_Pro_Driver_sy...	1.0.0.26	=
16	livewires_automation_ltd_lib_structu...	1.0.0.6	C:\Program Files (x86)\National ...	D	LiveWires_Automation_Ltd_Lib_Struct...	1.0.0.6	=
17	smi_lib_seesv_labview_api_i	1.0.1.20	C:\Program Files (x86)\National ...	D	smi_lib_seesv_labview_api_i_system	1.0.1.20	=
18	oglib_buttons	4.0.0.7	C:\Program Files (x86)\National ...	C	ogctl_buttons	2.3	>=
19	avgarde_systems_wireless_engineeri...	1.0.0.11	C:\Program Files (x86)\National ...	D	ni_lib_tplat	15.0.3.43	>=
20	ni_lib_tplat	15.0.3.43	C:\Program Files (x86)\National ...	C	ni_lib_third_party_licensing_and_acti...	1.0.0.24	>=

Figure 4.9: dependencies table from SQLite database

a dependency, whereas the install path is important to understand for which **LabVIEW** version the packages was installed for.

The *dependencies* table allows to keep track of dependencies. For example, by looking at [Figure 4.9](#), `oglib_appcontrol` package with version 4.0.0.5 depends on `oglib_error` and `oglib_lvdata`, and both of them must be with a version greater than 2.0.

The *install_files* table registers all the files installed for a package and in case of uninstallation, it will take the information from here.

4.6.1.3 repository.py

The class `Repository` is the parent class, representing the repository for storing and getting packages. Two children classes are implemented:

Table: install_files

	file_path	package_name	package_version	install_path	orphan
	Filter	Filter	Filter	Filter	Filter
1	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
2	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
3	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
4	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
5	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
6	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
7	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
8	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
9	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
10	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
11	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
12	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
13	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
14	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
15	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
16	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
17	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
18	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
19	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0
20	C:\Program Files (x86)\National Instruments\LabVIEW ...	astronics_test_systems_inc_lib_astro...	1.2.0.9	C:\Program Files (x86)\National ...	0

Figure 4.10: install_files table from SQLite database

	package_name	package_version	spec	os	labview_version	bits
	oglib_	Filter	Filter	Filter	Filter	Filter
4	oglib_numeric	4.0.0.5	{'Package': {'name': "oglib_numeric" ...	Windows	LabVIEW 2020	32
5	oglib_picture	4.0.0.13	{'Package': {'name': "oglib_picture", ...	Windows	LabVIEW 2020	32
6	oglib_string	4.0.0.6	{'Package': {'name': "oglib_string", ...	Windows	LabVIEW 2020	32
7	oglib_string	4.1.1.16	{'Package': {'name': "oglib_string", ...	Windows	LabVIEW 2020	32
8	oglib_string	5.0.0.25	{'Package': {'name': "oglib_string", ...	Windows	LabVIEW 2020	32
9	oglib_time	4.0.1.3	{'Package': {'name': "oglib_time", ...	Windows	LabVIEW 2020	32
10	oglib_appcontrol	4.1.0.7	{'Package': {'name': ...	Windows	LabVIEW 2020	32
11	oglib_array	4.1.0.13	{'Package': {'name': "oglib_array", ...	Windows	LabVIEW 2020	32
12	oglib_boolean	4.0.0.7	{'Package': {'name': "oglib_boolean", ...	Windows	LabVIEW 2020	32
13	oglib_comparison	4.0.0.3	{'Package': {'name': ...	Windows	LabVIEW 2020	32
14	oglib_error	4.0.0.12	{'Package': {'name': "oglib_error", ...	Windows	LabVIEW 2020	32
15	oglib_error	4.2.0.23	{'Package': {'name': "oglib_error", ...	Windows	LabVIEW 2020	32
16	oglib_file	4.0.1.22	{'Package': {'name': "oglib_file", ...	Windows	LabVIEW 2020	32
17	oglib_largefile	4.0.0.3	{'Package': {'name': "oglib_largefile", ...	Windows	LabVIEW 2020	32
18	oglib_lvdata	4.1.0.16	{'Package': {'name': "oglib_lvdata", ...	Windows	LabVIEW 2020	32
19	oglib_lvdata	5.0.0.27	{'Package': {'name': "oglib_lvdata", ...	Windows	LabVIEW 2020	32
20	oglib_lvzip	4.0.1	{'Package': {'name': "oglib_lvzip", ...	Windows	LabVIEW 2020	32
21	oglib_md5	4.1.0.8	{'Package': {'name': "oglib_md5", ...	Windows	LabVIEW 2020	32
22	oglib_msgqueue	4.0.0.15	{'Package': {'name': ...	Windows	LabVIEW 2020	32
23	oglib_numeric	4.1.0.8	{'Package': {'name': "oglib_numeric", ...	Windows	LabVIEW 2020	32
24	oglib_string	4.1.0.12	{'Package': {'name': "oglib_string", ...	Windows	LabVIEW 2020	32
25	oglib_string	4.1.2.18	{'Package': {'name': "oglib_string", ...	Windows	LabVIEW 2020	32
26	oglib_time	4.0.0.2	{'Package': {'name': "oglib_time", ...	Windows	LabVIEW 2020	32
27	oglib_variantconfig	4.0.0.5	{'Package': {'name': ...	Windows	LabVIEW 2020	32
28	oglib_appcontrol	4.0.0.5	{'Package': {'name': ...	Windows	LabVIEW 2020	32
29	oglib_array	4.0.0.9	{'Package': {'name': "oglib_array", ...	Windows	LabVIEW 2020	32
30	oglib_array	4.1.1.14	{'Package': {'name': "oglib_array", ...	Windows	LabVIEW 2020	32

Figure 4.11: Repository content filtered for OpenG library

EOSRepository(Repository) and NexusRadeRepository(Repository). EOS is the disk storage system at CERN for large physics data, created for the LHC computing requirements. The repository used for this project will be Nexus with EOS

mounted on top. The repository is responsible for the following functionalities:

- providing the list of packages available on the repository;
- querying if a package is available or not in the repository;
- downloading a package or package SPEC file from the repository on the user system;
- getting package children and the whole dependency tree, involving the dependency solver algorithm;
- caching the whole repository into another `.db` file.

Caching the whole repository into a `.db` file and saving this `.db` file locally allows to save more time when interfacing with the repository, and this repository update on a local `.db` file should be done on a regular basis.

In [Figure 4.11](#), the repository content is saved into a `.db` file and it shows some packages from the `OpenG` library. The important information saved from the repository to this file are: the name of the package, the version of the package, the SPEC file of that package, the OS compatible with the package, the `LabVIEW` version in which the package is compiled for, and, in case the OS is Windows, the `LabVIEW` application bitness where the package can run on.

In [Appendix F](#), the dependency solver algorithm is provided. The logic behind can be explained with a simple example shown in [Appendix F](#), and looking at [Table 4.1](#), where the user wants to install `package a`. There are two lists: the list of packages to process, and the list of packages to install, which is empty at the beginning. The user selects `package a` to be installed, therefore it will be added to the list of packages to process. Then, the algorithm will check if the first element of the list of packages to process is in the packages to install list. If it is not, then the package will be added, otherwise not. Moreover, the algorithm will check the children of the packages and in case of any, it will prepend them to the list of packages to process. Looking at the example the logic is the following:

1. The user selects `package a` to be installed: `package a` is added to the list of packages to process and the list of packages to install is empty.
2. The algorithm will check if `package a` is already in the list of packages to install. In this case, it is not so it will append it to the list. Then, `package a` is removed from the list of packages to process and its children are prepended.
3. Now, the first package in the list packages to process is `package b`. It is not present in the list of packages to install so it will be added there, removed from the list packages to process and its children must be prepended in the latter list. Therefore, `package f` will be prepended.
4. This logic continues until `package c` and `package d` are left in the list of packages to process. They are already in the other list of packages to install,

so the algorithm will finish its execution, giving as output the list of packages depicted in step 7 of [Table 4.1](#).

The example provided is ignoring that packages and their dependency have also a version. Usually, the requirement can be of the type “greater than” (e.g. `package a-1.0 requires package b >= 2.0`) or “strictly equal to” (e.g. `package a-1.0 requires package b = 2.0`). The algorithm developed takes versioning into account and is included in [Appendix F](#).

The dependency solver algorithm developed starts at the root node and explores as far as possible along each branch before backtracking, therefore it is [DFS](#) inspired [\[35\]](#).

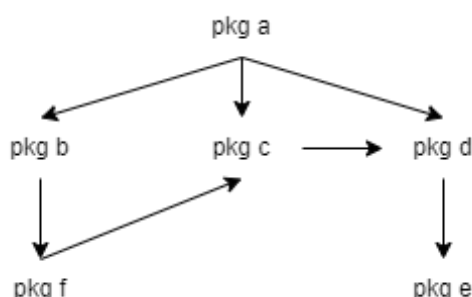


Figure 4.12: Dependency solver algorithm example

steps	packages to process	packages to install
0	A	/
1	B C D	A
2	F C D	A B
3	C C D	A B F
4	D C D	A B F C
5	E C D	A B F C D
6	C D	A B F C D E
7	/	A B F C D E

Table 4.1: Dependency solver algorithm example

4.6.1.4 `package_manager.py` and `package_creator.py`

`package_manager.py` is responsible for installing, updating, removing packages on a user system, whereas `package_creator.py` for creating them. They are call and use `VIPackage(Package)`, `SQLiteDatabase(Database)` and `NexusRadeRepository(Repository)`, executing the main steps schematised in [Figure 4.1](#) and [Figure 4.2](#). They have a higher level of awareness inside `rpkglib`, as [Figure 4.6](#) shows.

in the table. On the left of the table, the status of a package is shown. A status of a package can be installed, not installed, or updates available. Users can filter packages based on the name, or based on the status, or see which packages can be installed based on the LabVIEW version. In Figure 4.13, it can be also noticed that the system is detecting the number of LabVIEW versions installed and packages can be filtered according to this.

By clicking on a package, users can see the description of a package, shown in Figure 4.14, and see the versions available for that package. In this GUI page, the content of the SPEC file is shown, such that end users will get to know functionalities of a package and understanding if that package is exactly what they are looking for.

By clicking on the “install” button, users will have access to the installation

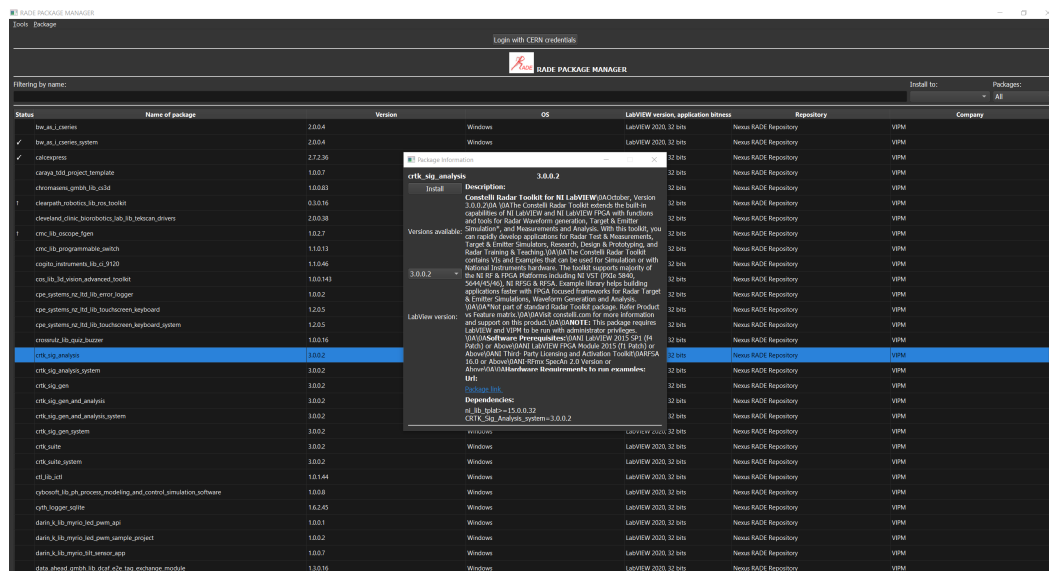


Figure 4.14: GUI Package information, included in Appendix G in a bigger format

page, having an overview of which packages are required and which are the ones to be installed. In Figure 4.15, it can be seen that the page is displaying all the dependencies needed, the ones that are already installed and those that have to be installed. Users may even decide to not include dependencies by un-checking the checkbox “Include dependencies”, even if not installing dependencies is highly not recommended.

From the main page, it can be seen that there are two menu bars: “Tools” and “Package”. From “Tools”, the page that will help users to create packages gets called, whereas from “Package”, users can install packages that they have locally and which are not stored in the repository. These two pages are shown respectively in Figure 4.16 and Figure 4.17.

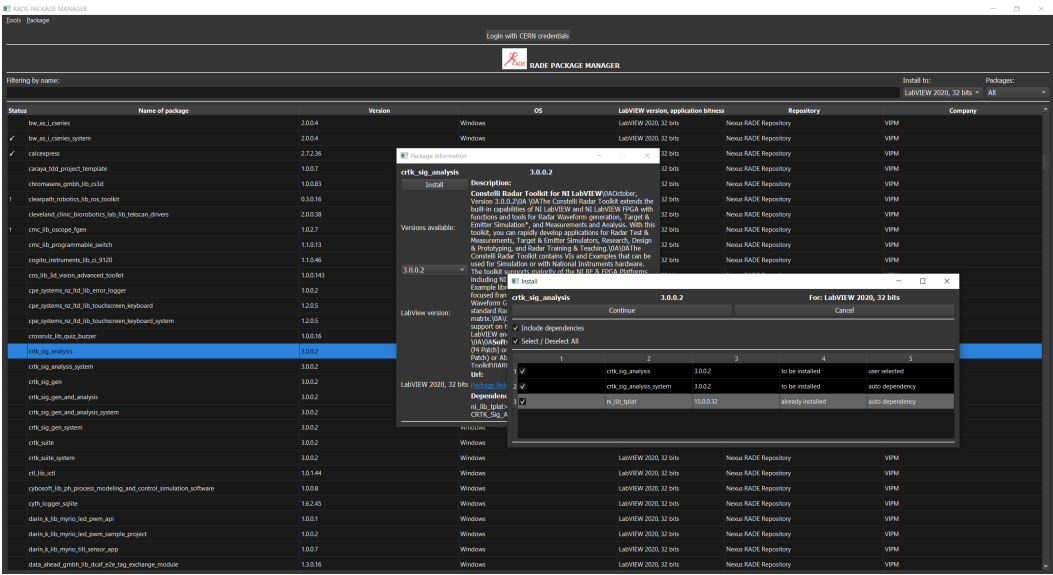


Figure 4.15: GUI Installation page, included in Appendix G in a bigger format

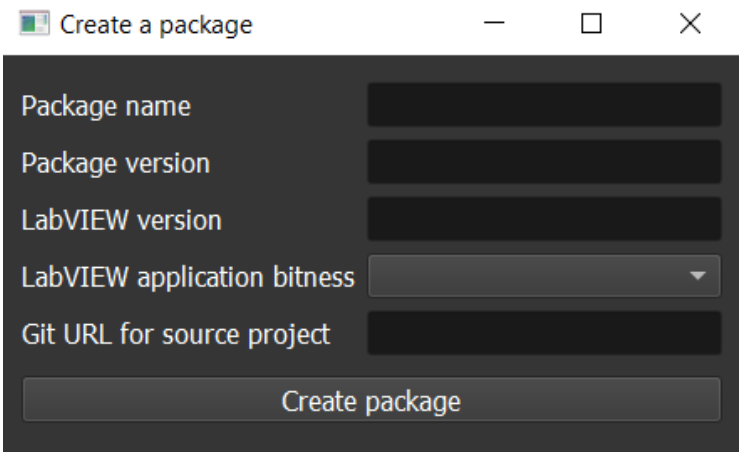


Figure 4.16: Menu bar of the GUI for creation of packages

4.6.3.1 CERN SSO from GUI

From the main GUI page, CERN users can log in using CERN Single Sign-On, as it is shown in Figure 4.18. By clicking on that graphical button, the sso.py script, shown in Figure 4.6, gets called. The protocol used is Open ID Connect, where a request to Single-Sign On is made to authenticate a user and then a token is sent back to the application. For this application, it is all handled by a python library called keycloak. Open ID Connect (OIDC) is a very common protocol and there are lots of libraries that can be used, and hence it was chosen to be used for RPKG. In addition to this, since RPKG is using CERN SSO, the application had to be registered at CERN.

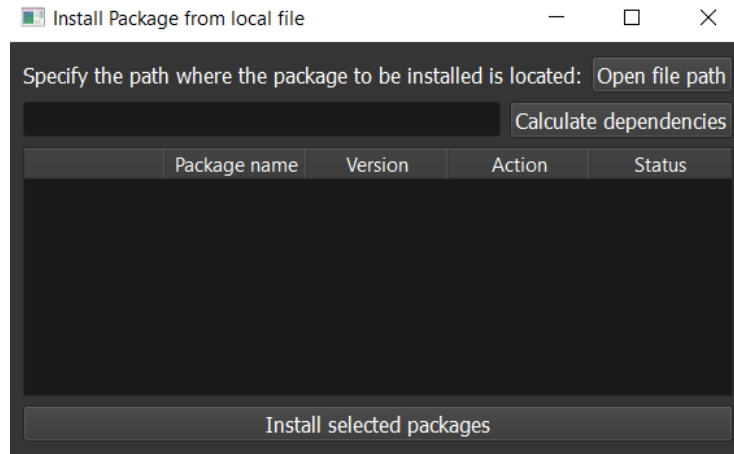


Figure 4.17: Menu bar of the GUI for local installation of packages

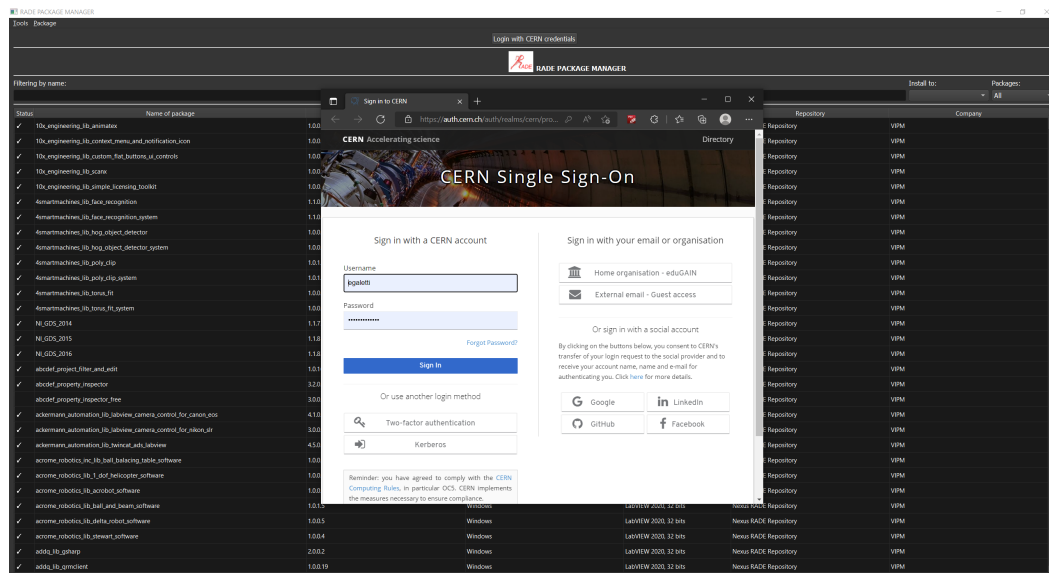


Figure 4.18: GUI CERN SSO, included in Appendix G in a bigger format

4.7 Chapter conclusions

In this chapter, the main features of RPKG are presented. In Section 4.2 and Section 4.3, the installation and creation processes are presented. In Section 4.4, the decoupling process of RADE is analysed. The key element was to find the dependencies that each RADE library relies on. After having found all of them, the first packages created are the ones with no dependencies, such that the next ones that are created can rely on them. In Section 4.5, it is explained how RPKG can run on devices under TN and how the cross-platform requirement is achieved. Software architecture and implementation details were given in Section 4.6, and the dependency solver algorithm is explained in Subsection 4.6.1.3. This algorithm was

developed *ad-hoc* in order to handle dependencies for RADE packages or packages written by NI or JKI. It is DFS inspired and many other package managers might use the same logic. It is indeed difficult to know which dependency algorithm is used by other package managers, but usually BFS [34] or DFS are used with imperative languages, whereas SAT solvers, mentioned already in Section 3.2, are used for pure functional languages.

In Subsection 4.6.3, the GUI is shown, and the main features are reported. The layout of RPKG's GUI follows the layout of other package managers, such as VIPM or NIPM package managers.

Evaluation

Contents

5.1	Dependency solver algorithm validation	53
5.2	Installation of packages validation	54
5.3	Creation of RADE packages validation	60
5.4	Cross-platform validation	62

Chapter outline

In this chapter, the **RPKG** application is tested and validated for the dependency solver algorithm, for both package installation and creation processes, and for the cross-platform requirement. Procedure and screenshots of the application are provided to show how the application works and how the requirements are satisfied. The focus for this evaluation is on functionality rather than performance, since **RPKG** does not have to be necessarily faster than **NIPM** [15] or **VIPM** [14]. Nevertheless, the starting time of **RPKG** application and installation process times are informally comparable with **NI** and **VI** package managers, and it would be difficult to say which application is faster or slower from an end-user perspective.

As future work for the evaluation part, it would be definitely interesting to time processes, for both **rpkglib** and **GUI**, to study where the bottlenecks of the application are and improve those execution times, perhaps by using some python libraries meant for that. An example would be to use **jit** compilation and apply the **jit** decorator to some functions such that they could be optimised by Numba's JIT compiler [76]. Another interesting part to time are related to data transfers and database queries, to see how to improve the input/output timing.

5.1 Dependency solver algorithm validation

RPKG's dependency solver algorithm is developed *ad-hoc*, instead of using already available SAT solvers.

To make sure that the dependency solver algorithm developed provides correct results, a comparison between **VIPM** [14], introduced in Table 1.1. Some **LabVIEW** packages available both on **VIPM** and **RPKG** repositories have been compared in terms of dependencies to check whether the same list of dependencies was provided

by both package managers. In Figure 5.1, the package **Certification Authority** 1.0.0.5 is shown in both package managers, **RPKG** on the left and **VIPM** on the right. As it can be seen, both package managers provide the same list of dependencies. Other five sample packages have been compared and the output list of dependencies is the same in all the cases, as can be seen in Figure 5.2 for the package **MGI File** 1.1.0.4, Figure 5.3 for the package **OpenG Toolkit** 4.0.1.9, Figure 5.4 for the package **OpenG File Library** 4.0.1.22, and Figure 5.5 for the package **Pane Relief** 2.0.3.1.

The packages **Certification Authority** 1.0.0.5 and **Pane Relief** 2.0.3.1 were chosen because they depend on different libraries (such as **JKI State Machine**, **Data manipulation**, **OpenSerializer**, **Open G**, and **MGI** libraries).

The packages **OpenG File Library** 4.0.1.22 and **MGI File** 1.1.0.4 were tested because they are widely used, whereas **OpenG Toolkit** 4.0.1.9 because it has quite long and complicated dependency list.

The algorithm developed for **RPKG** has been proved to work and to provide correct results for all the packages included in **VIPM** Repository from **JKI** and for **RADE** packages as well.

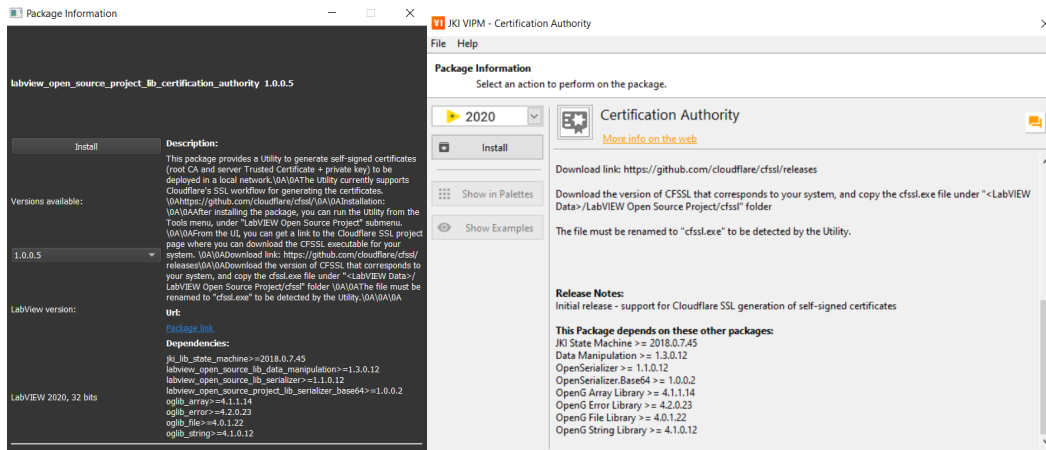


Figure 5.1: Certification authority 1.0.0.5

5.2 Installation of packages validation

In this section, the installation of packages will be validated and proved to be working. The first package to be installed is **oglib_error-4.2.0.23**, which does not have any dependency, as it is shown in Figure 5.6 and Figure 5.7. After the installation is completed, the application will mark the package as installed, as shown in Figure 5.8, and the database will add the specific entries for that, as shown in Figure 5.9 for the table *packages* and Figure 5.10 for the table *install_files*.

The database entry for the table *dependencies*, mentioned in Subsection 4.6.1.2, is not shown because it will be empty since the package **oglib_error-4.2.0.23** does

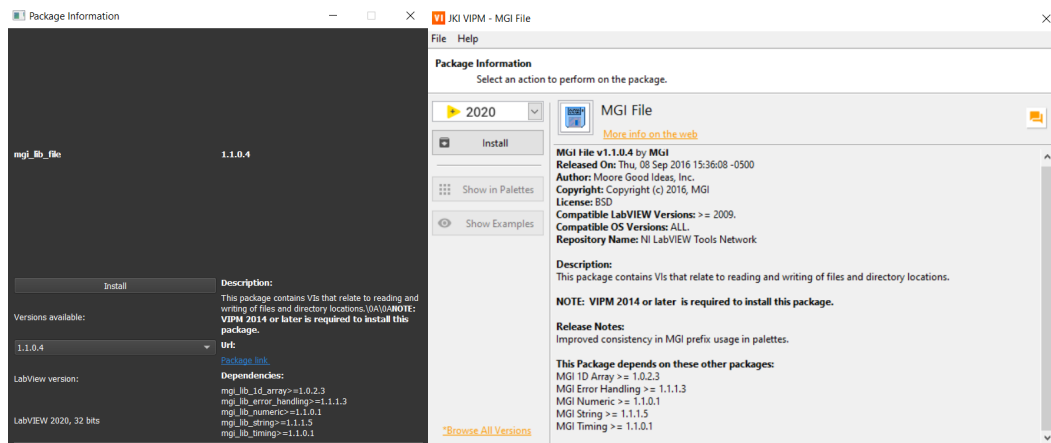


Figure 5.2: MGI File 1.1.0.4

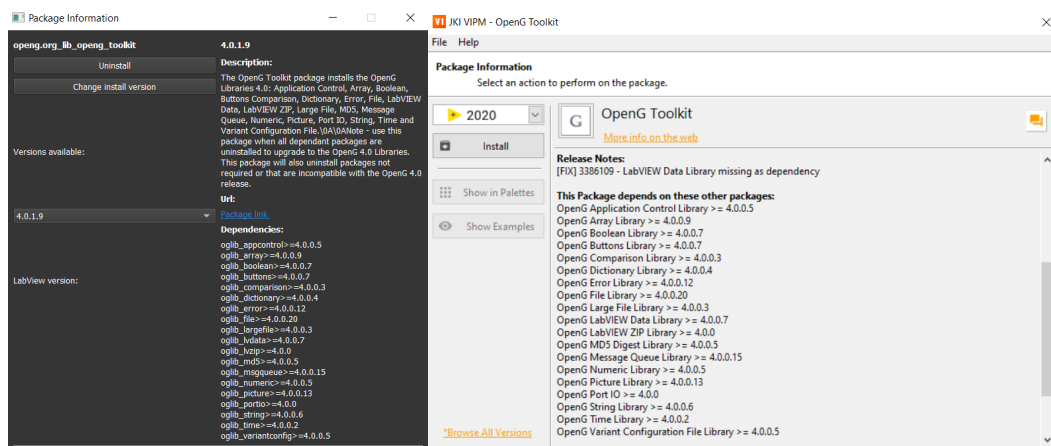


Figure 5.3: OpenG Toolkit 4.0.1.9

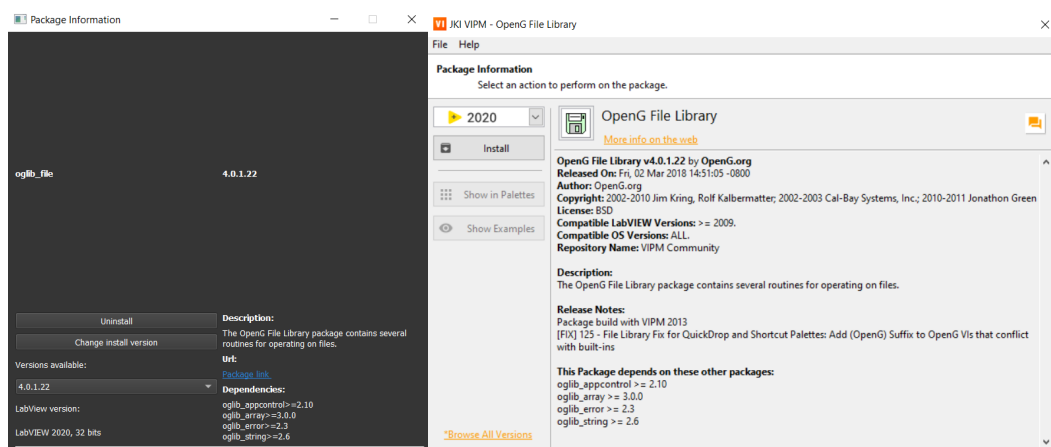


Figure 5.4: OpenG File Library 4.0.1.22

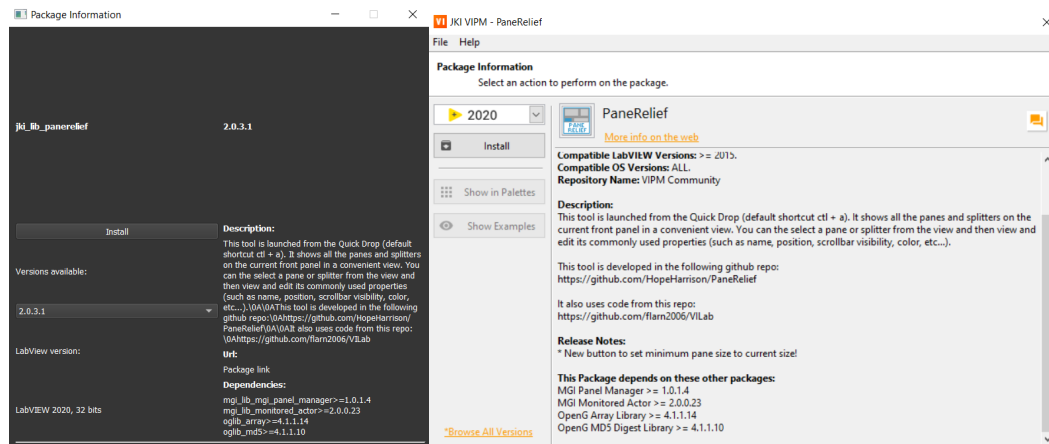


Figure 5.5: Pane Relief 2.0.3.1

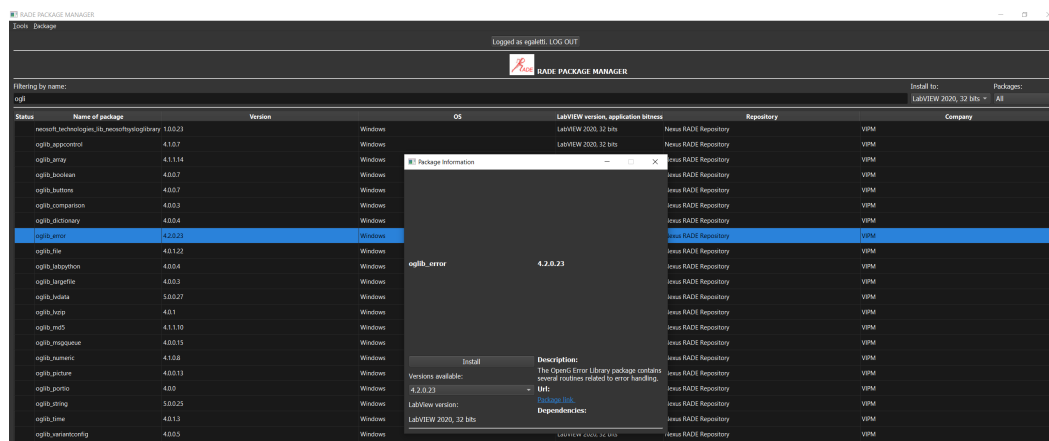


Figure 5.6: Package information page for oglib_error-4.2.0.23

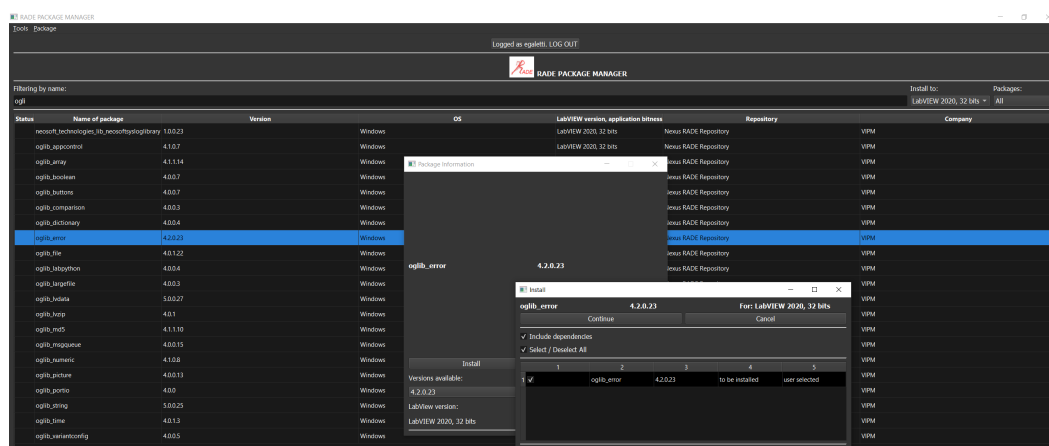


Figure 5.7: Installing oglib_error-4.2.0.23

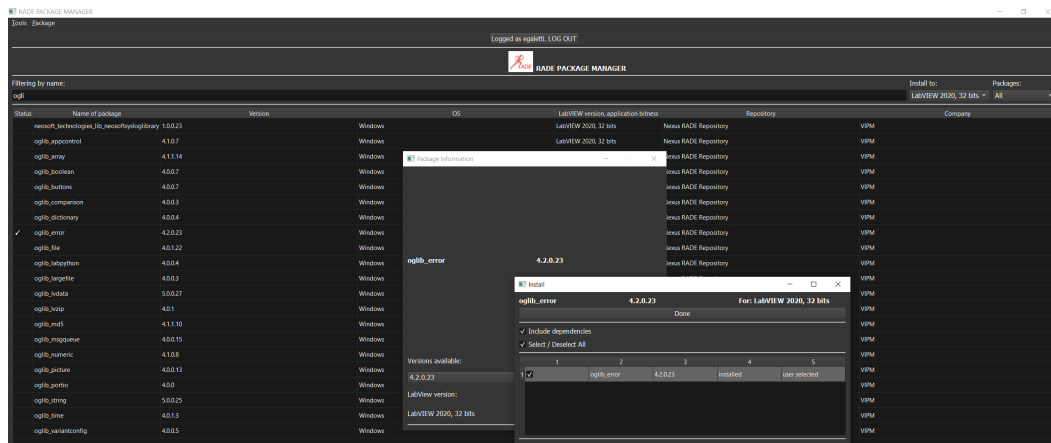


Figure 5.8: Installation completed for oglib_error-4.2.0.23

Database Structure Browse Data Edit Pragmas Execute SQL

Table: packages Filter in any column

	package_name	package_version	spec	install_from	install_mode	install_path
	Filter	Filter	Filter	Filter	Filter	
1	oglib_error	4.2.0.23	("P...	Nexus RAD...	user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020

Figure 5.9: Database entry for the table “packages” for oglib_error-4.2.0.23

Database Structure		Browse Data		Edit Pragmas		Execute SQL	
Table: Install_files  Filter in any column							
file_path		package_name	package_version	install_path		orphan	
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Build Error Cluster__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
2	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Case (Error IO)__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
3	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Clear All Errors__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
4	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib>Error Codes Ring ...	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
5	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib>Error Constant__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
6	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Filter Error Codes ...	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
7	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Filter Error Codes ...	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
8	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Filter Error Codes__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
9	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\Filtered Error Details - ...	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
10	C:\Program Files (x86)\National Instruments\LabVIEW 2020\user.lib\OpenG.lib\error\error.lib\VI Tree - error__ogtk.vi	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
11	C:\Program Files (x86)\National Instruments\LabVIEW 2020\menus\Categories\OpenG\functions_oglib_error.mnu	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0
12	C:\Program Files (x86)\National Instruments\LabVIEW 2020\menus\Categories\OpenG\dir.mnu	oglib_error	4.2.0.23	C:\Program Files (x86)\National ...			0

Figure 5.10: Database entry for the table “install_files” for oglib_error-4.2.0.23

not have dependencies. From Figure 5.10, it can be seen that 12 installation files are installed: ten of them are LabVIEW VI files and two are .mnu files.

After having installed oglib_error-4.2.0.23, the package oglib_file-4.0.0.22 is installed. Since oglib_error-4.2.0.23 is already installed on the system, it will be greyed out from the list of packages that have to be installed, as shown in Figure 5.11. After the installation of oglib_file-4.0.0.22 and its dependencies, the application will display that installation was successful, as it is shown in Figure 5.12. After installation of oglib_file-4.0.0.22, all the packages are registered in the database, as Figure 5.13 shows. The user can check in the installation path if files are there, but the “final proof” is to open the specific version of LabVIEW the packages are meant for, LabVIEW 2020 in this case, and check if functions are

there. This is shown in Figure 5.14. Before installing a package, the package

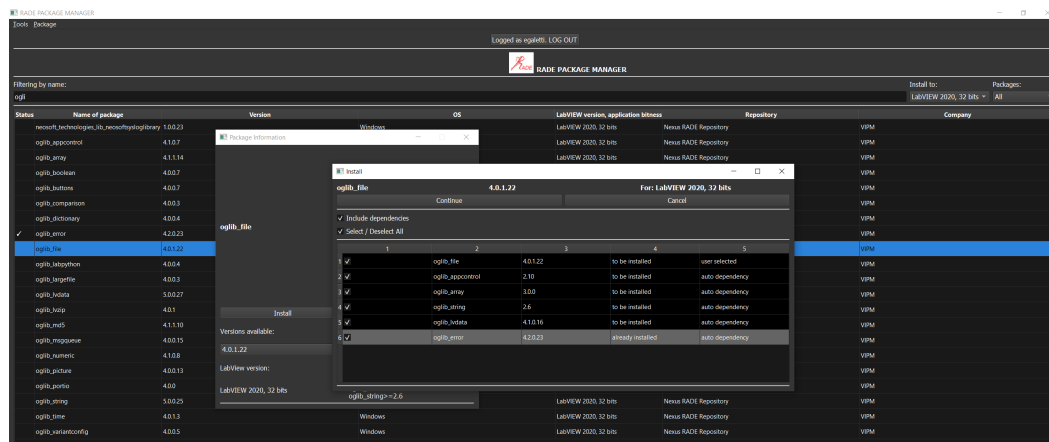


Figure 5.11: Installing oglib_file-4.0.0.22

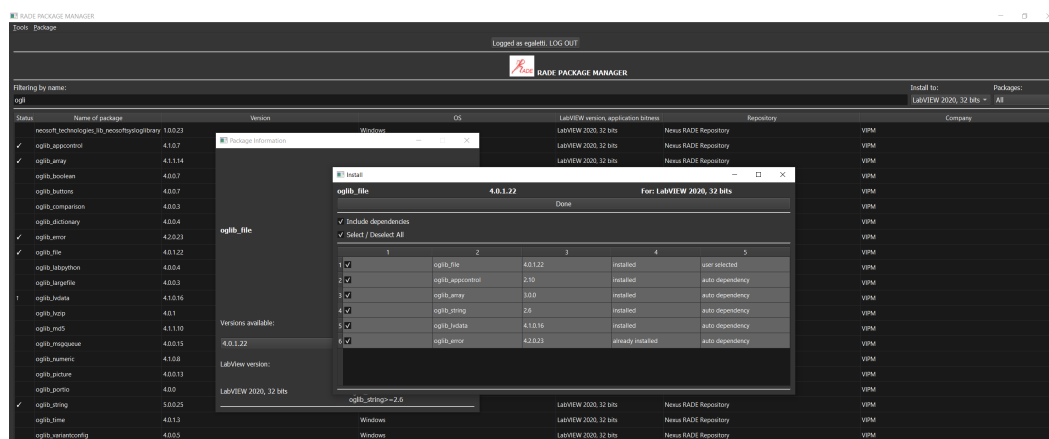


Figure 5.12: oglib_file-4.0.0.22 and its dependencies installed

Database Structure						
Table: packages						
package_name	package_version	spec	install_from	install_mode	install_path	
Filter	Filter	Filter	Filter	Filter	Filter	Filter
1 oglib_error	4.0.0.23	("Package": {"name": "oglib_error", "version": "4.0.0.23", "release": "", ...	Nexus RAD...	user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
2 oglib_appcontrol	4.1.0.7	("Package": {"name": "oglib_appcontrol", "version": "4.1.0.7", "release": "...	Nexus RAD...	auto ...	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
3 oglib_array	4.1.1.14	("Package": {"name": "oglib_array", "version": "4.1.1.14", "release": "", ...	Nexus RAD...	auto ...	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
4 oglib_string	5.0.0.25	("Package": {"name": "oglib_string", "version": "5.0.0.25", "release": "", ...	Nexus RAD...	auto ...	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
5 oglib_hdata	4.1.0.16	("Package": {"name": "oglib_hdata", "version": "4.1.0.16", "release": "", ...	Nexus RAD...	auto ...	C:\Program Files (x86)\National Instruments\LabVIEW 2020	
6 oglib_file	4.0.1.22	("Package": {"name": "oglib_file", "version": "4.0.1.22", "release": "", "id": ...	Nexus RAD...	user selected	C:\Program Files (x86)\National Instruments\LabVIEW 2020	

Figure 5.13: Database entry for the table “packages” for oglib_file-4.0.0.22 and its dependencies

manager will check if the package is well formatted or not, following package format guidelines explained in Subsection 2.4.3. If this is not the case, the RPKG application will show a generic error at the moment, as reported in Figure 5.15. This

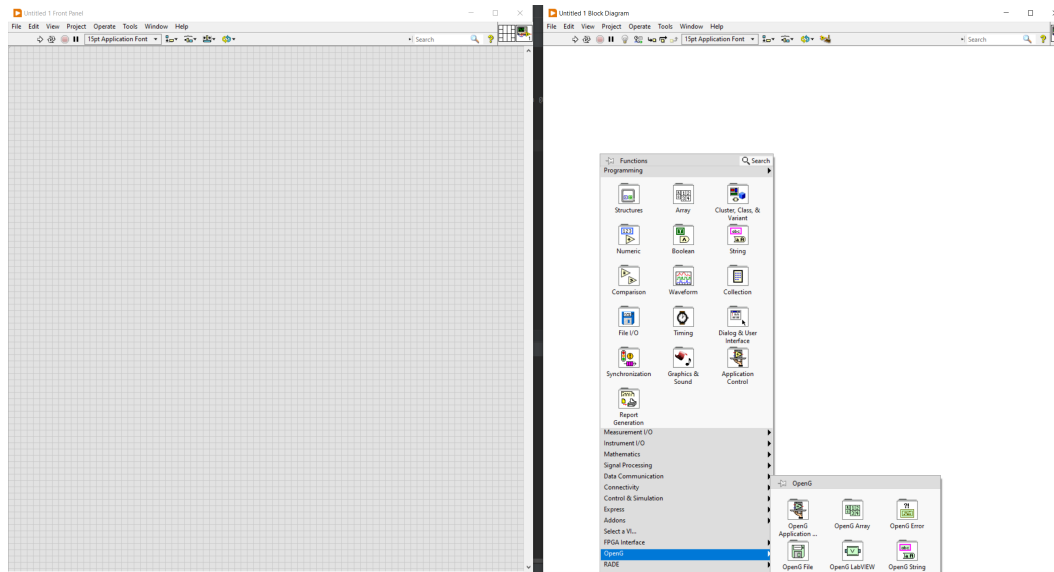


Figure 5.14: Open G palette in LabVIEW 2020

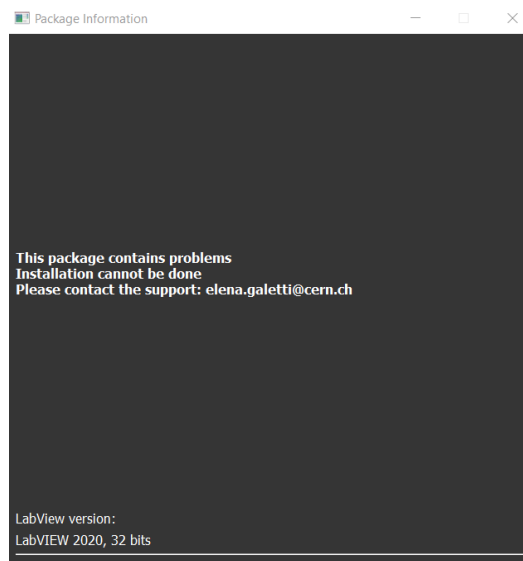


Figure 5.15: RPKG page detecting that the package is badly formatted

can be improved in the future, providing the end user with more details regarding the faulty package. Uninstalling and updating processes are part of the package management, therefore also those features have been tested using the same process of installation. The test procedure is the same as installing, hence the status of the local database and the [LabVIEW](#) application status has been checked.

In general, the installation and un-installation processes were validated by [MTA](#) team and throughout the whole development process, in coding sessions and test phases.

5.3 Creation of RADE packages validation

This test aims to validate the package creation process, discussed in [Section 4.3](#), starting from a source project and its build file. The package that will be created is from the source code of the library JSON Parser, mentioned already in [Subsection 4.3.1](#) and included into RADE framework. Since the goal is to create a package out of a library that is integrated in the RADE, a virtual machine where RADE has never been installed before will be used. A package will be created, uploaded to the repository, and then installed on the virtual machine. In this particular test, the package `ljl-1.0` is created for LabVIEW 2020 for Windows 32 bits application by following the steps described in [Section 4.3](#), and the result is a package, shown in [Figure 5.16](#). Right after the package is uploaded to the Nexus repository

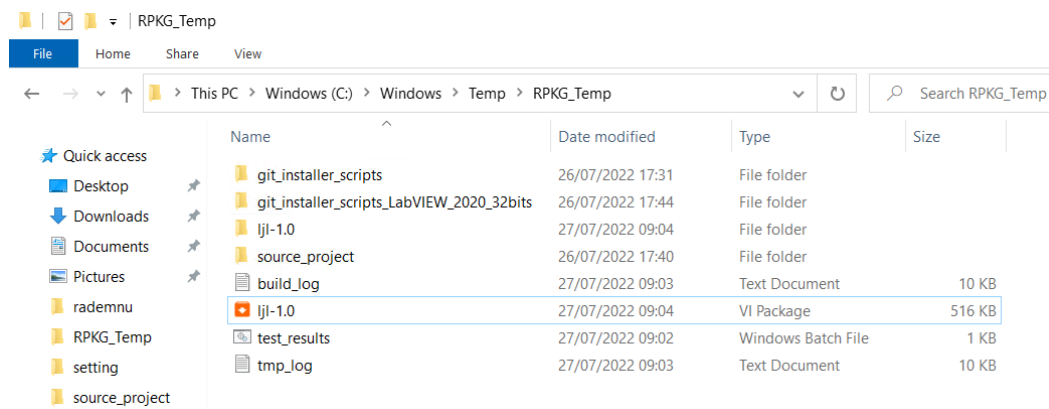


Figure 5.16: Package `ljl-1.0` created

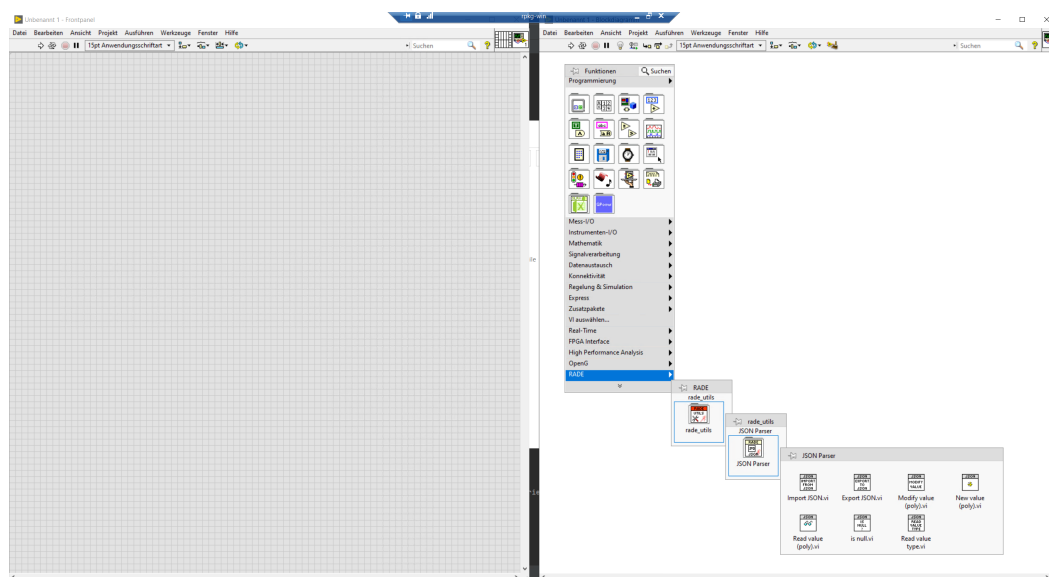


Figure 5.17: Package `ljl-1.0` inside RADE framework in LabVIEW 2020

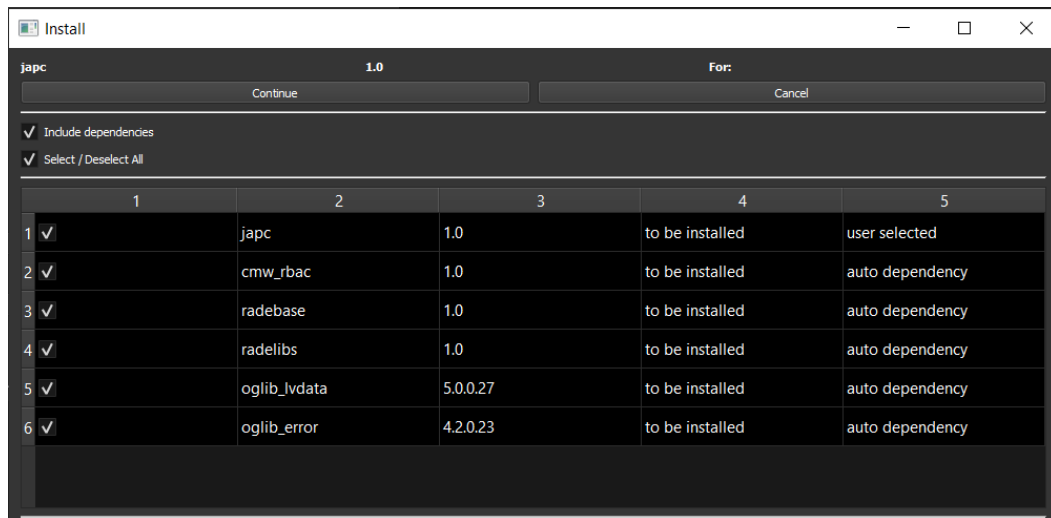


Figure 5.18: Dependencies for japc to be installed

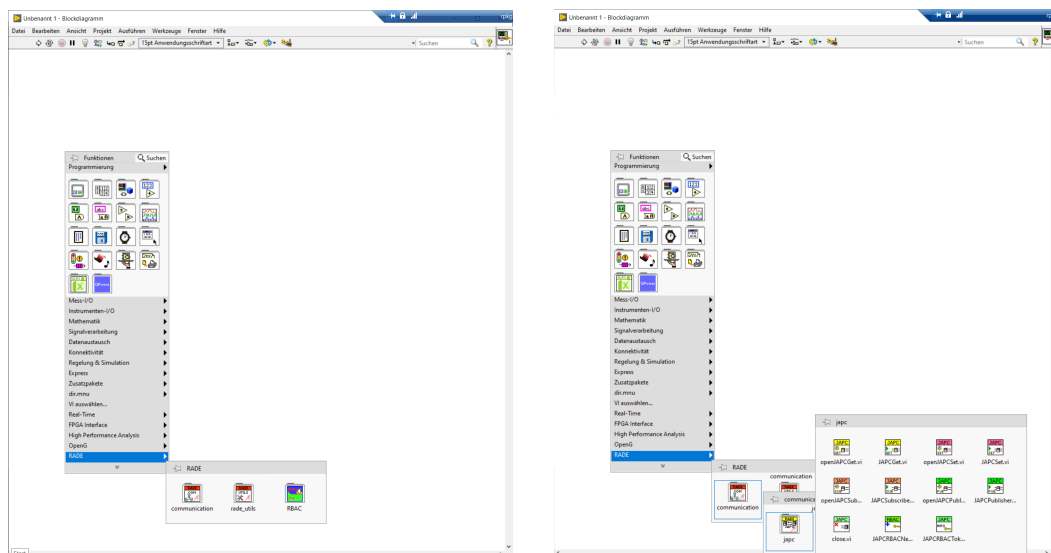


Figure 5.19: japc-1.0

and installed following the installation process, described in [Section 4.2](#) and tested in [Section 5.2](#). To prove everything works correctly, LabVIEW 2020 without the RADE framework is launched and the package 1j1-1.0 should be included in the palette with other LabVIEW functions. As expected, 1j1-1.0 is the only package inside RADE as shown in [Figure 5.17](#).

After having created the first package, another package is created to show how the RADE framework will be populated by installing RADE packages. At this time, the package to be created is JAPC, which is responsible for communication using Java API for Parameter Control. After having created the package, uploaded it to the repository, it can be installed. The GUI will show to the user the packages

to be installed, as reported in Figure 5.18. Once installed together with the dependencies, the RADE palette in LabVIEW will look as shown in Figure 5.19. In the palette, there is the folder `rade_utils`, containing `ljl-1.0` previously installed, and the other two folders: `communication` and `RBAC`. In `communication`, the package `japc-1.0` is contained, whereas in `RBAC` there is the package `cmw_rbac-1.0`. In fact, as Figure 5.18 and the graph in Figure 4.5 show, `japc` has some dependencies among RADE and OpenG libraries. Hence, those are installed as well. The reason why the packages `radebase` and `radelibs` do not show up in the palette is because they are common packages between RADE and they have functions which are not callable from the GUI. For the OpenG packages, it can be seen that those are installed as well in Figure 5.19.

In this way, by creating packages out of RADE, the decoupling requirement is achieved and a RADE user can install only the packages required instead of installing the whole framework, as it was previously the case.

All the packages from Figure 4.5 have been created using creation procedure and tested.

5.4 Cross-platform validation

CERN's users work on the three main operating systems: Windows, macOS and CERN CentOS 7 (CC7) [77], which is a Linux distribution. In addition to that, there are two different networks: GPN and TN. In the GPN, all the three main OS are used, whereas for the TN only computers with CC7 exposed to this network are working. Therefore, RPKG application has been tried on the three main OS and on a CC7 machine exposed to the TN.

The best test is the real world use, therefore the application has been:

1. developed and tested on a Windows10 computer on the GPN:
2. tested on a virtual machine, called *rpkg-linux*, with CC7 as operating system on the GPN, as shown in Figure 5.20;
3. tested on a virtual machine, called *cwe-513-vol383*, with CC7 as operating system on the GPN, as shown in Figure 5.21;
4. and tested on a Macintosh computer with macOS Monterey installed.

The OpenG library [74] is available in all the operating systems, therefore it has been used to check the installation, removing, and updating procedure on all the three OS. In Figure 5.20, Figure 5.21, and Figure 5.22 the application is running on Linux (under GPN and TN) and on mac OS.

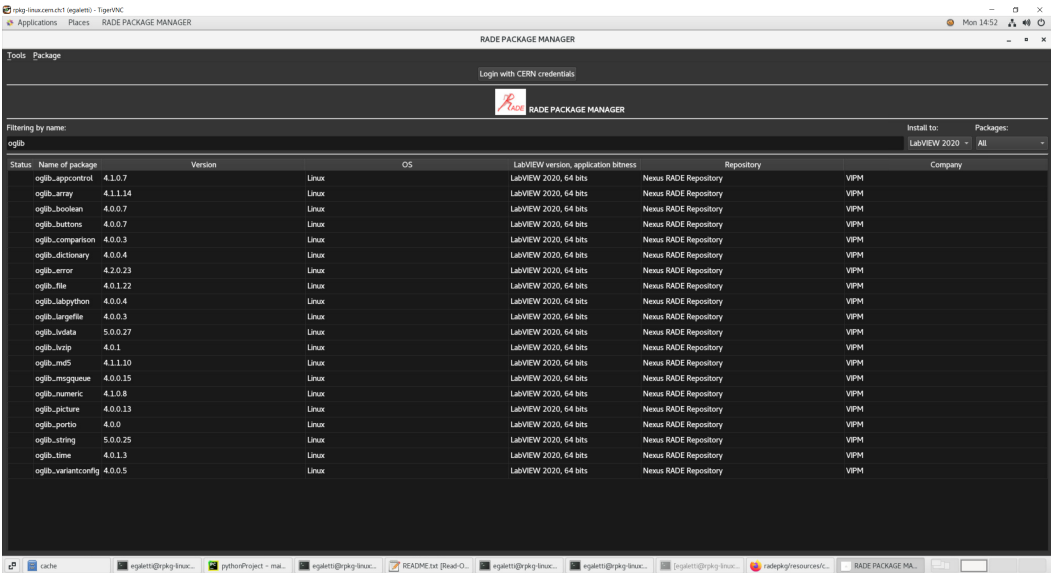


Figure 5.20: RPKG application running on Linux in the GPN

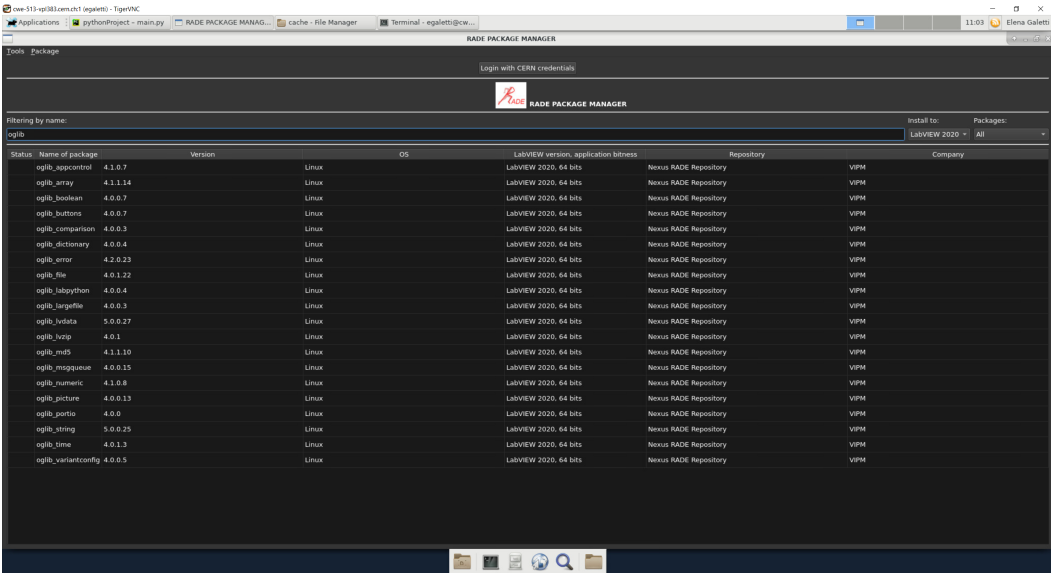


Figure 5.21: RPKG application running on Linux exposed to the TN

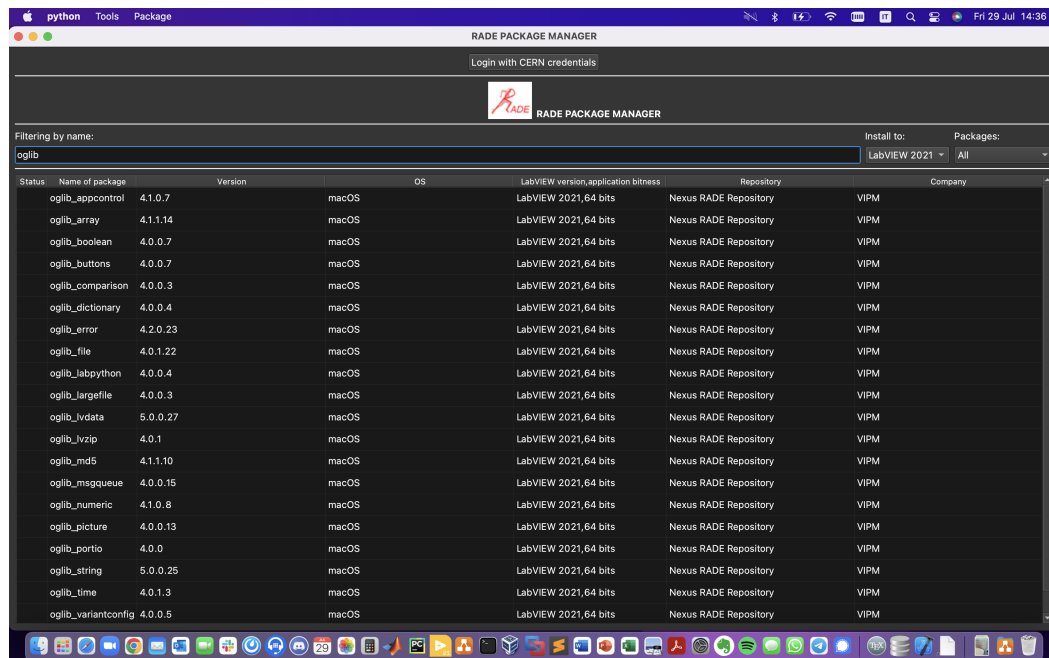


Figure 5.22: RPKG application running on macOS Monterey

Conclusions and future work

Contents

6.1	Conclusions	65
6.2	Future works	66

Chapter outline

In this final chapter, conclusions and future work are provided in the next two sections. [RPKG](#) application has been tested by the [MTA](#) team, and feedback has been taken into account and then incorporated for improvements. [RPKG](#) is still an ongoing project in the [MTA](#) section, where advancements and new features are constantly developed. [RPKG](#) will evolve as [RADE](#) will, and application might scale and distribute packages and projects written in multiple programming languages.

6.1 Conclusions

The aim of this work was to develop a new package manager, the [RADE](#) package manager ([RPKG](#)), tailored for [CERN](#)'s infrastructure and its users, helping the [MTA](#) team and [LabVIEW](#) users at [CERN](#) in distributing [LabVIEW](#) and [RADE](#) packages.

Therefore, [RPKG](#) can create, install, update and remove [LabVIEW](#) packages on a system. Thanks to the package creation process, [RPKG](#) is able to decouple the whole [RADE](#) framework into separate packages. This brings several benefits, such as the fact that end-users can install only the packages they want to use instead of the whole framework, or that [CI](#) will be triggered at the package level in case of changes, and not anymore at the framework level. This will make also the versioning process easier, since each package will have its own version.

Moreover, [RPKG](#) is tailored and developed according to the [CERN](#) control infrastructure and is cross-platform. These two features are essential since [CERN](#)'s users work on all the three main operating systems and the package manager must work on the two [CERN](#)'s networks, [GPN](#) and [TN](#) networks.

Before implementing [RPKG](#), an analysis of the *state-of-the-art* package managers has been carried out to understand if there was a tool available on the market that could solve [MTA](#)'s needs and requirements. However, all of them presented some

shortcomings, such as the fact of not being cross-platform or not having a public license.

RPKG is written in Python and it has a user-friendly **GUI**, written in PyQt5, such that end-users will not use any terminal to manage packages.

The evaluation in [Chapter 5](#) has verified that all the **RPKG** requirements, in [Chapter 4](#), are fully verified.

Currently **RPKG** is still under testing phase within the **MTA** team, and soon it will be deployed as an official tool.

6.2 Future works

RPKG application has not been evaluated and compared in terms of execution times to NIPM or VIPM, which could be considered as **RPKG**'s competitors. Therefore, a time execution study to eliminate the bottlenecks and understand which processes are taking more time is necessary to speed up the whole application. Some ideas on how to improve execution times, such as using **git** decorator, have been discussed at the beginning of [Chapter 5](#).

An important future work could be to exploit the class inheritance design, described in [Section 4.6](#), and scale the application from distributing and deploying **LabVIEW** code only, to other languages. Being a language agnostic package manager will facilitate the adoption of **RPKG** also in other teams at **CERN**. Therefore, it would be ideal to develop a new child class from the parent `Package()`, that can handle different packages than **LabVIEW** packages.

Regarding the package creation phase, an imminent future work is to link the creation phase with **CI**. Hence, every time a source project is modified, **CI** will be triggered at the latest tag, and a new package will be made.

As described in [Section 3.4](#), package managers suffer from security attacks and even the most popular used ones have their own security vulnerabilities. At the moment **RPKG** does not have any security mechanism implemented because it is exposed to **CERN**'s users only. To scale the application from **CERN** to the outside, a security mechanism is required to prevent malware to be uploaded to repositories where **RPKG** has access to for installing packages.

Example of a SPEC file

The following SPEC file is taken from oglib_file library.

```

1
2 [Package]
3 Name="oglib_file"
4 Version="4.0.1.22"
5 Release=""
6 ID=a37fb043e8ea135b8276dce9fb109eb8
7 File Format="vip"
8 Format Version="2010"
9 Display Name="OpenG File Library"
10
11
12 [Description]
13 Description="The OpenG File Library package contains several routines
14   for operating on files."
15 Summary="OpenG File Library"
16 License="BSD"
17 Copyright="2002–2010 Jim Kring, Rolf Kalbermatter; 2002–2003 Cal–Bay
18   Systems, Inc.; 2010–2011 Jonathon Green"
19 Distribution=""
20 Vendor="OpenG.org"
21 URL="http://wiki.openg.org/Oglib_file"
22 Packager="OpenG.org"
23 Demo="FALSE"
24 Release Notes="Package build with VIPM 2013\0A[FIX] 125 – File Library
25   Fix for QuickDrop and Shortcut Palettes: Add (OpenG) Suffix to
26   OpenG VIs that conflict with built-ins"
27 System Package="FALSE"
28 Sub Package="FALSE"
29 License Agreement="TRUE"
30
31
32 [Platform]
33 Exclusive_LabVIEW_Version=">=9.0"
34 Exclusive_LabVIEW_System="ALL"
35 Exclusive_OS="ALL"
36
37
38 [Script VIs]
39 PreInstall=""
40 PostInstall=""
41 PreUninstall=""
42 PostUninstall=""
43 Verify=""

```

```

40 PreBuild=""
41 PostBuild=""
42
43
44 [Dependencies]
45 AutoReqProv=FALSE
46 Requires="oglib_appcontrol >=2.10,oglib_array >=3.0.0,oglib_error >=2.3,
    oglib_string >=2.6"
47 Conflicts=""
48
49
50 [Activation]
51 License File=""
52 Licensed Library=""
53
54
55 [Files]
56 Num File Groups="3"
57 Sub-Packages=""
58 Namespaces="5F5F6F67746B"
59
60
61 [File Group 0]
62 Target Dir="<application>"
63 Replace Mode="Always"
64 Num Files=59
65 File 0="user.lib/_OpenG.lib/file/file.llb/Append Path to Root if
    Relative - Absolute or Relative Path Array__ogtk.vi"
66 File 1="user.lib/_OpenG.lib/file/file.llb/Append Path to Root if
    Relative - Array__ogtk.vi"
67 File 2="user.lib/_OpenG.lib/file/file.llb/Append Path to Root if
    Relative - Root Path Array__ogtk.vi"
68 File 3="user.lib/_OpenG.lib/file/file.llb/Append Path to Root if
    Relative - Scalar__ogtk.vi"
69 File 4="user.lib/_OpenG.lib/file/file.llb/Append Path to Root if
    Relative__ogtk.vi"
70 File 5="user.lib/_OpenG.lib/file/file.llb/Application Directory__ogtk.
    vi"
71 File 6="user.lib/_OpenG.lib/file/file.llb/Build Path - File Names and
    Paths Arrays - path__ogtk.vi"
72 File 7="user.lib/_OpenG.lib/file/file.llb/Build Path - File Names and
    Paths Arrays__ogtk.vi"
73 File 8="user.lib/_OpenG.lib/file/file.llb/Build Path - File Names Array
    - path__ogtk.vi"
74 File 9="user.lib/_OpenG.lib/file/file.llb/Build Path - File Names
    Array__ogtk.vi"
75 File 10="user.lib/_OpenG.lib/file/file.llb/Build Path - Traditional -
    path__ogtk.vi"
76 File 11="user.lib/_OpenG.lib/file/file.llb/Build Path -
    Traditional__ogtk.vi"
77 File 12="user.lib/_OpenG.lib/file/file.llb/Build Path__ogtk.vi"
78 File 13="user.lib/_OpenG.lib/file/file.llb/Compare File Binary__ogtk.vi
    "

```

```

79 File 14="user.lib/_OpenG.lib/file/file.llb/Compare Two Paths -
    Array__ogtk.vi"
80 File 15="user.lib/_OpenG.lib/file/file.llb/Compare Two Paths - Path1
    Array__ogtk.vi"
81 File 16="user.lib/_OpenG.lib/file/file.llb/Compare Two Paths - Path2
    Array__ogtk.vi"
82 File 17="user.lib/_OpenG.lib/file/file.llb/Compare Two Paths -
    Scalar__ogtk.vi"
83 File 18="user.lib/_OpenG.lib/file/file.llb/Compare Two Paths__ogtk.vi"
84 File 19="user.lib/_OpenG.lib/file/file.llb/Convert Dirs to VI Libraries
    (proxy)__ogtk.vi"
85 File 20="user.lib/_OpenG.lib/file/file.llb/Convert File Extension (Path
    )__ogtk.vi"
86 File 21="user.lib/_OpenG.lib/file/file.llb/Convert File Extension (
    String)__ogtk.vi"
87 File 22="user.lib/_OpenG.lib/file/file.llb/Convert File Extension__ogtk
    .vi"
88 File 23="user.lib/_OpenG.lib/file/file.llb/Convert VI Libraries to Dirs
    (proxy)__ogtk.vi"
89 File 24="user.lib/_OpenG.lib/file/file.llb/Copy with Options__ogtk.vi"
90 File 25="user.lib/_OpenG.lib/file/file.llb/Create Dir if Non-
    Existant__ogtk.vi"
91 File 26="user.lib/_OpenG.lib/file/file.llb/Current VI's Path__ogtk.vi"
92 File 27="user.lib/_OpenG.lib/file/file.llb/Current VIs Parent
    Directory__ogtk.vi"
93 File 28="user.lib/_OpenG.lib/file/file.llb/Default Directory__ogtk.vi"
94 File 29="user.lib/_OpenG.lib/file/file.llb/Delete Recursive__ogtk.vi"
95 File 30="user.lib/_OpenG.lib/file/file.llb/File Exists - Array__ogtk.vi
    "
96 File 31="user.lib/_OpenG.lib/file/file.llb/File Exists - Scalar__ogtk.
    vi"
97 File 32="user.lib/_OpenG.lib/file/file.llb/File Exists__ogtk.vi"
98 File 33="user.lib/_OpenG.lib/file/file.llb/File Info Record__ogtk.ctl"
99 File 34="user.lib/_OpenG.lib/file/file.llb/File Info__ogtk.vi"
100 File 35="user.lib/_OpenG.lib/file/file.llb/Force File Move__ogtk.vi"
101 File 36="user.lib/_OpenG.lib/file/file.llb/Instrument Library__ogtk.vi"
102 File 37="user.lib/_OpenG.lib/file/file.llb/List Directory
    Recursive__ogtk.vi"
103 File 38="user.lib/_OpenG.lib/file/file.llb/List Directory__ogtk.vi"
104 File 39="user.lib/_OpenG.lib/file/file.llb/List Top Level VIs__ogtk.vi"
105 File 40="user.lib/_OpenG.lib/file/file.llb/Merge Directories__ogtk.vi"
106 File 41="user.lib/_OpenG.lib/file/file.llb/OpenG Library__ogtk.vi"
107 File 42="user.lib/_OpenG.lib/file/file.llb/Set VI Top Level__ogtk.vi"
108 File 43="user.lib/_OpenG.lib/file/file.llb/Strip Path - Arrays__ogtk.vi
    "
109 File 44="user.lib/_OpenG.lib/file/file.llb/Strip Path -
    Traditional__ogtk.vi"
110 File 45="user.lib/_OpenG.lib/file/file.llb/Strip Path Extension - 1D
    Array of Paths__ogtk.vi"
111 File 46="user.lib/_OpenG.lib/file/file.llb/Strip Path Extension - 1D
    Array of Strings__ogtk.vi"
112 File 47="user.lib/_OpenG.lib/file/file.llb/Strip Path Extension -
    Path__ogtk.vi"

```

```

113 File 48="user.lib/_OpenG.lib/file/file.llb/Strip Path Extension -
      String__ogtk.vi"
114 File 49="user.lib/_OpenG.lib/file/file.llb/Strip Path Extension__ogtk.
      vi"
115 File 50="user.lib/_OpenG.lib/file/file.llb/Strip Path__ogtk.vi"
116 File 51="user.lib/_OpenG.lib/file/file.llb/Temporary Directory__ogtk.vi
      "
117 File 52="user.lib/_OpenG.lib/file/file.llb/Temporary Filename__ogtk.vi"
118 File 53="user.lib/_OpenG.lib/file/file.llb/User Library__ogtk.vi"
119 File 54="user.lib/_OpenG.lib/file/file.llb/Valid Path - Array__ogtk.vi"
120 File 55="user.lib/_OpenG.lib/file/file.llb/Valid Path -
      Traditional__ogtk.vi"
121 File 56="user.lib/_OpenG.lib/file/file.llb/Valid Path__ogtk.vi"
122 File 57="user.lib/_OpenG.lib/file/file.llb/VI Library__ogtk.vi"
123 File 58="user.lib/_OpenG.lib/file/file.llb/VI Tree - file__ogtk.vi"
124
125
126 [File Group 1]
127 Target Dir="<menus>/Categories/OpenG"
128 Replace Mode="Always"
129 Num Files=3
130 File 0="_functions_oglib_file_1.mnu"
131 File 1="_functions_oglib_file_2.mnu"
132 File 2="functions_oglib_file.mnu"
133
134
135 [File Group 2]
136 Target Dir="<menus>/Categories/OpenG"
137 Replace Mode="If Newer"
138 Num Files=1
139 File 0="dir.mnu"

```

Build file created for LabVIEW JSON Library

```
1 [Package]
2 Name = "ljl"
3 Version = "1.0"
4 File Format = "vip"
5 Format Version = ""
6 Display name = "LJL PROJECT"
7 Package type = "LabVIEW"
8
9 [Description]
10 Description = "This is the description of ljl project"
11 Summary = "Summary of ljl project"
12 License = "License ljl project"
13 Copyright = "Copyright of ljl"
14 Distribution = "Distribution of ljl"
15 Vendor =
16 Url = ""
17 Packager = ""
18 Demo = ""
19 Release Notes = ""
20 System Package = ""
21
22 [Platform]
23 Exclusive_LabVIEW_Version = ">=15.0"
24 Exclusive_LabVIEW_System = "ALL"
25 Exclusive_OS = "ALL"
26
27 [Dependencies]
28 AutoReqProv = FALSE
29 Requires = ""
30 Conflicts = ""
31
32 [Files]
33 Num File Groups = "3"
34 Sub-Packages = ""
35
36 [File Group 0]
37 Target Dir="user.lib\_RADE\ljl"
38 Replace Mode="Always"
39 Group type="Folder"
40 Source Dir="src\LabVIEW"
41
42 [File Group 1]
```

```
43 Target Dir="user.lib\_RADE\ljl"
44 Replace Mode="Always"
45 Group type="Folder"
46 Source Dir="palette"
47
48 [File Group 2]
49 Target Dir="user.lib"
50 Replace Mode="Always"
51 Group type="Dependency"
52 Top_level_VI="path to VI"
53
54 [rade_utils.mnu]
55 name = "rade_utils"
56 icon = "<userlib>/_RADE/ljl/palette/rademnu/img/Utils.png"
57 dest="<labview>/menus/Categories/RADE"
58 synch=TRUE
59 synchpath="<labview>/menus/Categories/RADE/_rade_utils"
60 vis = ""
61
62 [ljl.mnu]
63 name = "JSON Parser"
64 icon = "<userlib>/_RADE/ljl/palette/rademnu/img/ljl.png"
65 dest="<labview>/menus/Categories/RADE/_rade_utils"
66 synch=FALSE
67 vis = "ljl/vis/JSON value/Import JSON.vi,ljl/vis/JSON value/Export JSON
      .vi,ljl/vis/JSON value/Modify value (poly).vi,ljl/vis/JSON value/
      New value (poly).vi,ljl/vis/JSON value/Read value (poly).vi,ljl/vis
      /JSON value/is null.vi,ljl/vis/JSON value/Read value type.vi"
```

SPEC file generated by RPKG for LabVIEW JSON Library

The SPEC file generated by [RPKG](#) for LabVIEW JSON Library is shown below.

```
1 [Package]
2 name = "ljl"
3 version = "1.0"
4 file format = "vip"
5 format version = ""
6 display name = "LJL PROJECT"
7 package type = "LabVIEW"
8
9 [Description]
10 description = "This is the description of ljl project"
11 summary = "Summary of ljl project"
12 license = "License ljl project"
13 copyright = "Copyright of ljl"
14 distribution = "Distribution of ljl"
15 vendor =
16 url = ""
17 packager = ""
18 demo = ""
19 release notes = ""
20 system package = ""
21
22 [Platform]
23 exclusive_labview_version = ">=15.0"
24 exclusive_labview_system = "ALL"
25 exclusive_os = "ALL"
26
27 [Dependencies]
28 autoreqprov = FALSE
29 requires = ""
30 conflicts = ""
31
32
33 [Files]
34 Num File Groups = "3"
35 Sub-Packages = ""
36
37 [File Group 0]
38 Target Dir = "<application>"
39 Replace Mode = "Always"
40 Num Files = "75"
```

```

41 File 0 = "user.lib\_RADE\ljl\vis\Check value type.vi"
42 File 1 = "user.lib\_RADE\ljl\vis\Create.vi"
43 File 2 = "user.lib\_RADE\ljl\vis\JSON to map.vi"
44 File 3 = "user.lib\_RADE\ljl\vis\LV-JSON.lvproj"
45 File 4 = "user.lib\_RADE\ljl\vis\lvjson.mnu"
46 File 5 = "user.lib\_RADE\ljl\vis\Modify.vi"
47 File 6 = "user.lib\_RADE\ljl\vis\Read.vi"
48 File 7 = "user.lib\_RADE\ljl\vis\test FESA to VIRTUAL.vi"
49 File 8 = "user.lib\_RADE\ljl\vis\Test modification.vi"
50 File 9 = "user.lib\_RADE\ljl\vis\JSON array\Append element.vi"
51 File 10 = "user.lib\_RADE\ljl\vis\JSON array\Export JSON.vi"
52 File 11 = "user.lib\_RADE\ljl\vis\JSON array\JSON array.lvclass"
53 File 12 = "user.lib\_RADE\ljl\vis\JSON array\New array.vi"
54 File 13 = "user.lib\_RADE\ljl\vis\JSON array\Read array.vi"
55 File 14 = "user.lib\_RADE\ljl\vis\JSON array\Read value type.vi"
56 File 15 = "user.lib\_RADE\ljl\vis\JSON boolean\Export JSON.vi"
57 File 16 = "user.lib\_RADE\ljl\vis\JSON boolean\JSON boolean.lvclass"
58 File 17 = "user.lib\_RADE\ljl\vis\JSON boolean\New boolean.vi"
59 File 18 = "user.lib\_RADE\ljl\vis\JSON boolean\Read boolean.vi"
60 File 19 = "user.lib\_RADE\ljl\vis\JSON boolean\Read value type.vi"
61 File 20 = "user.lib\_RADE\ljl\vis\JSON null\Export JSON.vi"
62 File 21 = "user.lib\_RADE\ljl\vis\JSON null\is null.vi"
63 File 22 = "user.lib\_RADE\ljl\vis\JSON null\JSON null.lvclass"
64 File 23 = "user.lib\_RADE\ljl\vis\JSON null\New null.vi"
65 File 24 = "user.lib\_RADE\ljl\vis\JSON null\Read value type.vi"
66 File 25 = "user.lib\_RADE\ljl\vis\JSON number\Export JSON.vi"
67 File 26 = "user.lib\_RADE\ljl\vis\JSON number\JSON number.lvclass"
68 File 27 = "user.lib\_RADE\ljl\vis\JSON number\New number.vi"
69 File 28 = "user.lib\_RADE\ljl\vis\JSON number\Read float.vi"
70 File 29 = "user.lib\_RADE\ljl\vis\JSON number\Read integer.vi"
71 File 30 = "user.lib\_RADE\ljl\vis\JSON number\Read value type.vi"
72 File 31 = "user.lib\_RADE\ljl\vis\JSON object\Add member.vi"
73 File 32 = "user.lib\_RADE\ljl\vis\JSON object\Export JSON.vi"
74 File 33 = "user.lib\_RADE\ljl\vis\JSON object\JSON object.lvclass"
75 File 34 = "user.lib\_RADE\ljl\vis\JSON object\New object.vi"
76 File 35 = "user.lib\_RADE\ljl\vis\JSON object\Read object.vi"
77 File 36 = "user.lib\_RADE\ljl\vis\JSON object\Read value type.vi"
78 File 37 = "user.lib\_RADE\ljl\vis\JSON object\Remove member.vi"
79 File 38 = "user.lib\_RADE\ljl\vis\JSON string\Export JSON.vi"
80 File 39 = "user.lib\_RADE\ljl\vis\JSON string\JSON string.lvclass"
81 File 40 = "user.lib\_RADE\ljl\vis\JSON string\New string.vi"
82 File 41 = "user.lib\_RADE\ljl\vis\JSON string\Read string.vi"
83 File 42 = "user.lib\_RADE\ljl\vis\JSON string\Read value type.vi"
84 File 43 = "user.lib\_RADE\ljl\vis\JSON value\Depth to spaces.vi"
85 File 44 = "user.lib\_RADE\ljl\vis\JSON value\Export JSON.vi"
86 File 45 = "user.lib\_RADE\ljl\vis\JSON value\Export LabVIEW type.vi"
87 File 46 = "user.lib\_RADE\ljl\vis\JSON value\Extract value at path.vi"
88 File 47 = "user.lib\_RADE\ljl\vis\JSON value\Import JSON.vi"
89 File 48 = "user.lib\_RADE\ljl\vis\JSON value\is null.vi"
90 File 49 = "user.lib\_RADE\ljl\vis\JSON value\JSON value.lvclass"
91 File 50 = "user.lib\_RADE\ljl\vis\JSON value\Modify value (poly).vi"
92 File 51 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to array.vi"
93 File 52 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to boolean.vi"

```

```

94 File 53 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to null.vi"
95 File 54 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to number.vi"
96 File 55 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to object.vi"
97 File 56 = "user.lib\_RADE\ljl\vis\JSON value\Modify value to string.vi"
98 File 57 = "user.lib\_RADE\ljl\vis\JSON value\Modify value.vi"
99 File 58 = "user.lib\_RADE\ljl\vis\JSON value\New value (poly).vi"
100 File 59 = "user.lib\_RADE\ljl\vis\JSON value\Probe.vi"
101 File 60 = "user.lib\_RADE\ljl\vis\JSON value\Read value (poly).vi"
102 File 61 = "user.lib\_RADE\ljl\vis\JSON value\Read value type.vi"
103 File 62 = "user.lib\_RADE\ljl\vis\JSON value\Read value.vi"
104 File 63 = "user.lib\_RADE\ljl\vis\JSON value\value_types.ctl"
105 File 64 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Assert fixed string
.vi"
106 File 65 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse array.vi"
107 File 66 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse element.vi"
108 File 67 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse elements.vi"
109 File 68 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse member.vi"
110 File 69 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse members.vi"
111 File 70 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse number.vi"
112 File 71 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse object.vi"
113 File 72 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse string.vi"
114 File 73 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Parse value.vi"
115 File 74 = "user.lib\_RADE\ljl\vis\JSON value\Parser\Skip whitespace.vi"
116
117 [File Group 1]
118 Target Dir = "menus\Categories\RADE"
119 Replace Mode = "Always"
120 Num Files = "1"
121 File 0 = "rade_utils.mnu"
122
123 [File Group 2]
124 Target Dir = "menus\Categories\RADE\_rade_utils"
125 Replace Mode = "Always"
126 Num Files = "1"
127 File 0 = "ljl.mnu"

```


Calling LabVIEW from Python

```

1 def mass_compile(self):
2     """
3     This function is running the vi compile-all contained in the folder
4     created in the function copy_git_installer_scripts
5     :return:
6     """
7     git_installer_scripts_to_use = os.path.join(self.temp_path,
8         'git_installer_scripts_LabVIEW_' + self.lv_version + '_' + self.
9         lv_bitness + 'bits')
10    compile_all_vi = os.path.join(git_installer_scripts_to_use, 'mass-
11    compile', 'vis', 'compile-all.vi')
12    lv_exe_path = os.path.join(self.platform.calculate_labview_folder(
13        self.lv_bitness, self.lv_version),
14        'LabVIEW.exe')
15    command = '"' + str(lv_exe_path) + '" ' + str(compile_all_vi)
16    os.system(command)
17    print(command)
18
19 def generate_mnu_files(self):
20     """
21     This function is running the vi generate-single-mnu contained in
22     the folder created in the function copy_git_installer_scripts
23     :return:
24     """
25    git_installer_scripts_to_use = os.path.join(self.temp_path,
26        'git_installer_scripts_LabVIEW_' + self.lv_version + '_' + self.
27        lv_bitness + 'bits')
28    partial_pkg_path = self.config['File Group 0']['Target Dir'].
29        replace('<', '').replace('>', '').replace('"', '')
30    pkg_path = os.path.join(self.platform.calculate_labview_folder(self.
31        lv_bitness, self.lv_version),
32        partial_pkg_path)
33
34    lv_exe_path = os.path.join(self.platform.calculate_labview_folder(
35        self.lv_bitness, self.lv_version),
36        'LabVIEW.exe')
37    gen_mnu_vi = os.path.join(git_installer_scripts_to_use, 'mass-
38    compile', 'vis', 'generate-single-mnu.vi')
39    lv_path = self.platform.calculate_labview_folder(self.lv_bitness,
40        self.lv_version)
41
42    print(os.environ.get('pkg_path'))

```

```
32     os.environ['pkg_path'] = str(pkg_path)
33     os.environ['labview_path'] = str(lv_path)
34     os.environ['WORKSPACE'] = str(self.temp_path)
35
36
37     print(os.environ.get('pkg_path'))
38
39     command = '"' + str(lv_exe_path) + ' ' + '"' + str(gen_mnu_vi)
40     os.system(command)
41     print(command)
```

APPENDIX E

LabVIEW code

In the following pages, the mass compile VI and generation mnu VI are shown. Both VIs are involved in the creation of packages process.

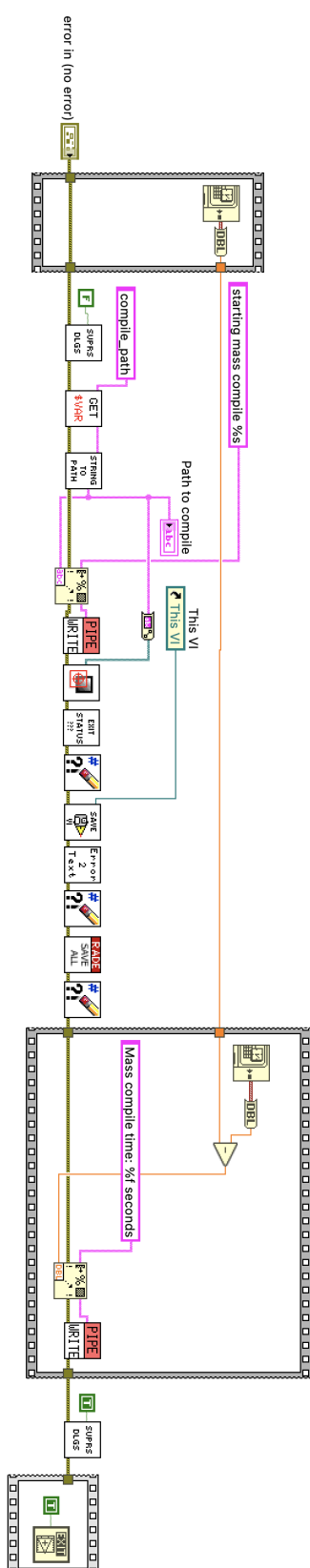


Figure E.1: Mass compile VI

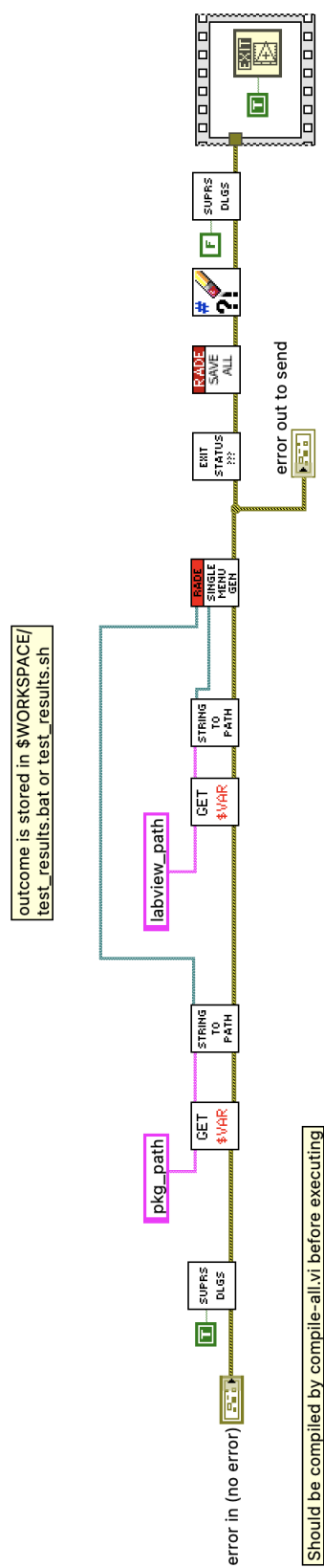


Figure E.2: Generation menu VI

Dependency solver algorithm

```

1 def get_package_child(self, package_name, package_version):
2     output_dependency_list = []
3     dependencies_list = []
4     c = self.connection.cursor()
5     c.execute(
6         "SELECT spec FROM cached_files WHERE package_name == ? COLLATE
          NOCASE AND package_version == ? COLLATE NOCASE",
7         (package_name, package_version))
8     data = c.fetchone()
9     if data is None:
10        package_version = self.get_different_version(package_name)
11    c.execute(
12        "SELECT spec FROM cached_files WHERE package_name == ? COLLATE
          NOCASE AND package_version == ? COLLATE NOCASE",
13        (package_name, package_version))
14    data = c.fetchone()
15    my_dict = ast.literal_eval(data[0])
16    try:
17        temp_dependency = my_dict["Dependencies"]["requires"]
18        dependencies_list_temp = [x.strip().lower() for x in
temp_dependency.split(',')]
19        for element in dependencies_list_temp:
20            dependencies_list.append(element.replace("'", ""))
21        for item in dependencies_list:
22            if ">=" in item:
23                temp_list = item.split(">=")
24                temp_list.append(">=")
25                output_dependency_list.append(tuple(temp_list))
26            elif "=" in item:
27                temp_list = item.split("=")
28                temp_list.append("=")
29                output_dependency_list.append(tuple(temp_list))
30    except KeyError:
31        self.rpkglib_logger.warning("Problem with package: %s ,
          version: %s" % (package_name, package_version))
32    return output_dependency_list
33
34 def get_package_successors(self, package_name, package_version,
done_list=[]):
35     to_do_list = []
36     to_do_list = self.get_package_child(package_name,
          package_version) # returns a
37     list of tuples containing children. A child is: (package_name,
package_version, package_sign)

```

```

38
39 while to_do_list:
40     first_tuple_of_list = to_do_list[0]
41     pkg_name = first_tuple_of_list[0]
42     pkg_version = first_tuple_of_list[1]
43     pkg_sign = first_tuple_of_list[2]
44     if pkg_name not in [x[0] for x in done_list]:
45         # append this tuple to done list
46         done_list.append(first_tuple_of_list)
47         # remove this tuple from to_do_list
48         to_do_list.remove(first_tuple_of_list)
49         # calculate the kids of this package
50         kids = self.get_package_child(pkg_name, pkg_version)
51         # then I must add the kids to the to_do_list
52         for kid in kids:
53             to_do_list.insert(0, kid)
54     else:
55         # if the sign is equal then we must append
56         if pkg_sign == '=':
57             done_list.append(first_tuple_of_list)
58             # remove this tuple from to_do_list
59             to_do_list.remove(first_tuple_of_list)
60             # calculate the kids of this package
61             kids = self.get_package_child(pkg_name, pkg_version)
62             # then I must add the kids to the to_do_list
63             for kid in kids:
64                 to_do_list.insert(0, kid)
65         else:
66             for x in done_list:
67                 if pkg_name == x[0] and version.parse(pkg_version)
> version.parse(x[1]):
68                     done_list.remove(x)
69                     done_list.append(first_tuple_of_list)
70                     to_do_list.remove(first_tuple_of_list)
71                     kids = self.get_package_child(pkg_name,
pkg_version)
72                     # then I must add the kids to the to_do_list
73                     for kid in kids:
74                         to_do_list.insert(0, kid)
75                     break
76             elif pkg_name == x[0] and version.parse(pkg_version)
) <= version.parse(x[1]):
77                 to_do_list.remove(first_tuple_of_list)
78                 break
79             else:
80                 pass
81 return done_list

```

RPKG's Graphical User Interface

In the next pages, the layout of the [GUI](#) is shown.



Figure G.1: GUI for RADE Package Manager

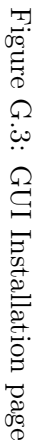


Figure G.3: GUI Installation page

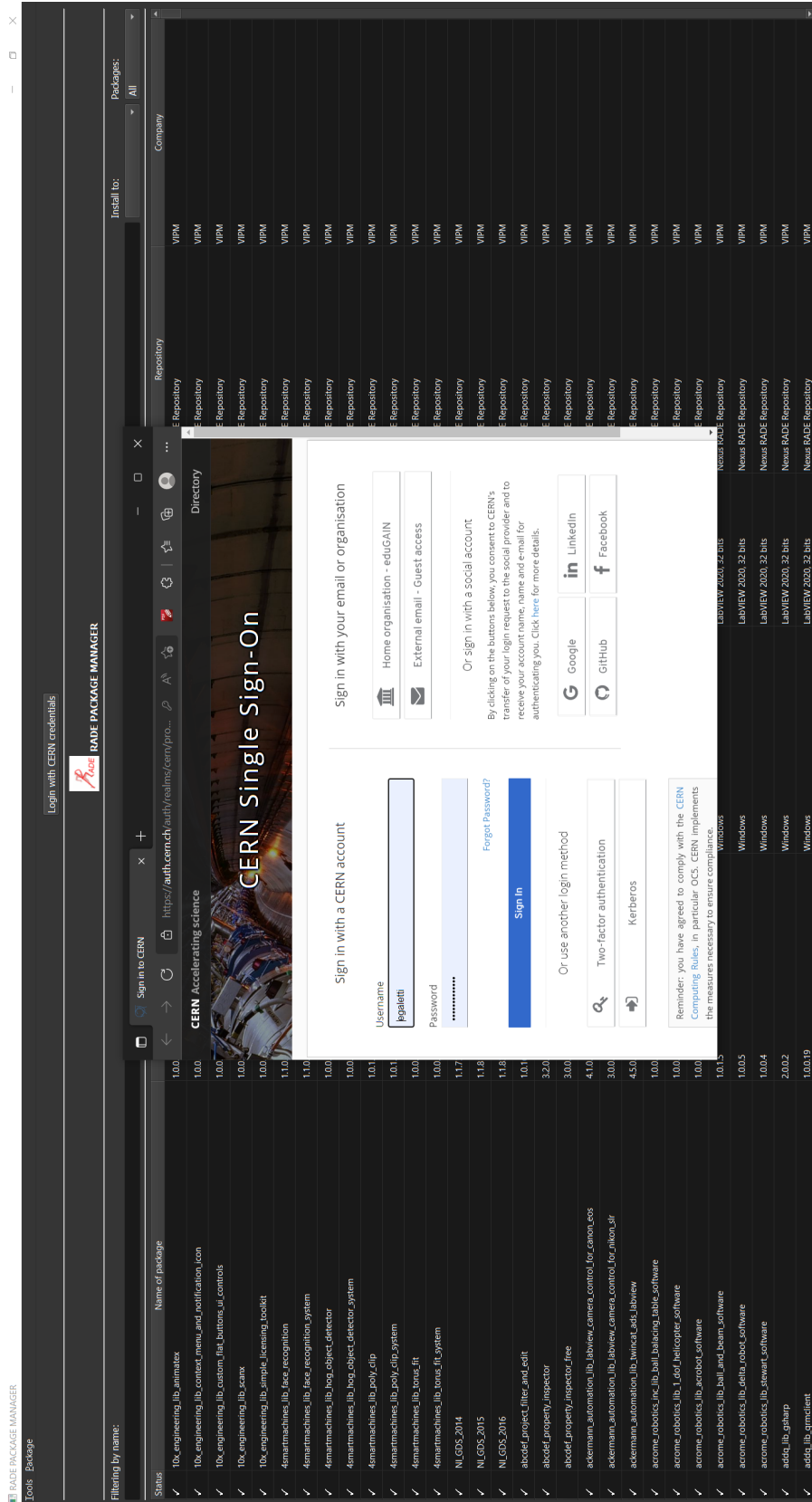


Figure G.4: GUI CERN SSO

Bibliography

- [1] “Complexe des accélérateurs du CERN - 2022.” <https://cds.cern.ch/images/CERN-GRAPHICS-2022-001-1>. Accessed: 2022-06-08. v, 8
- [2] “Our Impact.” <https://cds.cern.ch/record/2751223>. Accessed: 2022-05-07. v, 10
- [3] S. Deghaye and E. Fortescue-Beck, “Introduction to the BE-CO Control System,” tech. rep., 2020. v, 10, 11
- [4] “CERN Control Centre (CCC).” <https://home.cern/resources/image/accelerators/cern-control-centre-images-gallery>. Accessed: 2022-07-25. v, 11
- [5] “RADE Nexus Repository.” https://rade-vip-nexus.web.cern.ch/#browse/browse:eos_nexus. Accessed: 2022-07-28. v, 17, 18, 39
- [6] A. Miranda and J. Pimentel, “On the use of package managers by the C++ open-source community,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pp. 1483–1491, 2018. v, 20, 21, 24
- [7] J. Wang, Q. Wu, Y. Tan, J. Xu, and X. Sun, “A graph method of package dependency analysis on linux operating system,” in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, pp. 412–415, 2015. v, 22, 23
- [8] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, “A modular package manager architecture,” *Information and Software Technology*, vol. 55, no. 2, pp. 459–474, 2013. v, 23
- [9] O. Andreassen, C. Charrondière, M. Miskowiec, H. Reymond, A. Rijllart, *et al.*, “Synchronizing LabVIEW Development and Deployment environment,” 2019. vii, 3, 4, 5, 32
- [10] O. Andreassen, R. Sorokoletov, V. Shaipov, A. Rijllart, A. Raimondo, and D. Kudryavtsev, “The LabVIEW RADE framework distributed architecture,” in *Conf. Proc.*, vol. 111010, p. WEMAU003, 2011. 2, 9, 12
- [11] “LabVIEW.” <https://www.ni.com/en-gb/shop/labview.html#>. Accessed: 2022-03-15. 2
- [12] “Jenkins.” <https://www.jenkins.io>. Accessed: 2022-06-20. 3
- [13] O. Andreassen and A. Tarasenko, “Continuous Integration Using LabVIEW, SVN and Hudson,” 2014. 3

-
- [14] “National Instruments, VI Package Manager.” <https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000x1r4CAA&l=en-US>. Accessed: 2022-06-07. 4, 27, 53
- [15] “National Instruments, NI Package Manager.” <https://www.ni.com/en-us/support/downloads/ni-package-manager.html>. Accessed: 2022-06-07. 4, 27, 53
- [16] “The Debian package management tools.” <https://www.debian.org/doc/manuals/debian-faq/pkgtools.en.html>. Accessed: 2022-06-13. 4, 22
- [17] “Opkg lightweight package management system.” <https://git.yoctoproject.org/opkg/>. Accessed: 2022-06-13. 4, 22
- [18] M. Ewing and E. Troan, “The rpm packaging system,” in *Proceedings of the First Conference on Freely Redistributable Software, Cambridge, MA, USA*, 1996. 4, 22
- [19] “Brew, the Missing Package Manager for macOS (or Linux).” <https://brew.sh>. Accessed: 2022-06-14. 4
- [20] “Conda package manager.” <https://docs.conda.io/en/latest/>. Accessed: 2022-06-08. 4, 26
- [21] “Pip package installer.” <https://pip.pypa.io/en/stable/>. Accessed: 2022-06-08. 4, 27
- [22] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. De Supinski, and S. Futral, “The Spack package manager: bringing order to hpc software chaos,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015. 4, 5, 26
- [23] “FAQ. LHC The guide.” https://home.cern/sites/default/files/2018-07/CERN-Brochure-2017-002-Eng_0.pdf. Accessed: 2022-04-30. 8, 9
- [24] “CERN’s accelerator complex.” <https://home.cern/science/accelerators/accelerator-complex>. Accessed: 2022-06-08. 9
- [25] “Launch of the LHC Run 3.” <https://home.cern/events/launch-lhc-run-3>. Accessed: 2022-07-06. 9
- [26] “The Higgs boson: A landmark discovery.” <https://atlas.cern/higgs-boson-landmark-discovery>. Accessed: 2022-08-09. 9
- [27] “CERN Medipix.” <https://medipix.web.cern.ch/home>. Accessed: 2021-11-17. 9
- [28] “CERN MEDICIS.” <https://medicis.cern>. Accessed: 2022-05-07. 9

- [29] “First proposal for the World Wide Web.” <http://cdsweb.cern.ch/record/1405411/files/ARCH-WWW-4-010.pdf>. Accessed: 2022-06-30. 9
- [30] “LabVIEW Download - NI - National Instruments.” <https://www.ni.com/en-gb/support/downloads/software-products/download.labview.html#460613>. Accessed: 2022-07-25. 12
- [31] O. Andreassen, C. Charrondière, K. Develle, A. Rijllart, R. Rossel, J. Steen, J. Tagg, T. Zilliox, *et al.*, “Integrating COTS Equipment in the CERN Accelerator Domain,” 2019. 13
- [32] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, “A look in the mirror: Attacks on package managers,” in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 565–574, 2008. 13, 14, 17, 27, 29
- [33] “Package manager.” https://en.wikipedia.org/wiki/Package_manager. Accessed: 2022-07-07. 14
- [34] “Breadth-first search.” https://en.wikipedia.org/wiki/Breadth-first_search. Accessed: 2022-08-08. 15, 51
- [35] “Depth-first search.” https://en.wikipedia.org/wiki/Depth-first_search. Accessed: 2022-08-08. 15, 46
- [36] “Wikipedia contributors for the page Dependency hell.” https://en.wikipedia.org/wiki/Dependency_hell. Accessed: 2022-06-07. 15
- [37] “Nexus Repository.” <https://www.sonatype.com/products/nexus-repository>. Accessed: 2022-06-19. 17, 32
- [38] “List of software package management systems.” https://en.wikipedia.org/wiki/List_of_software_package_management_systems. Accessed: 2022-08-02. 19
- [39] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, “Dependency solving: a separate concern in component evolution management,” *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012. 21
- [40] D. C. Kozen, “Depth-first and breadth-first search,” in *The design and analysis of algorithms*, pp. 19–24, Springer, 1992. 21
- [41] A. Huebl, “Working with Multiple Package Managers,” 2020. 22
- [42] “YaST.” <https://en.opensuse.org/Portal:YaST>. Accessed: 2022-06-20. 22, 29
- [43] “YUM.” <https://www.redhat.com/sysadmin/how-manage-packages>. Accessed: 2022-06-20. 22, 29

-
- [44] “urpmi.” https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch07s03s05.html. Accessed: 2022-06-20. 22, 29
- [45] P. Trezentos, I. Lynce, and A. L. Oliveira, “Apt-pbo: solving the software dependency problem using pseudo-boolean optimization,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 427–436, 2010. 22
- [46] “aptitude.” <https://www.debian.org/doc/manuals/aptitude/rn01re01.en.html>. Accessed: 2022-06-28. 22, 29
- [47] “cupt - console package management tool.” <https://manpages.ubuntu.com/manpages/bionic/man1/cupt.1.html>. Accessed: 2022-07-28. 22
- [48] “smart package manager.” <https://github.com/smarterpm/smart>. Accessed: 2022-07-28. 22
- [49] M. F. Krafft, K.-J. Stol, and B. Fitzgerald, “How do free/open source developers pick their tools? A Delphi study of the Debian project,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 232–241, 2016. 22
- [50] “Portage, Gentoo package manager.” <https://packages.gentoo.org/packages/sys-apps/portage>. Accessed: 2022-06-19. 24, 29
- [51] R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez-Matamoros, “Gentoo package dependencies over time,” in *Proceedings of the 11th Working conference on mining software repositories*, pp. 404–407, 2014. 24
- [52] R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez-Matamoros, “Innovation diffusion in open source software: Preliminary analysis of dependency changes in the gentoo portage package database,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 316–319, 2014. 24
- [53] “Paludis, the official package manager of Exherbo linux.” <https://paludis.exherbo.org>. Accessed: 2022-06-19. 24
- [54] O. Bal-Pétre, P. Varlez, and F. Perez-Tellez, “Pacloud: Towards a Universal Cloud-based Linux Package Manager,” in *Proceedings of the 2019 International Communication Engineering and Cloud Computing Conference*, pp. 6–13, 2019. 24
- [55] L. Courtès, “Functional package management with guix,” *arXiv preprint arXiv:1305.4584*, 2013. 25
- [56] E. Dolstra, M. De Jonge, E. Visser, *et al.*, “Nix: A Safe and Policy-Free System for Software Deployment,” in *LISA*, vol. 4, pp. 79–92, 2004. 25

- [57] E. Dolstra and A. Löb, “NixOS: A purely functional Linux distribution,” in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pp. 367–378, 2008. 25
- [58] S. Jackman, I. Birol, S. Jackman, and I. Birol, “Linuxbrew and Homebrew for cross-platform package management,” *F1000Res*, vol. 5, p. 1795, 2016. 25
- [59] J. Hviid, A. Johansen, F. Caleb Sangogboye, and M. B. Kjærgaard, “Opm: An ontology-based package manager for building operating systems,” in *11th International Conference on the Internet of Things*, pp. 118–125, 2021. 26
- [60] “Docker.” <https://www.docker.com>. Accessed: 2022-06-18. 26
- [61] “Apache Maven.” <https://maven.apache.org>. Accessed: 2022-06-07. 26
- [62] “The comprehensive perl archive network (cpan).” <https://www.cpan.org>. Accessed: 2022-06-20. 26
- [63] “Conda (package manager).” [https://en.wikipedia.org/wiki/Conda_\(package_manager\)](https://en.wikipedia.org/wiki/Conda_(package_manager)). Accessed: 2022-08-09. 26
- [64] “JKI.” <https://www.jki.net>. Accessed: 2022-06-24. 27
- [65] “National Instruments, VI Package Manager enterprise license.” <https://www.vipm.io/download/>. Accessed: 2022-06-14. 27
- [66] A. Athalye, R. Hristov, T. Nguyen, and Q. Nguyen, “Package Manager Security,” tech. rep., Technical Report. [https://pdfs.semanticscholar.org/d398 ...](https://pdfs.semanticscholar.org/d398...), 2014. 27, 29
- [67] “apt-rpm.” https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch07s03s06.html. Accessed: 2022-06-20. 29
- [68] “pacman for Arch Linux.” <https://wiki.archlinux.org/title/pacman>. Accessed: 2022-06-20. 29
- [69] “Slackware package manager.” https://docs.slackware.com/slackware:package_management. Accessed: 2022-06-20. 29
- [70] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, “Detecting suspicious package updates,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 13–16, IEEE, 2019. 29
- [71] “Sqlite.” <https://www.sqlite.org/index.html>. Accessed: 2022-06-20. 32
- [72] “CERN Gitlab.” <https://gitlab.cern.ch>. Accessed: 2022-06-20. 34, 37

-
- [73] “LabVIEW JSON Library.” <https://gitlab.cern.ch/rade/rade-core/libs/programming/ljl>. Accessed: 2022-06-24. 36
 - [74] “OpenG by OpenG - Toolkit for LabVIEW Download.” https://www.vipm.io/package/openg.org_lib_openg_toolkit/. Accessed: 2022-07-28. 39, 62
 - [75] “Red Hat OpenShift.” <https://www.redhat.com/en/technologies/cloud-computing/openshift>. Accessed: 2022-08-09. 39
 - [76] “Compiling Python code with @jit.” <https://numba.pydata.org/numba-doc/latest/user/jit.html>. Accessed: 2022-07-28. 53
 - [77] “CERN CentOS 7.” <https://linux.web.cern.ch/centos7/>. Accessed: 2022-07-28. 62