



F3

**Faculty of Electrical Engineering
Department of computer science**

Master's Thesis

Automation of testing of operating system backup and recovery

Bc. Anton Voznia
Open informatics

January 2023

Supervisor: RNDr. Pavel Cahyna, Ph.D.

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Voznia** Jméno: **Anton** Osobní číslo: **474440**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávající katedra/ústav: **Katedra počítačů**
Studijní program: **Otevřená informatika**
Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Automatické testování zálohy a obnovy operačního systému

Název diplomové práce anglicky:

Automation of testing of operating system backup and recovery

Pokyny pro vypracování:

The goal of the thesis is to propose solution for automation of testing of offered changes of ReaR (Relax-and-Recover) disaster recovery tool source repository (continuous integration testing).

Guidelines:

- 1) Describe requirements to infrastructures for ReaR Continuous integration testing. Examine existing infrastructures (for example Travis CI and CentOS CI) for suitability.
- 2) Execute at least one test in infrastructure via TMT (Test Management Tool).
- 3) Propose a solution for static code analysis of the ReaR program code, or alternatively, of unit testing of internal functions in the program.
- 4) Describe disadvantages of ReaR for CI testing and extend ReaR to overcome them.
- 5) Document the result.

Seznam doporučené literatury:

- 1) W. Preston: Backup & Recovery. O'Reilly, 2009.
- 2) Jez Humble, Devid Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)) 1st Edition
- 3) Vladimir Khorikov: Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C# 1st Edition

Jméno a pracoviště vedoucí(ho) diplomové práce:

Mgr. Pavel Cahyna, Ph.D. ÚFP AVČR, Za Slovankou 3, 182 00 Praha 8

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.02.2022**

Termín odevzdání diplomové práce: **10.01.2023**

Platnost zadání diplomové práce: **30.09.2023**

Mgr. Pavel Cahyna, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

Acknowledgement / Declaration

I would like to thank my thesis adviser RNDr. Pavel Cahyna, Ph.D., for support, good ideas, and a lot of knowledge I got from him. I would like to thank my family, especially thank my mother, for supporting and advice. Also, thanks to my colleagues for giving me time for the master's thesis, and my friends.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, January 9, 2023

.....

Abstrakt / Abstract

Tato diplomová práce popisuje různé přístupy k testování otevřeného soutvarého projektu ReaR. Tady se ukazuje, jak je možné automatizovat testování bez manuálního spouštění testů. V této práci se používá moderní metody pro spouštění testů v infrastruktuře nebo oblačné systémy s použitím CI, které jsou populární v dnešní době. Jsou popsány dvě hlavní oblasti: testování základní funkcionality a statická analýza kódu. V dokumentu jsou uvedeny pokyny pro spouštění testů v infrastruktuře, a rozšíření existujících testů pro spouštění v cloudu a infrastruktuře. Kvůli omezení cloudových systémů pro konfigurace, byly popsány několik metod pro ladění.

Klíčová slova: infrastruktura; ReaR; relax-and-recover; testování; CI; záloha; obnova; statická analýza; operační systém.

Překlad titulu: Automatické testování zálohy a obnovy operačního systému

This diploma thesis describes different approaches for testing the open-source project ReaR. It demonstrates how it is possible to automate testing without manual test execution. It uses modern techniques to execute the testing in an infrastructure or cloud systems by using continuous integration, which is a trendy method today. Two main testing fields are described: testing the basic functionality of ReaR and static code analysis. In this document was provide instruction on how it is possible to execute the tests in infrastructure and an extension for the tests to run in available cloud and infrastructures. Because of restrictions of cloud systems for modification, this document described a few methods for debugging.

Keywords: infrastructure; ReaR; relax-and-recover; testing; CI; continuous integration; cloud; backup; restore; recovery; disaster recovery; static analysis; operating system.

/ Contents

1 Motivation	1
2 ReaR	3
2.1 Description	3
2.2 Example with ReaR	3
2.3 ReaR testing requirements	9
3 Infrastructure for ReaR	10
3.1 Continuous integration	10
3.2 Building	10
3.3 Infrastructure for backup and recovery testing	11
4 ReaR testing in infrastructure	12
4.1 Test Management Tool (TMT)	12
4.2 ReaR, recovery tests	16
4.3 Execute TMT test in an infrastructure	20
4.4 Testing recovery over NFS	21
4.5 Run test in infrastructure	24
4.6 Type of backup	27
4.7 ISO backup. Modification recovery test for infrastructure	30
4.8 Debugging	36
5 File name validation	39
6 Static code analysis	41
6.1 ShellCheck	41
6.2 Criteria for static analyzer	43
6.3 Differential ShellCheck	44
6.4 Script for automatic PRs	47
6.5 Script modification	49
7 Conclusion	50
References	51

/ **Figures**

2.1	Initial parameters of virtual machine	4
2.2	Make bootable second disk	5
2.3	The second disk for backup	5
2.4	lsblk	5
2.5	lsblk	6
2.6	Boot menu of ReaR	8
2.7	Automatic recover	8
2.8	Rear ask for reboot in automatic mode	8
2.9	Files after recover	8
4.1	TMT discover tests	13
4.2	TMT executes simple tests	14
4.3	TMT executes simple tests on our VM	15
4.4	No bootable device	19
4.5	VM cloning	22
4.6	TMT run multi-host tests	24
4.7	Simple test in Testing Farm ...	27
4.8	Simple test passed	28
4.9	GRUB, list disks and partition scheme	32
4.10	Passed ISO backup test	36
4.11	Failed test, instructions	38
5.1	make validate failed	40
5.2	make validate passed	40
6.1	ShellCheck example	42
6.2	Differential ShellCheck Checks output	45
6.3	Differential ShellCheck files changed	45
6.4	Add new lines into the file	46

Chapter 1

Motivation

Making a backup is an essential and usual process for now. Operating systems became more complicated. That causes to problems during development or usage it. A bit changes may crash the whole system, and cause loss of data. To prevent it, a lot of methods were invented for backup data. Most of them is just an naive data storage that copies all files on the backup storage. The main problem with such methods is the lack of disaster recovery. The term “disaster recovery” means an opportunity to restore the whole system without reinstalling an operating system. For example, in case the partition table, the boot partition, or some essential files for booting OS (like *initrd*) had been lost or broken, before recovery data, we would install a clean image of an OS, and we could restore data. In most cases, it could be more practical and may take a lot of time to configure the new system to the previous set-up.

ReaR is an open-source project that has existed since 2006. ReaR means “relax and recover”. Unfortunately, ReaR does not provide a simple solution for backup and recovery. It is more complicated to configure and make a backup. Regardless of the existing complications, ReaR has a lot of advantages. The most important is “disaster recovery”. The word “relax” in the program’s name means that we can make a backup and not worry about data loss because ReaR allows us to restore the system at the moment the backup is made.

ReaR is a very popular and famous project. Every week developers create pull requests on GitHub with solved issues and suggested enhancements. Such an approach allows fast-evolving projects, adding new features and integrating new ideas. That is a very strong side of all open-source projects, and simultaneously it has a big disadvantage. Every integration goes through a few steps in reviewing and discussing on GitHub. But it cannot ensure a lack of regressions. For that reason, were developed different approaches for testing before integration. One of them that is very popular today is “continuous integration”. It is a set of steps that the product should go through successfully before changes must be integrated. Though ReaR has existed since 2006, it does not have strict test rules and tests itself. Unfortunately, in some cases, it caused a regression.

The main idea of the work is to suggest automatic testing for backup and restoring an operating system via ReaR. The developers of ReaR should not have manually executed many times the same test. It would be well to have a solution that could verify if a change suggested by the maintainer of the project does not break the package build and does not introduce an obvious regression. Besides regression detection, it should use static analysis for the code of the project autonomously.

So, in chapter 2, I described what ReaR is and demonstrated an example of how to make a backup and then restore it. Then in chapter 3, I described the requirements for automatic testing. I explained why I chose a specific infrastructure. Chapter 4 explains tests for ReaR that Lukáš Zaoral had already written. In chapter 5, I showed what framework TMT is and how it may be used for our aims, and in detail, I described how I extended the tests from Lukáš Zaoral. Chapter 6 contains a description of an

Chapter 2

ReaR

2.1 Description

Relax-and-Recover (ReaR) is an open-source project that provides a simple way to make backup and disaster recovery. ReaR has a lot of benefits against usual solutions (copying via *dd*, *rsync* ...) for backup. ReaR can store data on different types of boot media (ISO, USB, eSATA, ...), and supports network protocols (iSCSI, NFS, ftp, sftp ...). By default ReaR, besides making of backup, copies the kernel (including all drivers) of a Linux distribution you are using in a place we can boot from (it may be remote disk, local disk, ISO image, etc.). Such an approach allows a backup and minimal bootable system with an interactive interface to restore data in case of a total loss of your operating system. The minimal bootable system is defined as *rescue system*.

Because ReaR copies kernel and drivers for interactive restoring, it is possible to configure and control the whole process of recovery. It supports such protocols as SSH (Secure Shell); you can connect to the machine (in case it is not disabled in ReaR configuration) and use it as a typical setup with bit limits.

The most important benefit of ReaR usage is the remaining disk layout. In case a machine has different disks with varied file systems for every disk, it is possible to restore the whole system to its original state.

For example, it is possible to make a backup by the command *dd*. The advantage of such an approach is simplicity. It is a program that copies block by block a file into another file. It allows copying a whole disk into another. Or copy a disk into a file. The disadvantage of this usage is that *dd* also copies unused blocks on the disk space. Because of the disk's defragmentation, the command processes unused disk space. Another complication of the usage, the program makes just a copy of a disk. It is not capable of creating a disk layout and restoring it. The user should restore the data disk by disk, in case of a few disks using its very simple job. It is impossible to restore if it uses different file systems across a set of disks.

Additionally, ReaR supports just making a backup of data without the need of copying the rescue system.

I showed an example with a backup on the NFS server in section 4.4. I used the term "multi-host". Because we need two machines: the NFS client (a system we want to backup) and the NFS server (a system where we want to store the backup), I called such an approach as "multi-host".

2.2 Example with ReaR

In this section, I will demonstrate an example of how to use ReaR. We will create a Virtual Machine with Fedora Linux 37, add an additional disk for ReaR backup, make the backup, and look at the recovery part.

For this example I used QEMU. "QEMU is a generic and open source machine emulator and virtualizer. QEMU can be used in several different ways. The most

common is for “system emulation”, where it provides a virtual model of an entire machine (CPU, memory and emulated devices) to run a guest OS. In this mode the CPU may be fully emulated, or it may work with a hypervisor such as KVM, Xen, Hax or Hypervisor.Framework to allow the guest to run directly on the host CPU. The second supported way to use QEMU is “user mode emulation”, where QEMU can launch processes compiled for one CPU on another CPU. In this mode the CPU is always emulated.”¹

To simplify QEMU usage, I installed Virtual Machine Manager.² It is a graphical interface for QEMU (also for KVM, but we do not use KVM).

Fedora 37 was chosen because the author is familiar with the OS and is an active user of the Linux distribution. I recommend downloading Fedora 37 Server because it takes a small amount of disk space and doesn’t contain extra packages (such as audio player, graphical interfaces e.t.c.) that are not needed for our purposes.

Also, I want to mention that all commands I executed as root user, which means a reader could see “#” at the beginning of the command line.

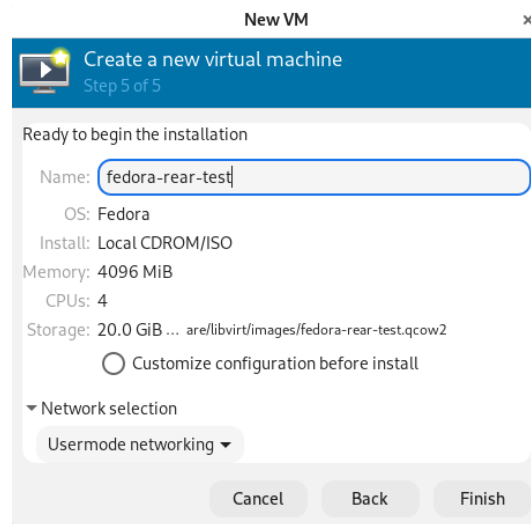


Figure 2.1. Initial parameters of virtual machine

In the Figure 2.1 is shown a VM with 4GiB RAM, 4 CPUs, and 20.0 GiB space storage. After installation, we should add a second disk which will be used for a backup of the system. From my own experience with the ReaR, it would be enough a 5 GiB for a disk (Figure 2.3).

Figure 2.2 shows that it is needed to make the second disk bootable. To simplify tests, we can mark that we want to use the boot menu, where we will choose the disk we want to boot from.

After booting, we need to install ReaR. Be care that all commands I used there are for Fedora and CentOS distributions. They may not be valid for other Linux distributions. Before making a backup, we should verify if we have a partition table for the second disk.

As we can see in Figure 2.4 there is no partition for our disk `/dev/vdb` From that, we need to create a partition table in this step.

¹ <https://www.qemu.org/docs/master/about/index.html>

² <https://virt-manager.org/>

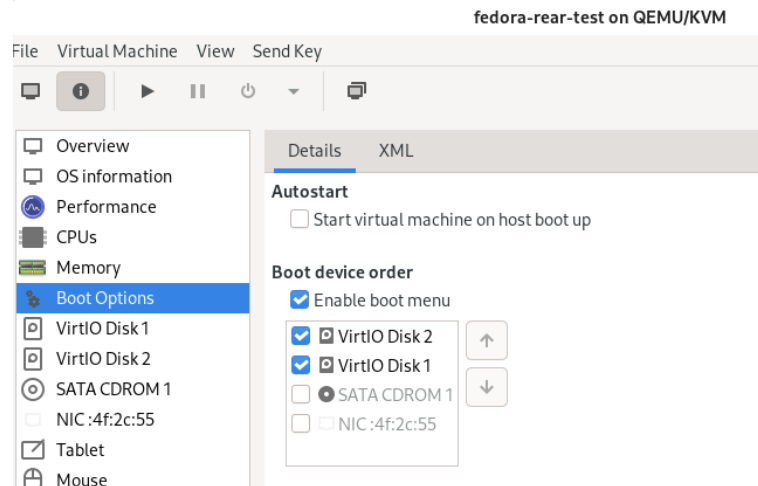


Figure 2.2. Make bootable second disk

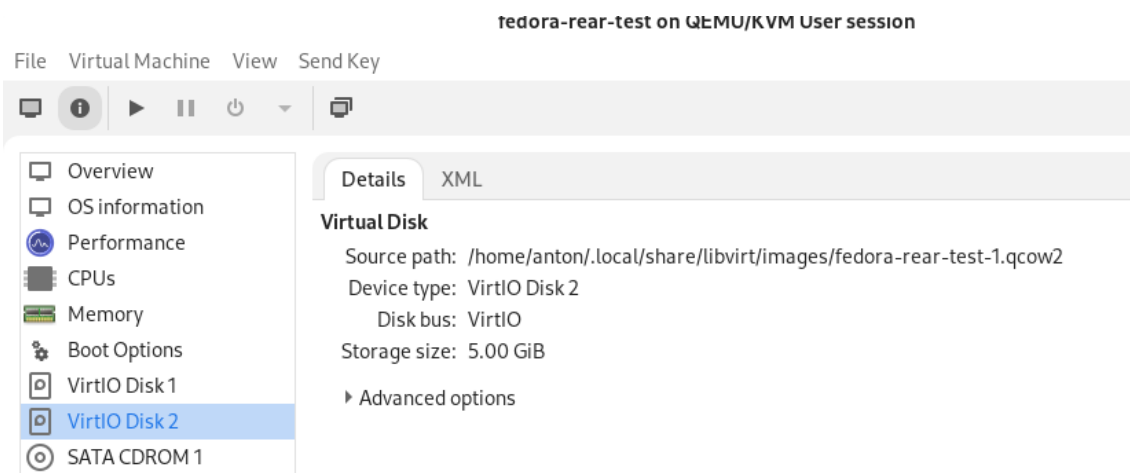


Figure 2.3. The second disk for backup

```
[root@localhost ~]# lsblk -f
NAME                FSTYPE    FSVER    LABEL UUID                                FSAVAIL FSUSE% MOUNTPOINTS
sr0
zram0
vda
├─vda1
├─vda2              xfs
├─vda3              LVM2_member LVM2 001 08KE3P-000i-nug1-2q3y-3PV0-Bc09-nwZ1r2 745.7M 22% /boot
└─fedora-root xfs      2f11ee06-7262-469a-8522-89cf183b9181 13.3G 11% /
vdb
```

Figure 2.4. All disks and their partitions

```
[root@localhost ~]# fdisk /dev/vdb
```

```
Welcome to fdisk (util-linux 2.38.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.
```

```
Command (m for help): n
```

```
Partition type
```

```
  p   primary (0 primary, 0 extended, 4 free)
```

```

e extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1):
First sector (2048-10485759, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P}
(2048-10485759, default 10485759):

Created a new partition 1 of type 'Linux' and of size 5 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.

```

Verify via `lsblk -f` that the partition has been created.

```

[root@localhost ~]# lsblk -f
NAME            FSTYPE     FSVER    LABEL UUID                                FSAVAIL FSUSE% MOUNTPOINTS
sr0
zram0
vda
├─vda1
├─vda2          xfs        1ca8556e-fd6a-48ac-bff8-260717c8463d  744.6M  22% /boot
├─vda3          LVM2_member LVM2 001  08KE3P-000i-nug1-2q3y-3PV0-Bc09-nwZ1r2
└─fedora-root  xfs        2f11ee06-7262-469a-8522-89cf183b9181  13.3G  11% /
vdb
└─vdb1

```

Figure 2.5. vdb contains partition table

Now we can install ReaR and make a backup:

```

## install ReaR
[root@localhost ~]# sudo dnf install rear -y

## configuration for ReaR backup
[root@localhost ~]# cat /etc/rear/local.conf

# Create a bootable USB disk (using extlinux).
# Specify the USB storage device by using USB_DEVICE.
OUTPUT=USB

BACKUP=NETFS

# To backup to USB storage device, use
# BACKUP_URL=usb:///dev/disk/by-label/REAR-000
# or use a real device node or a specific filesystem
# label. Alternatively, you can specify the device
# using USB_DEVICE=/dev/disk/by-label/REAR-000.
BACKUP_URL="usb:///dev/disk/by-label/REAR-000"

```

The description of the configuration you can read in detail on the webpage ³

³ <https://github.com/rear/rear/blob/master/doc/user-guide/03-configuration.adoc>

After ReaR installation, a backup is needed to create a filesystem suited for ReaR backup. It is possible to do this via the following command `# rear format /dev/vdb`. Type “Yes” and rear creates ext3 filesystem on the `/dev/vdb1` device. It is essential to note that in my case, the second disk is named “vdb”. In other cases, it may be different.

The first time I tried to make a backup, I got the following error in a log file `/var/log/rear/rear-localhost.log`

```
2022-12-03 23:32:37.488532799 ERROR: Executable extlinux is missing!
Please install syslinux-extlinux or alike
```

That means ReaR from the standard repository doesn’t have dependencies for some packages. So we can install it and make a backup. Additionally, we make a simple and naive test to see that ReaR works correctly. Let’s create ten files:

- file0...file4 will be created before backup with content of date creation
- file5...file9 will be created after backup with the same content of date creation.

file0...file4 would be exist after recovery, and file5...file9 would be deleted. In such a way, we can verify that recovery is working correctly.

```
[root@localhost ~]# for i in {0..4}; do echo $(date) > "file$i"; done

[root@localhost ~]# dnf install syslinux-extlinux -y
...
[root@localhost ~]# rear mkbackup -v
...
[root@localhost ~]# for i in {5..9}; do echo $(date) > "file$i"; done
[root@localhost ~]# cat file*
Sun Dec 4 05:08:08 PM CET 2022
Sun Dec 4 05:08:08 PM CET 2022
Sun Dec 4 05:08:08 PM CET 2022
Sun Dec 4 05:08:08 PM CET 2022
Sun Dec 4 05:08:08 PM CET 2022
Sun Dec 4 05:11:41 PM CET 2022
Sun Dec 4 05:11:41 PM CET 2022
Sun Dec 4 05:11:41 PM CET 2022
Sun Dec 4 05:11:41 PM CET 2022
Sun Dec 4 05:11:41 PM CET 2022
```

Now reboot the system. After boot, you can see a similar menu in Figure 2.6. Choose in the Recovery images section `localhost`.

Let’s use the automatic mode; ReaR will perform the following steps (Figure 2.7).

At the end of the recover, I got the following issue (Figure 2.8). In previous steps, we selected automatic mode, which means ReaR would be done by itself. But for unknown reasons, it asks to interact with the interface. But regardless the chosen mode it asks to interact with the interface. It is caused that we did not set the following variable in the configuration file `ISO_RECOVER_MODE=unattended`. From the default configuration file `/usr/share/rear/conf/default.conf` “`ISO_RECOVER_MODE=unattended` boots the ReaR recovery system with the ‘unattended’ kernel command line option that runs `rear recover` automatically plus automated reboot after successful rear recover”

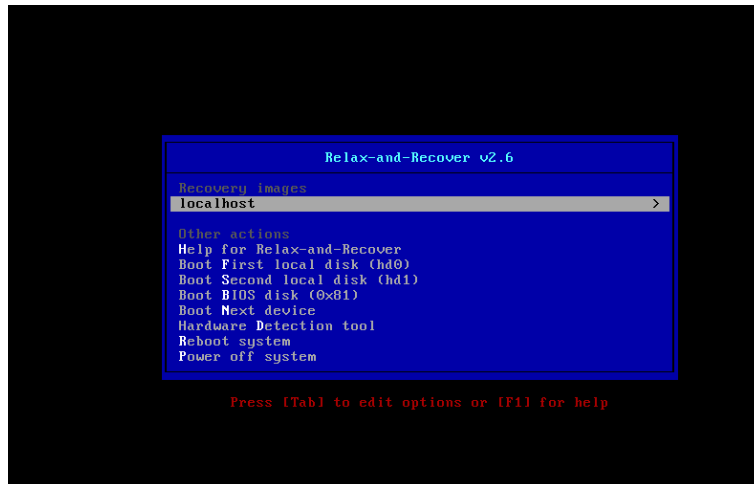


Figure 2.6. Boot menu of ReaR

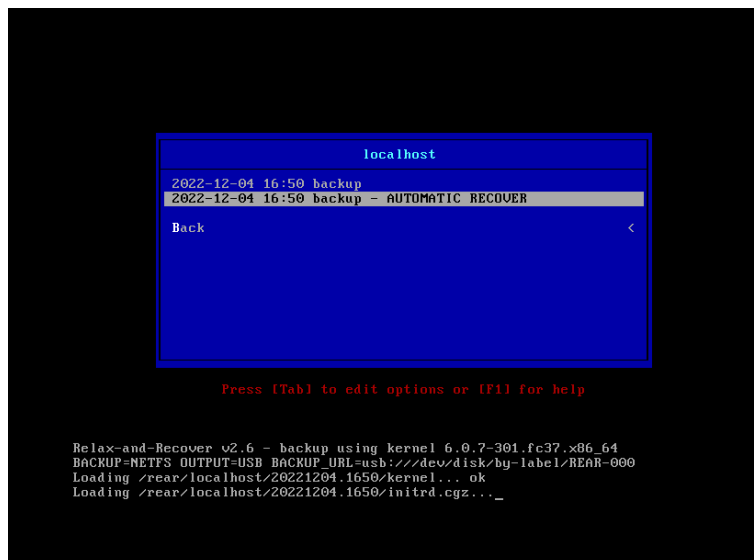


Figure 2.7. Automatic recover

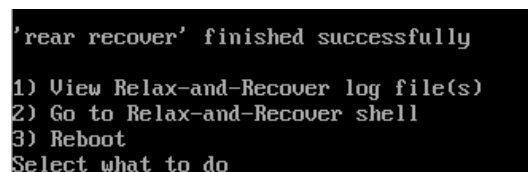


Figure 2.8. Rear ask for reboot in automatic mode

```
[root@localhost ~]# ls
file0 file1 file2 file3 file4 rear-2022-12-04T17:22:24+01:00.log
file5 file6 file7 file8 file9
```

Figure 2.9. Files after recover

After recovering and rebooting, we can see the files we suggested they'll stay (Figure 2.9). File `rear-2022-12-04T17:22:24+01:00.log` contains log outputs during recover part. Files `file5..file9` were “deleted” because they are not a part of original backup.

2.3 ReaR testing requirements

In the previous sections, I described a simple way to make a backup and restore the data. The work has aimed at the testing of ReaR.

ReaR needs two primary types of tests: a test of backup and recovery (as I showed in the previous section) and a static analysis of code.

Automatic testing of backup and recovery would help very fast and in a timely manner to detect regressions of basic functionality; it allows to speed up development because of needless manual testing the basic functionality. Also, it can help prevent new bugs and errors due to the absence of unpredictable and non-obvious testing. Developers may occasionally miss testing cases because of the changed module that affects others components of the software.

Every software has its code base and rules on how to write code. But developers often make similar mistakes in code that conflict with the language specification they are using. ReaR needs a static analysis for its code base to prevent such errors.

Because of open source, new developers may have reasonable solutions that need to be corrected. To simplify integrating suggested solutions, ReaR needs to have continuous integration testing. Such an approach allows making testing regardless of who suggests changes.

To satisfy the testing described above, we need an infrastructure that allows making these tests. We need an infrastructure that allows the following:

- install a ReaR package with the changes from the PR
- configure the second disk where we will place the backup
- possible to reboot the system several times
- has enough space to pace backup

Chapter 3

Infrastructure for ReaR

In this chapter I defined and described what continuous integration (CI) is, how we can use it and what type infrastructure we need. I tested few infrastructures and choosed certain that matches our demands.

3.1 Continuous integration

“Continuous integration was first written about in Kent Beck’s book Extreme Programming Explained (first published in 1999). As with other Extreme Programming practices, the idea behind continuous integration was that, if regular integration of your codebase is good, why not do it all the time? In the context of integration, “all the time” means every single time somebody commits any change to the version control system.” [1]

Another words, continuous integration is a set of steps that may verify correctness of our changes, and detect usual errors and mistakes. It includes such steps as building, installing, and testing. And our requirements is to make it for committed changes. So, when whoever make any pull request for ReaR project on GitHub, we would like to have a system setup that allowed to trigger building, installing, testing. And all the steps must be automated in some way, because our goal is to simplify development and minimize errors introductions.

In following chapter I am going to introduce a software that allows making CI.

3.2 Building

Since I started the project with Pavel Cahyna who is an employee of Red Hat, they are using a Packit infrastructure for building packages for Fedora Linux and CentOS Linux.

“An open source project aiming to ease the integration of your project with Fedora Linux, CentOS Stream and other distributions.”¹

It is possible to integrate Packit in your project on GitHub. Packit has functions that triggers building on different actions on GitHub: pushes, making pull requests, merging... After successful building it reproduces a public link which is possible to use for installing package with needless to setup your server with repository or download manually the package. Enough to have an access to internet.

Packit corresponds our requirements for building that is possible to trigger by any event on GitHub. In the chapter 4 I showed how to set-up Packit building in your repository on GitHub.

To enable on my GitHub Packit I should ask for accessing to Fedora Account System account.²

¹ <https://packit.dev/>

² <https://docs.fedoraproject.org/en-US/fedora-accounts/user/>

Moreover, Packit has quite interesting features. It can automatically recognize new pull request that is made into GitHub repository and build with the changes a new package. Packit supports interaction via GitHub conversation in opened pull requests. It can be forced to do according command. For example, `packit build` will force Packit to build or rebuild packages; `packit test` will force Packit to run configured tests.

3.3 Infrastructure for backup and recovery testing

The most important part of the work is making automated testing for backup and recover ReaR usage. There were 3 main infrastructure I have tested:

- Travis CI is a cloud infrastructure that in past was free for open-source projects. ³ I registered on their web-site and, unfortunately, I had not found a free usage.
- GitHub Actions. It is very powerful infrastructure that is integrated into GitHub. The big benefit of usage GitHub Actions is that simple to build your project, because the source codes are on the system you have choosen. Another thins is that you do not need additional settings and manipulating with your project, enough to create a *workflow* with a script you want it to run. And another one is that GitHub offers a variety of Linux distributions for usage. But it has a disadvantage that makes for us the usage impossible. GitHub Actions are finishing since the machine is rebooted. It is possible to install ReaR and make backup. But after rebooting the infrastructure recognize the machine as turned-off. The lack of reboot does not correspond to the requirements from section 2.3.
- Testing Farm. “Testing Farm is a reliable and scalable Testing System as a Service for Red Hat internal services, Red Hat Hybrid Cloud services, and open source projects related to Red Hat products. It is commonly used as a test execution back-end of other services or CI systems. Thanks to its HTTP API it can be easily integrated into any other service.” ⁴ From the description it is seen that the infrastructure corresponds our requirements. ReaR is supported by Red Hat, employees from Red Hat develop the software. Also, thanks the API and that Packit and Testing Farm are Red Hat’s products, they have close integrated functionality. We can configure Packit for execution our tests, and Packit allows execute tests in Testing Farm without difficult configuration. Along with the building system the Testing Farm looks like more appropriate infrastructure. Additionally, I verified that Testing Farm supports reboot of their machines.

Referred to described above, and having experience with Packit, I choosed Testing Farm as infrastructure for CI.

³ <https://www.travis-ci.com/>

⁴ <https://docs.testing-farm.io/general/0.1/index.html>

Chapter 4

ReaR testing in infrastructure

In this chapter, I described how to execute tests in chosen infrastructure in conjunction with Test Management Tool (TMT). I demonstrated a simple example of how to set up TMT on a local VM and execute a test. Then I extended existing tests for ReaR and wrote my tests for backup in an ISO file and on an NFS server.

For part of testing an ISO backup, I mainly showed the steps to get a successful result. Because I executed the tests in the Testing Farm, which is running in a cloud, I don't have access to it; it was challenging to get some log files, understand and investigate the problems I encountered, and do a debug and invent a technique for that. It is a familiar approach or method that can be used in some way. The section may be helpful for people who encounter familiar problems. It is just a list of different techniques and experiments of myself which I tried.

Although the multi-host (backup on an NFS server as I described it in section 2.1) has yet to be done, I shortly described the idea of such tests. I tested it and verified that the test works correctly. Developers from Testing Farm have not done work with multi-host testing, and that was a reason why I have not tested backup on an NFS server in the cloud. My supervisor and I contacted the developers of TMT and Testing Farm with a concept we wanted to test. Then they sent us a concept for multi-host testing, which I tested for them locally on my own VMs.

4.1 Test Management Tool (TMT)

The TMT tool provides a user-friendly way to work with tests. You can comfortably create new tests, safely and efficiently run tests across different environments, review test results, debug test code, and enable tests in the CI using a consistent and concise config.

The python module and command-line tool implement the Metadata Specification, which allows storing all needed test execution data directly within a git repository. Together with the possibility to reference remote repositories, it makes it easy to share test coverage across projects and distros.

“The Flexible Metadata Format *fmf* is used to store data in both human and machine-readable ways close to the source code. Thanks to inheritance and elasticity, metadata are organized in the structure efficiently, preventing unnecessary duplication.”¹

Let's say we have a project we want to test with the TMT.

```
[anton@fedora tmp]$ mkdir simple-project
[anton@fedora tmp]$ cd simple-project/
```

To initialize the TMT environment in the simple project is needed to execute *tmt init*, then the TMT tool creates a sub-directory *.fmf* with a test version.

¹ <https://github.com/teemtee/tmt>

```
[anton@fedora simple-project]$ tmt init
Tree '/tmp/simple-project' initialized.
To populate it with example content, use --template with mini,
base or full.
[anton@fedora simple-project]$ ls -la
total 0
drwxr-xr-x  3 anton anton  60 Dec  6 22:34 .
drwxrwxrwt 28 root  root  680 Dec  6 22:34 ..
drwxr-xr-x  2 anton anton  60 Dec  6 22:34 .fmf
[anton@fedora simple-project]$ cat .fmf/version
1
```

Now our project doesn't contain any tests. One of the helpful features of tmt is that the user can discover all tests in a project by running command `tmt run discover`. To be precise, if we skip `discover` tmt will run all tests we have assigned in the project.

```
[anton@fedora simple-project]$ tmt run discover
/var/tmp/tmt/run-004

/plans/default
discover
  how: fmf
  directory: /tmp/simple-project
  summary: 0 tests selected
  warning: No tests found, finishing plan.
[anton@fedora simple-project]$ tmt run
/var/tmp/tmt/run-005
```

Figure 4.1. Example of TMT discover tests

Let's add a simple test that prints us system information via command `uname -a` and print on the "screen" the following message via `echo` command `echo "Test has been passed."` TMT supports so-called *plans*. "Plans, also called L2 metadata, are used to group relevant tests and enable the CI. They describe how to discover tests for execution, how to provision the environment, how to prepare it for testing, how to execute tests, report results and finally how to finish the test job." ²

We can use the plans to describe exactly where we want to execute tests (on our VM, or tmt should use our virtual images), the sequence of scripts, and how many machines we need for the tests (more, for example, with NFS backup).

```
[anton@fedora simple-project]$ mkdir plans/
...
## Create a file which describes our simple test
[anton@fedora simple-project]$ cat plans/main.fmf
provision:
  how: virtual
  connection: system

execute:
  how: shell
```

² <https://tmt.readthedocs.io/en/stable/spec/plans.html>

```

script:
  - uname -a
  - echo "Test has been passed."

```

- provision - describes where we want to execute a test. In this case, “virtual” means that TMT downloads an image and starts a VM.
- executes - describes or, better to say, answers “how” we want to execute our tests and “what” we want to execute. In this example, I used *shell scripts* to get system information and print a string “Test has been passed.” In the next section, I’ll show how to execute the tests on own VM we have access to.

Now let’s run the test `tmt run`. As shown in Figure ?? tmt successfully executed two tests. It corresponds `uname -a` and `echo “Test has been passed.”` The logs files of TMT running is placed in `/var/tmp/tmt/`.

```

/plans
warn: The 'shell' execute method has been deprecated.
warn: Use 'how: tmt' in the execute step instead (L2).
warn: Set 'framework: shell' in test metadata (L1).
warn: Support for old methods will be dropped in tmt-2.0.
discover
  how: shell
  summary: 2 tests selected
provision
  how: virtual
  user: root
  key: []
  image: fedora
  memory: 2048 MB
  disk: 10 GB
  arch: x86_64
  progress: booting...
  summary: 1 guest provisioned
prepare
  summary: 0 preparations applied
execute
  how: tmt
  summary: 2 tests executed
report
  how: display
  summary: 2 tests passed
finish
  guest: stopped
  guest: removed
  summary: 0 tasks completed

total: 2 tests passed

```

Figure 4.2. TMT executes simple tests

In my case, the output we can find in the file `/var/tmp/tmt/run-013/log.txt`:

```

...
15:29:25          out: Linux testcloud 6.0.7-301.fc37.x86_64 #1
SMP PREEMPT_DYNAMIC Fri Nov 4 18:35:48 UTC 2022
x86_64 x86_64 x86_64 GNU/Linux

```

```
...
15:29:25          out: Test has been passed.
...
```

I'll demonstrate how to set up and execute the same test on the VM we created in the first chapter. Start the VM and get the IP:

```
[root@localhost ~]# ip addr | grep inet
  inet 127.0.0.1/8 scope host lo
  inet6 ::1/128 scope host
  inet 192.168.122.98/24 brd 192.168.122.255
  scope global dynamic noprefixroute enp1s0
  inet6 fe80::5054:ff:fe4f:2c55/64 scope link noprefixroute
```

The IP is *192.168.122.98*. When I create the VM, I set root password “pswd”. So, if we want to execute the same test on the VM we created in the first chapter, then the `main.fmf` should contain the following provision stage:

```
## template
provision:
  how: connect
  guest: hostname or ip address
  user: username
  password: password

## for my VM it is
provision:
  how: connect
  guest: 192.168.122.98
  user: root
  password: pswd
```

Now, if we run again `tmt run` it already executes on our VM (see Figure 4.3)

```
provision
  how: connect
  guest: 192.168.122.98
  user: root
  password: pswd
  summary: 1 guest provisioned
```

Figure 4.3. TMT executes simple tests on our VM

Log files `/var/tmp/tmt/run-014/log.txt` contains similar output as we got before, exception is for `output`

```
...
15:48:18          out: Linux localhost.localdomain
6.0.7-301.fc37.x86_64 #1 SMP PREEMPT_DYNAMIC Fri Nov 4 18:35:48
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
...
15:48:18          out: Test has been passed.
...
```

Even if the project aims to execute tests for backup and recovery in infrastructure, running the tests locally on VM is very useful. It allows us to verify the correctness of

such an approach locally and then use it in the cloud. We spend our time on something other than necessary debugging and waiting for some responses from the Testing Farm.

4.2 ReaR, recovery tests

In this section, I described ReaR-testing. It is a set of tests that I used as a basis and extended them. Below you can find an explanation of the tests, and I described the most important steps. I didn't write details all steps because that is unnecessary. If you are interested, follow the link.³ You can contact me or the test's author (Lukáš Zaoral) in case of questions.

rear-testing is a repository created by Lukáš Zaoral. The repository contains a set of tests for ReaR. It makes a backup and then restores an OS. It helps quickly to find regression and verify the basic functionality of ReaR. Lukáš Zaoral wrote the tests for a few boot firmware architectures: BIOS, UEFI, PowerVM, and s390x. The tests are working on the internal infrastructure of Red Hat. Since I am not an employee of the company, I do not have access to the infrastructure. That is why I did not use the actual tests and modified them as a part of the work.

- BIOS. “A basic input output system (BIOS) is the lowest level software component of a computer's operating system.” [2]
- “UEFI (Unified Extensible Firmware Interface) is a set of specifications written by the UEFI Forum. They define the architecture of the platform firmware used for booting and its interface for interaction with the operating system.”⁴
- “PowerVM is developed by IBM. It is server virtualization without limits. Businesses are turning to PowerVM server virtualization to consolidate multiple workloads onto fewer systems, increasing server utilization and reducing cost.”⁵
- s390x “z/Architecture, initially and briefly called ESA Modal Extensions (ESAME), is IBM's 64-bit complex instruction set computer (CISC) instruction set architecture, implemented by its mainframe computers. IBM introduced its first z/Architecture-based system, the z900, in late 2000.”⁶

PowerVM and s390x are particular architectures and depend on hardware such as processor architecture. That is the reason why I used BIOS and UEFI. For them, it is possible to configure VM and run the tests there. During familiarizing myself with the rear-testing I encountered a problem with UEFI. Further descriptions, investigations, and developments come out from rear-testing for BIOS.

My first step was to go through the test code and understand it. Then I tried to execute the tests.

In the head of the file are two includes:

```
. /usr/bin/rhts-environment.sh || exit 1
. /usr/share/beakerlib/beakerlib.sh || exit 1
```

“BeakerLib is a shell-level integration testing library providing convenience functions that simplify writing, running, and analyzing integration and black box tests.”⁷ BeakerLib allows writing tests that support different journalling phases (preparation, configuration, execution, getting results, etc.).

The essential features include:

³ <https://github.com/lzaoral/rear-testing>

⁴ <https://en.wikipedia.org/wiki/UEFI>

⁵ <https://www.ibm.com/products/ibm-powervm>

⁶ <https://en.wikipedia.org/wiki/Z/Architecture>

⁷ <https://github.com/beakerlib/beakerlib>

- “Journal - uniform logging mechanism (logs & results saved in flexible XML format, easy to compare results & generate reports)”
- “Phases - logical grouping of test actions, clear separation of setup / test / cleanup (preventing false fails)”
- “Asserts - common checks affecting the overall results of the individual phases (checking for exit codes, file existence & content...)”
- “Helpers - convenience functions for common operations such as managing services, backup & restore”

8

Referring to the above, Lukáš Zaoral used the library for the rear-testing. The whole code is placed in the scope:

```
rlJournalStart
...
rlJournalEnd
```

It is an initialization of journalling functionality. As I wrote above, it is possible to divide our testing into different phases. That approach is used in the rear-testing.

To print any message *rlLog* command is used. For example:

rlLog “Select device for REAR” - that will print in log file the message on the right side from the literal. To execute command is used *rlRun*. Of course, we can use usual shell commands without the *rlRun* but they won’t be visible in the log file that BeakerLib creates. For example, *rlRun* is executed in the following way:

```
rlRun -l "lsblk" 0 "lsblk executed"
```

-l is an option that says we want to log output of the command via *rlLog*.

lsblk is a command that we executed.

0 is status which we expect the command o program would return. It may differs if you use another program. In our case it is 0.

lsblk executed is a comment that describe the *rlRun*.

rlFileSubmit commands allows to submit a file from the machine we tested on. The file would appear among logs.

Variable *PACKAGES* stores a list of packages needed for the testing. Obviously, it is *rear* package, and *syslinux-extlinux*. Package *syslinux-extlinux* has the following contents:

```
[anton@fedora ~]$ dnf repoquery -l syslinux-extlinux
/etc/extlinux.conf
/sbin/extlinux
/usr/lib/.build-id
/usr/lib/.build-id/fc
/usr/lib/.build-id/fc/17033b5213bd5c04a72ff38e4c580ead46c12b
/usr/share/man/man1/extlinux.1.gz
```

During the test, the machine will be rebooted several times. From the tests’ point of view, the machine will be rebooted once (after the backup is made). The machine reboots more times: after backup, after the restore stage, and the last, after relabeling partitions. The *REBOOTCOUNT* variable is used to define the number of reboots. *REBOOTCOUNT* contains the value of the count of reboots during the test. It is

⁸ <https://github.com/beakerlib/beakerlib/wiki/man>

used to know if we just started the test and we need to make a backup, then *REBOOTCOUNT* is equal to 0. Or we have already restored the system, and we are in the second stage when we want to decide if the test has been passed or failed, then *REBOOTCOUNT* is equal to 1.

Of course, during the test, the tested machine reboots several times. Since the TMT framework uses *ssh* protocol for manipulating and executing the commands on the machine, it sees the booted machine only phase of backup and after the recovery stage. Because I turned off the *ssh* daemon process in the configuration file for the rescue system, TMT does not “think” we boot the system and can continue running the test scripts in rescue mode. In this way, we reboot the machine once from the TMT side only.

The following code asserts that all packages in variable *PACKAGES* are installed:

```
rlPhaseStartSetup
    rlAssertRpm --all
rlPhaseEnd
```

It uses a second disk for backup store. The following configuration for ReaR match it:

```
rlPhaseStartSetup
    rlFileBackup "/etc/rear/local.conf"
    rlRun "echo 'OUTPUT=USB"
BACKUP=NETFS
BACKUP_URL=usb:///dev/disk/by-label/REAR-000
ISO_DEFAULT=automatic
ISO_RECOVER_MODE=unattended' | tee /etc/rear/local.conf" 0\
"Creating basic configuration file"
    rlAssertExists "/etc/rear/local.conf"
rlPhaseEnd
```

ISO_RECOVER_MODE="unattended" it sets automatically execution of *rear recover* command and *reboot* after successful execution. If we also set *ISO_DEFAULT*="automatic", the restore will be full-automated. There is no need for any interaction with the interface.

Before the backup we create a file *drive_layout.old* which stores information about block devices:

```
rlRun -l "lsblk | tee drive_layout.old" 0 "Store lsblk output\
in recovery image"
```

The file will be a part of a backup. After the restore stage, we should also see the file. It is possible to configure ReaR not to include some files in the backup. But by default, */root* directory (because in Unix-like systems directory is a file. Actually, all is a file.) is included.

The next step is to make a backup:

```
rlPhaseStartTest
    rlRun "rear -d mkbackup" 0 "Creating backup to $REAR_ROOT"
    rlFileSubmit /var/log/rear/rear*.log rear-mkbackup.log
    if ! rlGetPhaseState; then
        rlDie "FATAL ERROR: rear -d mkbackup failed.\
```

```

    See rear-mkbackup.log for details."
fi
rlPhaseEnd

```

Variable `REAR_ROOT` contains the relative path to the block device we make a backup on. It is just for logging and is not necessary to have such a variable. Also, we want to submit a log file with info during the backup stage. If the command `rear mkbackup` hasn't been done successfully, we will fail and not continue the test.

After it, we make one more file. This file will not be restored as well because we created it after the backup stage.

```

rlPhaseStartSetup
  rlRun "touch recovery_will_remove_me" 0\
  "Create dummy file to be removed by recovery"
  rlAssertExists recovery_will_remove_me
rlPhaseEnd

```

The last steps before reboot may look complicated for a reader. Because we have the backup on the second disk, we need to change chain-loading in GRUB. We set in the GRUB configuration file that, by default, for boot is assigned the second disk. But for unknown reasons, it failed in boot with the message: “Missing operating system. No bootable device”, Figure 4.4



Figure 4.4. No bootable device

I spent some time trying to realize why the problem occurred. Eventually, I rewrite it with `extlinux` usage. I rewrite the super-block of the boot partition by the `extlinux` binary file, and for now, it uses `extlinux`. You can see the change in the following commit on GitHub.⁹

“EXTLINUX is a Syslinux variant which boots from a Linux filesystem.”¹⁰ “SYS-LINUX is a boot loader for the Linux operating system which runs on an MS-DOS/Windows FAT filesystem. It is intended to simplify first-time installation of Linux, and for creation of rescue and other special purpose boot disks.”¹¹

⁹ <https://github.com/lzaoral/rear-testing/commit/7621cccd842074ea5e6f96a1f49c47f1b87a118c>

¹⁰ <https://wiki.syslinux.org/wiki/index.php?title=EXTLINUX>

¹¹ <https://wiki.syslinux.org/wiki/index.php>

So, I used *extlinux* as a loader instead of GRUB. We have already installed the package with *extlinux* (for RPM packages it is *syslinux-extlinux* package). To initialize *extlinux* for boot directory is needed to execute the following command:

```
extlinux --install /boot/extlinux
```

extlinux needs entries (similar as GRUB needs it). The main configuration file for *extlinux* is `/boot/extlinux/extlinux.conf`. To boot from the second disk (*hd1* name from *extlinux* side) that has the rescue system, we need the following entry:

```
LABEL rear
  MENU LABEL Chainload ReaR from hd1
  MENU DEFAULT
  COM32 chain.c32
  APPEND hd1
```

chain.c32 - it is a binary program for so-called chain-loading. That allows us to load another bootloader. In our case, it will load *syslinux*, which was created by the *rear* command, and then boot the rescue system.

MENU DEFAULT - set the entry by default for loading.

MENU LABEL Chainload ReaR from hd1 - menu name for the entry.

APPEND hd1 - to use the second disk for boot.

I used the following command to replace GRUB loader with the *extlinux* by rewriting super-block:

```
cat /usr/share/syslinux/mbr.bin > /dev/$ROOT_DEVICE
```

ROOT_DEVICE - it is a variable with a block device name corresponding to the boot partition.

Now the test makes reboot by *rhts-reboot* command.

After reboot, TMT run the test script again, it starts from the beginning. We got to the second part by if-condition that *REBOOTCOUNT* equals 1. There the test verifies that the file doesn't exist, and logs from *ReaR* and *drive_layout.old* still are after restoration. And additionally, the test confirms that doesn't happen a change with the list of block devices:

```
rlAssertNotExists recovery_will_remove_me
rlAssertExists drive_layout.old
rlAssertExists /root/rear*.log

rlRun -l "lsblk | tee drive_layout.new" 0\
"Get current lsblk output"
if ! rlAssertNotDiffer drive_layout.old drive_layout.new; then
  rlRun -l "diff -u drive_layout.old drive_layout.new" \
  1 "Diff drive layout changes"
fi
```

4.3 Execute TMT test in an infrastructure

In the second chapter, I described the Testing Farm infrastructure and Packit system, which allows the building of a package and running tests in the Testing Farm for the built package. In this section, I described how to set up *ReaR* GitHub repository to execute tests for PRs. Lately, we will use it to perform the test described in the previous section.

4.4 Testing recovery over NFS

In this section, I described and showed a solution for automatically testing a backup on NFS server.

“The Sun Network Filesystem (NFS) provides transparent, remote access to filesystems. Unlike many other remote filesystems implementations under UNIXt, the NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation(XDR) specification to describe protocols in a machine and system-independent way. The NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.” [3]

From the description of NFS, such technologies are convenient if we want to make a backup and store it remotely. ReaR supports a making backups in this way. TMT supports executing tests on a few machines simultaneously. So it is helpful to start two machines: the first one may be used as a client which makes backup, and the second may be used as an NFS server to store the backup. I wrote the test as a proof-of-concept to demonstrate TMT and Testing Farm developers what kind of usage for TMT and the infrastructure we needed. Unfortunately, such a solution cannot be integrated into the current infrastructure. Referred above, I chose Testing Farm for automatic tests. Pavel Cahyna and I asked TMT and Testing Farm developers if they support that testing. They confirmed they are working on multi-host support and sent me back a demo version of TMT with supported multi-host testing. Eventually, I verified their solution, and it works as we expected, but it still needs to be introduced in Testing Farm. I am going into this work to show a reader how to set up it, up because it is essential and valuable for the future expanding variable of testing for ReaR.

To set ReaR to use NFS for backup I used the following configuration:

```
BACKUP_URL="nfs://$NFS_SERVER_IP/var/tmp/nfsshare"
GRUB_RESCUE=y
```

- `NFS_SERVER_IP` - variable contains an IP of NFS server
- `/var/tmp/nfsshare` - path to a directory which is shared via NFS (actually, it is a path on a server where ReaR places a backup)
- `GRUB_RESCUE=y` - I used it to simplify boot. The flag asks the rear to set up in the GRUB rescue system boot. The rescue image and kernel is placed on the same disk that is used by the system.

Now we will configure two machines to run a test. Then I described what is different and what I modified in the *rear-testing*.

We need to have 2 VMs: NFS - client and NFS - server. It is possible just to clone in the Virtual Manager machine we created in the first chapter. In the Figure 4.5 is an example of how to create a copy of the VM. I also copied the disk storage where Fedora 37 is installed. If we don't copy the disk, then the client and server will refer to the same disk, which leads to data corruption. Also, I excluded the disk for backup because we will store the backup on the server.

Boot the two machines and get the IP. We need it for further configuration. In my case it was:

```
client: 192.168.122.98
server: 192.168.122.208
```

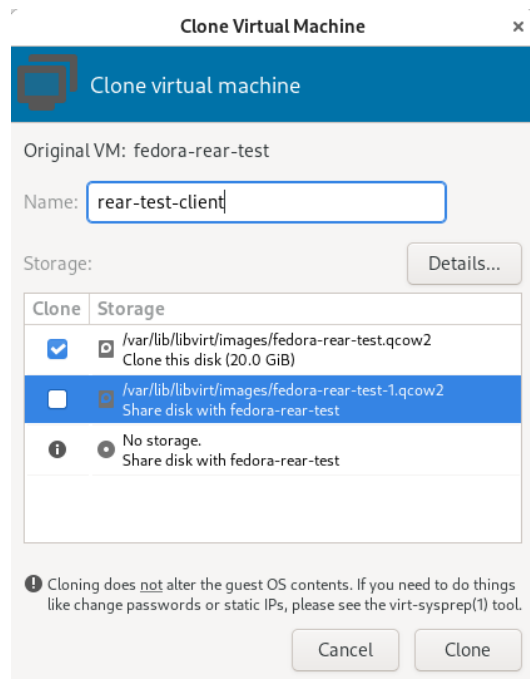


Figure 4.5. VM cloning

Let's add a plan for ReaR where we placed code to execute tests referred to above from my GitHub branch `test-multihost`.¹²

I described the process in code-block bellow, with comments marked by `##`:

```
## Clone a project from my fork on GitHub
[anton@fedora tmp]$ git clone https://github.com/antonvoznia/rear.git
Cloning into 'rear'...
remote: Enumerating objects: 57078, done.
remote: Counting objects: 100% (254/254), done.
remote: Compressing objects: 100% (145/145), done.
remote: Total 57078 (delta 123), reused 155 (delta 102), pack-reused 56824
Receiving objects: 100% (57078/57078), 10.72 MiB | 4.11 MiB/s, done.
Resolving deltas: 100% (28520/28520), done.
[anton@fedora tmp]$ cd rear/
## Create a new branch which we will use for the tesing.
[anton@fedora rear]$ git checkout -b tmt-test-multihost
Switched to a new branch 'tmt-test-multihost'
## Create a new plans
[anton@fedora rear]$ mkdir -p plans/multihost
## Initiate TMT tool envirenmet for ReaR
[anton@fedora plans]$ tmt init
Tree '/var/tmp/rear/plans' initialized.
To populate it with example content, use
--template with mini, base or full.
[anton@fedora rear]$ cd plans/multihost/
...
## Create 2 files. main.fmf contains information
```

¹² <https://github.com/antonvoznia/rear-testing/tree/test-multihost>

```

## about where we want to execute and how.
## sanity.fmf refers on a branch in my GitHub repo
## rear-testing.
[anton@fedora multihost]$ ls
main.fmf  sanity.fmf

rovision:
  - name: client
    how: connect
    guest: 192.168.122.98
    user: root
    password: pswd

  - name: server
    how: connect
    guest: 192.168.122.208
    user: root
    password: pswd

prepare:
  how: shell
  where: server
  script:
    - dnf -y install nfs-utils
    - mkdir /var/tmp/nfsshare
    - echo "/var/tmp/nfsshare 192.168.0.0/16(rw,no_root_squash)" \
    >> /etc/exports
    - systemctl enable --now rpcbind nfs-server
    - firewall-cmd --add-service=nfs --permanent
    - firewall-cmd --reload

# Use the internal executor
execute:
  how: tmt
  where: client

```

That is a stage for the configuration NFS server. This will be running before the *execute* step. To clarify, the commands may be executed for Linux distribution with the DNF package manager.

- *dnf -y install nfs-utils* - install utils for NFS server configuration
- *mkdir /var/tmp/nfsshare* - create a directory which will be shared across NFS
- *echo "/var/tmp/nfsshare 192.168.0.0/16(rw,no_root_squash)" » /etc/exports* - adding in file */etc/exports* configuration for a directory we want to share (*/var/tmp/nfsshare*). *192.168.0.0/16* represents sub-net with 16 bits of hosts. *rw* Allow both read and write requests on this NFS volume. *no_root_squash* Turn off root squashing.
- *systemctl enable --now rpcbind nfs-server* - enable daemon process for NFS server.
- the 2 last commands are for excepting NFS from firewall checking. In some cases it can block NFS access.

```

provision
  how: connect
  name: client
  guest: 192.168.122.98
  user: root
  password: pswd

  how: connect
  name: server
  guest: 192.168.122.208
  user: root
  password: pswd

  summary: 2 guests provisioned
prepare
  how: multihost
  summary: Setup guest for multihost testing
  name: multihost
  guest: client

  how: install
  summary: Install required packages
  name: requires
  guest: client
  package: beakerlib

  how: multihost
  summary: Setup guest for multihost testing
  name: multihost
  guest: server

  how: shell
  guest: server
  where: server
  overview: 6 scripts found

  how: install
  summary: Install required packages
  name: requires
  guest: server
  package: beakerlib

  summary: 5 preparations applied
execute
  how: tmt
  summary: 1 test executed
report
  how: display
  summary: 1 test passed
finish
  summary: 0 tasks completed

total: 1 test passed
[anton@fedora plans]$

```

Figure 4.6. TMT run multi-host tests

Remain to run the command `tmt run`. The example is in Figure 4.6

4.5 Run test in infrastructure

In this section, I demonstrated how to execute a test in infrastructure. We will write a simple test for ReaR (to be more clear, the test will not be about ReaR specifics or ReaR testing, it will be just an example). I have already described Packit. It solves two problems for us: build a ReaR package with changes proposed by us, and it executes tests in the infrastructure Testing Farm. It is a very useful and important point. We

want to test precisely the new arrived changes. Packit allows quickly build a package for the architecture we denoted.

There is a guide on how to set up a Packit for your account, whether on GitHub or GitLab. It is possible to activate Packit on a certain repository but not for all repositories.

We need to create a configuration file to force Packit to build and run tests. Packit uses configuration files in YUML format. Valid names for the configuration files are restricted by:

- `.packit.yaml`
- `.packit.yml`
- `packit.yaml`
- `packit.yml`

One of the files I mentioned above needs to be placed in the repository's root directory. ReaR project already has enabled Packit. I needed to do the same for my repository. We need to expand existed `.packit.yaml` file and add the test we want to execute. In section 4.1, we wrote a simple test. It is possible to execute the same test in the Testing Farm infrastructure. Let's clone the ReaR from my repository and configure Packit.

```
[anton@fedora tmp]$ git clone https://github.com/antonvoznia/rear.git
Cloning into 'rear'...
remote: Enumerating objects: 57083, done.
remote: Counting objects: 100% (297/297), done.
remote: Compressing objects: 100% (147/147), done.
remote: Total 57083 (delta 163),
reused 198 (delta 143), pack-reused 56786
Receiving objects: 100% (57083/57083), 10.73 MiB | 22.32 MiB/s, done.
Resolving deltas: 100% (28523/28523), done.
[anton@fedora tmp]$ cd rear/
[anton@fedora rear]$ cat .packit.yaml
downstream_package_name: rear
jobs:
- job: copr_build
  targets:
  - fedora-all
  - centos-stream-8-x86_64
  - centos-stream-9-x86_64
  - opensuse-leap-15.3-x86_64
  - opensuse-tumbleweed-x86_64
  trigger: pull_request
- job: production_build
  scratch: True
  targets:
  - fedora-latest-stable
  - epel-all
  trigger: pull_request
specfile_path: packaging/rpm/rear.spec
files_to_sync:
- .packit.yaml
- dest: rear.spec
```

```
src: packaging/rpm/rear.spec
upstream_package_name: rear
```

The most important part of `.packit.yaml` for us is `jobs` section. Packit defines the `jobs` that needs to do. In the example above are 2 jobs: `copr_build` and `production_build`. The first `copr_build` submits a task into Fedora CORP¹³.

“Copr is an easy-to-use automatic build system providing a package repository as its output.

Start with making your own repository in these three steps:

- 1) choose a system and architecture you want to build for
- 2) provide Copr with `src.rpm` packages

3) let Copr do all the work and wait for your new repo” `targets` - a list of distributions the build is for. In our case, it is for all active releases of Fedora, CentOS 8 and 9, and 2 versions of OpenSUSE. `trigger` - an event on GitHub (GitLab) that triggers the Packit to apply for a job. In the example above, it is a pull request for our repository. For every new PR Packit applies a new task.

The second job `production_build` submits a task for another build system - Fedora Koji build system.¹⁴ `targets` in the second jobs are Fedora last and stable versions (it may differ in the future, by now, it is from Fedora 35 up to Fedora 37) and Extra Packages for Enterprise Linux (EPEL) for CentOS 7, CentOS 8, and CentOS 9.

Now, let’s copy `plans/main.fmf` from section 4.1 where I showed how to execute the test with TMT.

```
[anton@fedora rear]$ tmt init
...
[anton@fedora rear]$ cat plans/main.fmf
provision:
  how: virtual
  connection: system
execute:
  how: shell
  script:
    - uname -a
    - echo "Test has been passed."
```

To trigger tests to execute in Testing Farm we need add own job:

```
- job: tests
  trigger: pull_request
  metadata:
  targets:
    - fedora-latest-stable
    - centos-stream-8-x86_64
    - centos-stream-9-x86_64
```

Now we will commit the change and push it into our repository. Then we should create a PR against the master branch (it would be a different branch, but I made it with the most similar branch with minor changes). After a creating PR in it in sub-section “Checks” we can see the jobs we have defined in `.packit.yaml`. Figure 4.7

¹³ <https://copr.fedorainfracloud.org/>

¹⁴ <https://koji.fedoraproject.org/koji/>

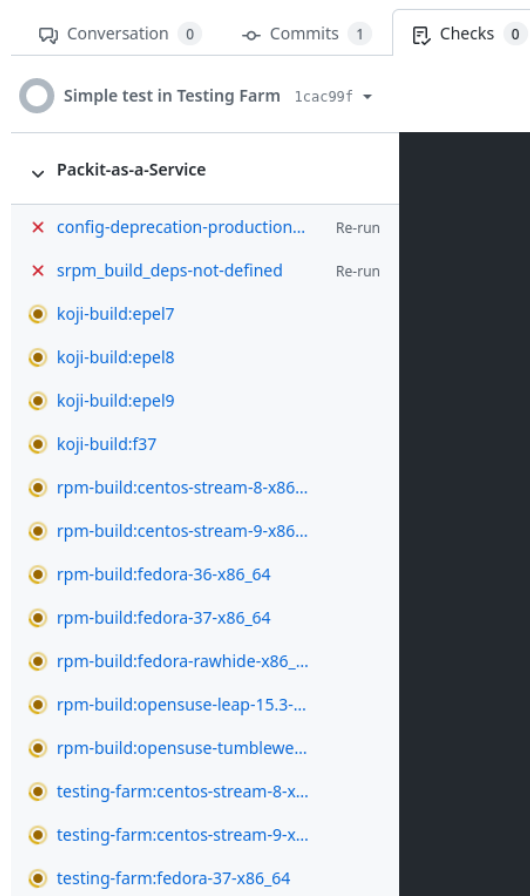


Figure 4.7. Simple test in Testing Farm

The red “X” means that the test is failed, the status bar - the test is in progress, and the green check mark - the test is passed. If we open the test for Fedora 37, which had been passed (Figure 4.8) and click for the log file

In the log-file *log.txt* are two lines with the output for the shell scripts we used for the test. For `uname -a` the output is:

```
10:22:15 out: Linux
ip-172-31-23-184.us-east-2.compute.internal 6.0.12-300.fc37.x86_64
#1 SMP PREEMPT_DYNAMIC Thu Dec 8 16:58:47 UTC 2022 x86_64
x86_64 x86_64 GNU/Linux
```

For the command `echo "Test has been passed."`:

```
10:22:15 out: Test has been passed.
```

4.6 Type of backup

Referred to above, ReaR supports different formats to store rescue image. It is possible to store an output of rescue image in:

- RAMDISK - create a copy of kernel and initramfs in selected location
- ISO - create ISO ISO9660, that is bootable
- PXE - create on a remote server wheret PXE or NFS selected files

Simple test in Testing Farm #359

🔗 Open antonvoznia wants to merge 3 commits into `master` from `packit-run-in-testing-farm`

🗨 Conversation 0 ↶ Commits 3 📄 Checks 16 📁 Files changed 3

🟢 remove deprecated 'how: shell' 51674da ▾

- 🟡 testing-farm:centos-stream-8-x...
- 🟡 testing-farm:centos-stream-9-x...
- ✓ koji-build:epel7
- ✓ koji-build:epel8
- ✓ koji-build:epel9
- ✓ koji-build:f37
- ✓ rpm-build:centos-stream-8-x86...
- ✓ rpm-build:centos-stream-9-x86...
- ✓ rpm-build:fedora-36-x86_64
- ✓ rpm-build:fedora-37-x86_64
- ✓ rpm-build:fedora-rawhide-x86_...
- ✓ rpm-build:opensuse-leap-15.3-...
- ✓ rpm-build:opensuse-tumblewee...
- ✓ testing-farm:fedora-37-x86_64

Packit-as-a-Service / testing-farm:fedora-37-x86_64
succeeded now in 0s

Tests passed ...

Name/Job	URL
Dashboard	https://dashboard.packit.dev/results/testing-farm/157577
Testing Farm	https://artifacts.dev.testing-farm.io/e21c58f8-8f21-492e-a19b-13d963b050de

🔗 View more details on Packit-as-a-Service

Figure 4.8. Simple test passed

- USB - create a bootable disk with backup (it doesn't have to be usb exactly, it may be another type of disk)
- RAWDISK - create a bootable raw disk

In section 4.2, I described how to execute the recovery tests for ReaR from Lukáš Zaoral. It uses a USB format and makes a backup on the second disk. Using the same test in the Testing Farm infrastructure would be easy. But unfortunately, the Testing Farm does not yet support the option to add additional hardware. That means we cannot just add the second disk. Developers from Testing Farm replied that they would introduce such a feature in the near future. By default, Testing Farm has only one disk. To get the information about images and setups used in Testing Farm, I wrote a very simple plan (plans were described in section 4.1):

```
execute:
  script:
    - free -h
    - df -h
    - lsblk
```

I used it to collect different information about the machines in the infrastructure, such as free memory (`free -h`), free and used disk spaces (`df -h`), a list of block devices (`lsblk`). The list of block devices has only one disk `nvme0n1` with different partitions.

Filesystem	Size	Used	Avail	Use%	Mounted on
devtmpfs	4.0M	0	4.0M	0%	/dev
tmpfs	1.9G	0	1.9G	0%	/dev/shm
tmpfs	772M	572K	772M	1%	/run
/dev/nvme0n1p5	39G	772M	38G	2%	/
tmpfs	1.9G	4.0K	1.9G	1%	/tmp

```

/dev/nvme0n1p2 966M   53M  848M   6% /boot
/dev/nvme0n1p5 39G   772M 38G   2% /home
/dev/nvme0n1p3 100M   12M  89M  12% /boot/efi
tmpfs          386M    0  386M   0% /run/user/0
Shared connection to 18.118.36.202 closed.

```

The base test storing the backup on the second disk is not suitable for our case. We could make a backup over NFS. It would work as I described in section 4.4. Also, in the section, I wrote why it is not yet possible to use the method; Testing Farm still needs to support the multi-host. I could write two tests that would start simultaneously. The tests would trigger starting of two independent machines in the infrastructure. One of the test's two machines could be used as a server, and another is a client. But there are a lot of problems like getting IPs of server and client, interconnecting, synchronization issues, etc.

I mentioned *syslinux* and *extlinux* programs in previous sections. One part of the project (*syslinux*) is a program *memdisk*. Memdisk is a binary program that allows boot different legacy operating systems. And memdisk supports boot ISO images. If we store a backup and ReaR's kernel in an ISO image, we may use memdisk to boot from the image. It would work in the following way: a firmware (let's say BIOS) will start and load GRUB, the GRUB has a configuration to load the *memdisk*, the *memdisk* has a configuration to load the image we have set. In our case, it is the ISO file that was created after backup by ReaR. An advantage of the *memdisk* usage is that we can just set a relative path to the ISO file with our backup to memdisk configuration. We do not need additional hardware. And memdisk is possible to load from GRUB. It means we do not need to rewrite the super-block in the boot partition.

Probably *memdisk* is not capable to boot large ISO images. Because the project aims to boot legacy OS that took just few hundreds megabytes. I encounter a problem during my work and solve it. I described it in the following section.

Making a backup in ISO image is the most convenient solution for testing in the infrastructure at this time.

As well, GRUB supports booting from the ISO image but in a different way. GRUB uses ISO as a file with a path to a bootable kernel. Memdisk allows the use ISO, similar to CD/DVD driver, because we want to test in a very similar way it would be used in real cases. And when we set up the GRUB to boot the ISO, we should configure the path to the rescue image and kernel we want to boot. That means GRUB loads at once the rescue system at; the menu in Figures 2.6 and 2.7 is not showing. In other words, we also do not test the whole ISO image. We skip the phase and start the recovery stage.

Also, there is a reasonable question. If we have a backup in our ISO image on only one disk, that is the same for our system; and during the restore stage, we will rewrite the disk and rear the backup from the disk at the same time. That is all cause of data corruption. So, how can we make restore? We can use ramdisk. It is possible to create a disk in RAM.

“The RAM disk driver is a way to use main system memory as a block device. It is required for *initrd*, an initial filesystem used if you need to load modules in order to access the root filesystem (see Documentation/admin-guide/initrd.rst). It can also be used for a temporary filesystem for crypto work, since the contents are erased on reboot.”¹⁵

¹⁵ <https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/blockdev/ramdisk.rst>

When ReaR has been booted in memory, we will create a copy of the ISO image in ramdisk. ReaR will see it as a CD with the label “REAR-ISO.” ReaR will recognize the disk by the label “REAR-ISO” and start the restore stage. From the ReaR side, it would look like we use a CD driver, though the ISO is loaded in RAM. Notice we should create a ramdisk before ReaR starts recovery. Because during the recovery stage, it formats the disks and deletes the ISO file with the backup.

4.7 ISO backup. Modification recovery test for infrastructure

In this section, I described how I have modified and extended the tests mentioned above (recovery tests) to execute them in the Testing Farm infrastructure. I showed the problems I encountered and what I was trying to solve it. As a basis, I took recovery tests, which I have already extended for multi-host testing with over NFS. Also, I demonstrated how it is possible to debug in the cloud platform Testing Farm.

Though the base test exists, we could try to execute them in the infrastructure. The restriction for the execution of the tests is a lack of modification images in Testing Farm. Actually, Testing Farm uses AWS (Amazon Web Services) cloud. Currently, they do not have the option to add additional hardware or modify requirements to the images we need to use. The test used for testing ReaR needs the second disk where we place backups with the rescue system. The problem is that we can not (at least by now) add the second disk in the cloud machine. That is a reason why we need to extend the tests and adapt them to the Testing Farm infrastructure. Let’s say we need a “workaround” to set up testing in infrastructure. To solve it, I used a *memdisk*. It is a binary program that allows it to boot from an ISO image. *memdisk* usage makes us possible to boot from the ISO image and place the ISO file on the same disk the machine has.

Today most of the clouds use BIOS to boot an OS. The same is valid for the images used in Testing Farm. That is an argument for why I chose BIOS tests as a basis.

The first step I did was changing the ReaR configuration for backup in the file */etc/rear/local.conf*. We want ReaR after reboot to run automatic restore (needless interface interaction). So, we keep the following two variables:

```
ISO_DEFAULT=automatic
ISO_RECOVER_MODE=unattended
```

To force ReaR to create ISO image with a backup and bootable kernel, which starts the restoring, we set the following variables:

```
OUTPUT=ISO
BACKUP_URL=iso:///backup
```

OUTPUT=ISO says that ReaR will make ISO image, *BACKUP_URL=iso:///backup* make a backup a part of the ISO image. ReaR allows separate backup from the bootable kernel of ReaR. In our case, we want to keep it locally. And the best way to make it a part of the same ISO image.

The following variables make ReaR create backup locally (it does not copy backup onto the remote server, it keeps them locally)

```
OUTPUT_URL=null
BACKUP=NETFS
```

Notice that variables `OUTPUT_URL=null` and `BACKUP_URL=iso:///backup` we should use together because we want to create a local backup and, at the same time, make it a part of the ISO image. I removed the value `GRUB_RESCUE=y` from the configuration because it is not the usual usage of ReaR. And we want to have a very similar test to actual case usage. The `GRUB_RESCUE` enabled was helpful for an occurrence we have a backup on the NFS server. It forces ReaR to set up GRUB in a way that it will boot the rescue system after the reboot.

In the previous section, I mentioned that ReaR should use `ramdisk` to load the ISO image. That all should be done before ReaR starts restoring. Fortunately, in the ReaR configuration file might be a set variable containing a list of commands executed before restoring. The following code is responsible for it:

```
PRE_RECOVERY_SCRIPT=("mkdir /tmp/mnt;" "mount /dev/vda2 /tmp/mnt/;" \
"modprobe brd rd_nr=1 rd_size=2097152;" \
"dd if=/tmp/mnt/var/lib/rear/output/rear-fedora.iso \
of=/dev/ram0;" "umount /tmp/mnt/;")
ISO_FILE_SIZE_LIMIT=4294967296
```

The variable `PRE_RECOVERY_SCRIPT` contains the following steps:

1. `mkdir /tmp/mnt` - create a temporary directory where we will mount the partition with `root` directory.
2. `mount /dev/vda2 /tmp/mnt/` - mount the `root` directory into `/tmp/mnt/`.
3. `modprobe brd rd_nr=1 rd_size=2097152` - it loads the `brd` (*block ram disk*) module, and the disk will be created when the module will have been loaded. `rd_nr=1` - count of disks we want. `rd_size=2097152` - the size of the disk in kilobytes.
4. `dd if=/tmp/mnt/var/lib/rear/output/rear-fedora.iso of=/dev/ram0` - copy the ISO image in ramdisk we have created.
5. `umount /tmp/mnt/` - unmount the mounted `/dev/dsk/vda2`.

After the steps above, we will have the ISO image in ramdisk. ReaR would find it by the label “REAR_ISO” during the restore stage.

The value `ISO_FILE_SIZE_LIMIT=4294967296` extends the standard size of the ISO image. By default value in ReaR, it is not enough.

The next steps with creating files before and after the command execution `rear -v mkbackup` I left the same. Because the basic logic of the test verifies if the default functions of ReaR works correctly, as before, we expect that after the restore stage, the file `drive_layout.old` will remain, and the file `recovery_will_remove_me` will be deleted.

The following code adds to the GRUB entry with boot from the ISO:

```
rlRun "echo 'menuentry \"ReaR-recover\" {'
loopback loop \
(hd0,msdos3)/root/var/lib/rear/output/rear-fedora.iso
linux (loop)/isolinux/kernel rw selinux=0 \
console=ttyS0,9600 console=tty0 auto_recover unattended
initrd (loop)/isolinux/initrd.cgz
}
set default=\"ReaR-recover\" >> /boot/grub2/grub.cfg"
```

loopback loop - command to mount ISO file, and we can use *loop* as a reference to the mounted root files system.

(*hd0,msdos3*) - the disk (*hd0*) and partition scheme (*msdos3*) where is placed root files system. It follows the path to the ISO file we want to mount. The names “*hd0*” and “*msdos3*” may be different for different platforms. I set it according to my VM configuration. For more information it is possible to see the configuration file for GRUB: */boot/grub2/grub.cfg* and files in the following directory */etc/grub.d/*. Also, it is possible to get in GRUB, where the GRUB is loaded type *c*. It opens a GRUB console. By the command *ls* you can list disks, partition schemes, and stores of the disk. Example in Figure 4.9

```

GNU GRUB version 2.06

Minimal BASH-like line editing is supported. For the first word, TAB
lists possible command completions. Anywhere else TAB lists possible
device or file completions. ESC at any time exits.

grub> ls
(hd0) (hd0,msdos3) (hd0,msdos2) (hd0,msdos1)
grub> ls (hd0,msdos3)/
root/
grub> ls (hd0,msdos3)/root
dev/ run/ boot/ afs/ lib sbin srv/ media/ usr/ opt/ lib64 bin home/ etc/ var/ t
mp/ sys/ mnt/ proc/ root/
grub>

```

Figure 4.9. GRUB, list disks and partition scheme

The line following after it is information for GRUB that we use Linux kernel to boot and the path to the kernel; *rw* - enable write and read during running the system; disable *selinux*; *console* defines a terminal for output and port, or just virtual terminal (*console=tty0*); arguments *auto_recover* and *unattended* are passed to the kernel for ReaR that we want to use automatic mode without an interaction with interface.

initrd - driver for the kernel to create a temporary root file system during the boot process. ReaR place the driver into the ISO file. We assign the path to the ISO file.

set default="ReaR-recover" - set the entry “ReaR-recover” as default. Since we want automated testing, we need the entry to be started by default.

The rest of the script is the same as in multi-host testing. Before executing the test in the infrastructure, I verify the correctness on the local VM. I described how to create and run the tests on the local VM in detail. For now, I just copied the already installed and configured VM.

To execute on the local VM we need to create a plan for TMT. In my case, it looks like that:

```

summary:
  Sanity tests

framework: beakerlib

discover:
  how: fmf
  path: /var/tmp/rear-testing
  name: Sanity/make-backup-and-restore-iso

```

That corresponds to the local place of the test.

I ran the test locally, and it worked correctly. As mentioned above, we need to use `memdisk` to boot the ReaR kernel. In GRUB, we assigned kernel and `initrd` manually by setting the relative path to files `isolinux/initrd.cgz` and `isolinux/kernel`. `memdisk` allows to use ISO images like a burned CD. It is more close to real cases in practice.

To have `memdisk` it is necessary to install an additional package. Since I worked with Linux distributions from Red Hat, which were Linux Fedora, CentOS 8, and CentOS 9, the needed package is `syslinux-nonlinux`. Just add the package to the following variable:

```
ADDITIONAL_PACKAGES=("syslinux-extlinux" "syslinux-nonlinux")
```

Also, since I used GRUB, I needed to copy `memdisk` in `/boot` directory that `memdisk` would be available to the GRUB, and rewrite the `menuentry`:

```
REAR_ISO_OUTPUT="/var/lib/rear/output"
...
rlRun "cp /usr/share/syslinux/memdisk /boot/"
...
rlRun "echo 'menuentry \"ReaR-recover\" {
    linux16 (hd0,msdos1)/memdisk iso raw selinux=0\
    console=ttyS0,9600 console=tty0 auto_recover unattended
    initrd16 (hd0,msdos2)$REAR_ISO_OUTPUT/rear-fedora.iso
}'
set default=\"ReaR-recover\"' >> /boot/grub2/grub.cfg"
```

During the testing, I encountered another problem. `memdisk` is not developing today. Unfortunately, it does not support large files. ISO image file with backup might take up to 3GiB of disk space. And `memdisk` was not capable of booting from such a large file. I tried `memdisk` with different sizes of ISO images. And successful boot was for files that were less than 1GiB. One possible solution is to create a copy of the ISO image and remove the backup files from it. The backup files are not needed during the ReaR boot. They are required in the restore stage only. To modify ISO images exist, a program `xorriso`. We need to add the corresponding package:

```
ADDITIONAL_PACKAGES=("syslinux-extlinux" "syslinux-nonlinux" "xorriso")
```

And then afterm `rearm mkbackup -v` command make a copy of file and remove `backup` directory from it:

```
rlRun "xorriso -as mkisofs -r -V 'REAR-ISO' -J \
-J -joliet-long -cache-inodes -b isolinux/isolinux.bin\
-c isolinux/boot.cat -boot-load-size 4 \
-boot-info-table -no-emul-boot -eltorito-alt-boot \
-dev $REAR_ISO_OUTPUT/rear-fedora.iso\
-o $REAR_ISO_OUTPUT/small-rear.iso -- -rm_r backup"
```

The code above will create a new file `small-rear.iso` with the removed backup.

Also, executing the test on the local VM worked as we expected. To execute the same test in infrastructure is necessary to add plans with the link on the test. TMT supports test execution from the GitHub repository. In my case, the plan looked like this:

```
[anton@fedora rear]$ cat plans/sanity.fmf
summary:
```

```

Sanity tests
framework: beakerlib
discover:
  how: fmf
  url: https://github.com/antonvoznia/rear-testing
  ref: test-iso-to-del

```

test-iso-to-del - name of git branch that contains test.

Unfortunately, it was not working in Testing Farm as we expected. It looked like the machine was not booted. We can recognize it because TMT uses ssh protocol to execute commands on machines. I ran the test several times and always got the following message:

```
ssh: connect to host 192.168.1.23 port 22: Connection timed out
```

It may mean issues with the TCP connection (the TCP driver also might not have been started). If the TCP driver has already been loaded but ssh there are some problems with the ssh connection (for example, the sshd daemon is not started), we would see *Connection refused* message on the terminal. For example, let's try to stop the sshd daemon:

```

[anton@fedora ~]$ systemctl stop sshd
[anton@fedora ~]$ ssh root@127.0.0.1
ssh: connect to host 127.0.0.1 port 22: Connection refused

```

I wrote simple plan for TMT to collect more information about the machines used in Testing Farm:

```

execute:
  how: shell
  script:
    - df -h
    - lsblk
    - lsblk -f
    - lsblk -l
    - free -h
    - uname -a
    - ls -la /
    - ls -la /boot/
    - ls /boot/grub2/device.map
    - cat /boot/grub2/device.map
    - ls /sys/firmware/
    - cat /etc/grub2-efi.cfg
    - cat /etc/grub2.cfg

```

The output of the command *lsblk -l*:

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINTS
zram0	252:0	0	3.8G	0	disk	[SWAP]
nvme0n1	259:0	0	40G	0	disk	
nvme0n1p1	259:1	0	1M	0	part	
nvme0n1p2	259:2	0	1000M	0	part	/boot
nvme0n1p3	259:3	0	100M	0	part	/boot/efi
nvme0n1p4	259:4	0	4M	0	part	

```
nvme0n1p5 259:5    0 38.9G  0 part /home
/
Shared connection to 18.118.36.202 closed.
```

In this setup, images is used NVMe controllers. However, we used the *VDA* hard-coded driver. Also, there is a difference in partition schemes. On our local machine, we used *msdos* instead of it in the Testing Farm; They are using *GPT (GUID Partition Table)*. I rewrote the test to make it as much as possible independent of the architecture we use.

To get a disk that we can mount, I used the following command:

```
ROOT_DISK=$(df -hT | grep /$ | awk '{print $1}')
```

Using the variable *ROOT_DISK* in code won't depend on the type of disk the system operates.

On some machines, the relative path to the root is */*. On others, it would be */root/*. The same is true for the *boot* directory. To get the relative paths, I added variables:

```
ROOT_PATH=$(grub2-mkrelpath /)
BOOT_PATH=$(grub2-mkrelpath /boot)
```

In some cases, the */root* directory and */boot* are placed on different partitions. To simplify the boot, I retrieve the UUID of the partitions and set it to GRUB menu entry that it finds by itself:

```
BOOT_FS_UUID=$(grub2-probe --target=fs_uuid /boot)
ROOT_FS_UUID=$(grub2-probe --target=fs_uuid /)
```

And, to prevent hard-coded names of the machine (because the ISO file name depended on it), I stored the host-name in variable *HOST_NAME=\$(hostname -s)*.

To summarize and compile above, we need to modify scripts for the pre-recover stage:

```
PRE_RECOVERY_SCRIPT=(\ "mkdir /tmp/mnt;\ " \
\ "mount $ROOT_DISK /tmp/mnt/;\ " \
\ "modprobe brd rd_nr=1 rd_size=2097152;\ " \
\ "dd if=/tmp/mnt/$ROOT_PATH/var/lib/rear/output/rear-$HOST_NAME.iso\
of=/dev/ram0;\ " \
\ "umount /tmp/mnt/;\ ")
```

Then I rewrote the GRUB menu entry by that:

```
search --no-floppy --fs-uuid --set=bootfs $BOOT_FS_UUID
search --no-floppy --fs-uuid --set=rootfs $ROOT_FS_UUID
terminal_input serial
terminal_output serial
menuentry \"ReaR-recover\" {
linux16 (\$bootfs)$BOOT_PATH/memdisk iso raw selinux=0\
console=ttyS0,9600 console=tty0 auto_recover unattended
initrd16 (\$rootfs)$ROOT_PATH/$REAR_ISO_OUTPUT/rear-rescue-only.iso
}
```

At this development step, I had all prepared for execution in the Testing Farm. In ReaR repository in *.packit.yaml* files I used configuration:

```

- job: tests
  trigger: pull_request
  metadata:
  targets:
  - fedora-latest-stable
  - centos-stream-8-x86_64
  - centos-stream-9-x86_64

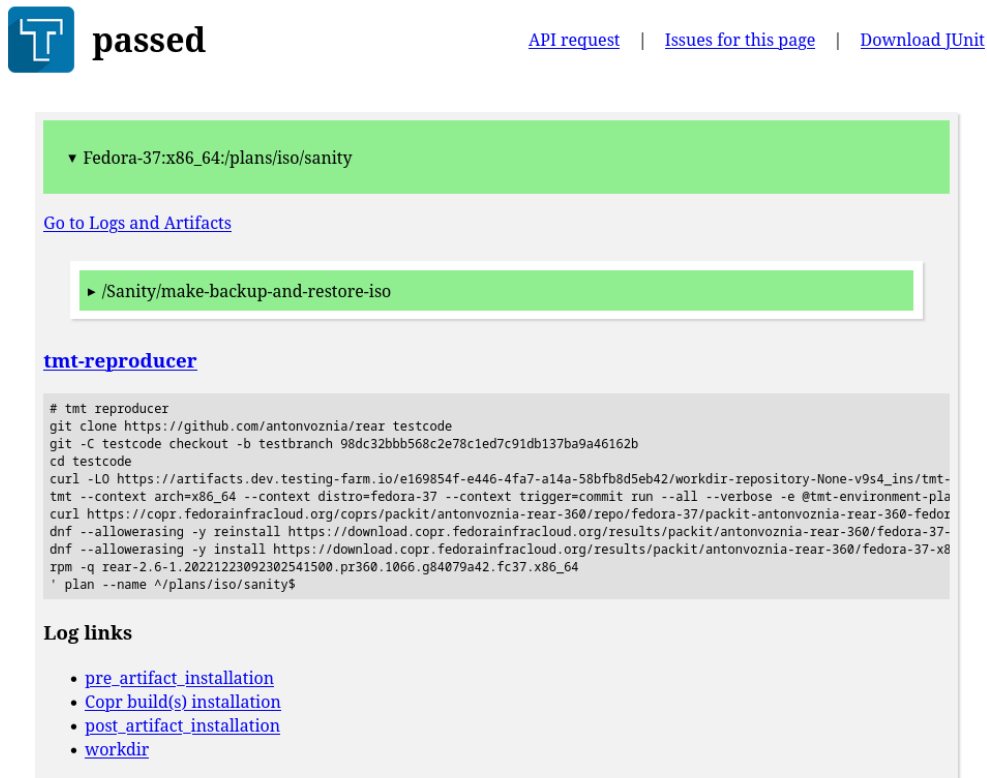
```

CentOS 8 and CentOS 9 are very stable distributions of Linux. They do not need to update critical parts of the kernel more often. That is the reason why I wanted to use such distributions. *fedora-latest-stable* it alias always corresponds to the latest stable versions of Fedora Linux. When I tested, it was Fedora 37.

The passed test for Fedora 37 you can see in the Figure 4.10

There are two links on branches in my GitHub for the testing set-up:

- The ReaR recovery tests for ISO backup ¹⁶
- Configured ReaR repository ¹⁷



passed [API request](#) | [Issues for this page](#) | [Download JUnit](#)

▼ Fedora-37:x86_64/plans/iso/sanity

[Go to Logs and Artifacts](#)

► /Sanity/make-backup-and-restore-iso

tmt-reproducer

```

# tmt reproducer
git clone https://github.com/antonvoznia/rear testcode
git -C testcode checkout -b testbranch 98dc32bbb568c2e78c1ed7c91db137ba9a46162b
cd testcode
curl -LO https://artifacts.dev.testing-farm.io/e169854f-e446-4fa7-a14a-58bf8d5eb42/workdir-repository-None-v9s4_ins/tmt-
tmt --context arch=x86_64 --context distro=fedora-37 --context trigger=commit run --all --verbose -e @tmt-environment-pla
curl https://copr.fedorainfracloud.org/coprs/packit/antonvoznia-rear-360/repo/fedora-37/packit-antonvoznia-rear-360-fedora-37-
dnf --allowerase -y reinstall https://download.copr.fedorainfracloud.org/results/packit/antonvoznia-rear-360/fedora-37-
rpm -q rear-2.6-1.20221223092302541500.pr360.1066.g84079a42.fc37.x86_64
plan --name ^/plans/iso/sanity$

```

Log links

- [pre artifact installation](#)
- [Copr build\(s\) installation](#)
- [post artifact installation](#)
- [workdir](#)

Figure 4.10. Passed ISO backup test

4.8 Debugging

During the work on the project I encountered significant problem. Before to run something in the Testing Farm infrastructure, I tested it on VM. That is easy to get log

¹⁶ <https://github.com/antonvoznia/rear-testing/tree/test-backup-iso>

¹⁷ <https://github.com/antonvoznia/rear/tree/test-backup-iso>

files from the local VM or just “pause” the execution and get actual files from it. The same way is not possible when we test something in cloud. As I wrote in the document, execute in Testing Farm simple script that collect me information about free memory, disk, partition table and map configuration.

Another problem I’ve run into was that in log files reported about failed test after making backup and `tmt-reboot` command. TMT uses `ssh` protocol for connecting to the test machine and executing commands via it. The `ssh` connections failed with status code 255:

```
14:15:40    err: ssh: connect to host\
3.142.79.215 port 22: Connection refused
14:15:40    Command returned '255'.
```

That means that after first reboot ReaR was loaded in memory and it loaded TCP driver. In case if TCP driver had not been loaded, I would get “Connection timeout...” message. To prove it I allowed in ReaR configuration file `/etc/rear/local.conf` `ssh` connection in rescue mode:

```
SSH_FILES=yes
PROGS+=( ps lsblk sleep cat lsattr tmt )
COPY_AS_IS+=( /usr/share/beakerlib \
${TMT_PLAN_DATA}/data/discover}/default-0\
/tests/Sanity/make-backup-and-restore-iso )
```

Because in rescue mode of ReaR is used Linux kernel and minimal count of binary programs, we need to add additional programs needed for debugging in the variable `PROGS`. During `rear mkbackup -v` ReaR adds the mentioned programs. The last one line with the variable `COPY_AS_IS` - it is files that will be accessible in rescue mode. Because I run the debug in the second stage of `rear-testing` (after first reboot) via TMT tool. So for the rescue mode, we need the library “beakerlib” that our test was able to run such commands like: `rlRun`, `rlLog`, etc. The directory `$TMT_PLAN_DATA/data-discover/default-0/tests/Sanity/make-backup-and-restore-iso` corresponds to the test we are running.

For the second stage I rewrote `runtest.sh`, that I described a modification for ISO backup in the section 4.7, in the following way:

```
elif [ "$TMT_REBOOT_COUNT" -eq 1 ]; then
    rlRun "{ rear -D recover;\
cat /var/log/rear/rear*.log; dmesg; } &"
    rlRun "sleep 500"
    rlRun "dmesg";
    rlRun "ps -e | grep -i rear";
    rlRun "lsblk -f";
    rlRun "cat /var/log/rear/rear*.log";
    rlRun "tmt-reboot"
...

```

The script above will be executed after the first reboot that is verifies via the variable `TMT_REBOOT_COUNT`. It “manually” executes a sequence of command in background: recovery, printing rear log, and printing kernel ring buffer. The log messages and outputs of `dmesg` will be in the TMT summary log file in case if the recovery will be executed successfully. We do not know in advance if the recovery had been

done or not. That was a reason to run the sequence of commands in background mode. Then I executed `sleep 500` that allowed my script wait 500 seconds before the script continue to collect data and information. The value `500 seconds` was picked according to the experience with restoring time. It would be enough for recovery. Then we printed again `dmesg`, verify if the `rear` process is still running (in case if the process hanged out), prints a list of block devices, get the rear log (we may find out the place where `rear recovery` failed), and the last one - I forced the system to reboot.

Similar debugging allows me to find an issues in my ISO backup test. The file `/var/log/rear/rear*.log` had an error where prints that in the `PRE_RECOVERY_SCRIPT` scripts it can't find the file `/var/lib/rear/output/rear-fedora.iso`. On some images the relative path to the system's root is `/`, others have `/root/`. That explains why we are using the following variable in the section 4.7 `ROOT_PATH=$(grub2-mkrelpath /)`.

Another problem I was solving was local TMT test execution with image from the cloud. Testing Farm suggested a list of instructions to run it locally if I got a failed test. Example in Figure 4.11.



failed [Issues for this page](#) | [Download HTML](#)

Go to Logs and Artifacts

Fedora-37:x86_64/plans/iso/sanity

tmt-reproducer

```
# tmt reproducer
git clone https://github.com/antonovoznia/rear-testcode
git -C testcode checkout -b testbranch 986c2bb8568c2e78c1ed7c91db337ba9445162b
cd testcode
curl -LO https://artifacts.dev.testing-farm.io/107d7f8b-4ee5-4e26-b228-768d915e21d/workdir/repository-home-15e3c1b6/tmt-environment-plans-iso-sanity.yaml
tmt --context arch=x86_64 --context distro=fedora-37 --context trigger=commit run --all --verbose -e @tmt-environment-plans-iso-sanity.yaml provision --how virtual --image Fedora-37 prepare --how shell --script '
curl https://copr.fedorainfracloud.org/coprs/packit/antonovoznia-rear-147/repo/fedora-37/packit-antonovoznia-rear-147-fedora-37.repo --entry 5 --output /etc/yum/repos/dcopr_build-antonovoznia-rear-147-1.repo
dnf --allowdowngrading -y reinstall https://download.copr.fedorainfracloud.org/results/packit/antonovoznia-rear-147/fedora-37-x86_64/98880266-rear/rear-2.6-1.20221203143421993221.pr147.1066.g977f406.fc37.x86_64.rpm || true
dnf --allowdowngrading -y install https://download.copr.fedorainfracloud.org/results/packit/antonovoznia-rear-147/fedora-37-x86_64/85880266-rear/rear-2.6-1.20221203143421993221.pr147.1066.g977f406.fc37.x86_64.rpm
rpm -q rear-2.6-1.20221203143421993221.pr147.1066.g977f406.fc37.x86_64
plan --name */plans/iso/sanity'
```

Log links

- [pre_artifact installation](#)
- [Copr build\(s\) installation](#)
- [post_artifact installation](#)
- [workdir](#)

Figure 4.11. Failed test, instructions

Chapter 5

File name validation

In this elementary and small chapter, I showed a test for validating the correctness of the names of source files of ReaR. Though ReaR is a set of bash scripts, it has a *Makefile*. It is possible to use different stages: build the project, install, clean, and validate sources. Validate stage does a few checks:

- Existence of configuration files exist in the directories *etc/* and *usr/share/rear/conf/*.
- Existence of “binary” executable rear file.
- Correctness of files name for ReaR code. All files (except sub-directories lib, skel, and conf) should end with 3 digits / 3-tuple. For example, the file *usr/share/rear/layout/recreate/default/220_verify_layout.sh*. If there were an incorrect file name, the “make” would fail. Let’s make a copy of the file mentioned above and remove from the name 1 digit (it will be 2-tuple at the start of the file), then execute *make validate*.

```
[anton@fedora rear]$ cp usr/share/rear/layout/recreate\
/default/220_verify_layout.sh usr/share/rear/layout\
/recreate/default/22_verify_layout.sh
[anton@fedora rear]$ make validate
...
ERROR: script usr/share/rear/layout/recreate/\
default/22_verify_layout.sh must start with 3 digits
make: *** [Makefile:95: validate] Error 1
```

As we can see above, we got the correct error message.

I set up and enabled such checking for RPM packages. Since the RPM package building is an automated system, creating a file with the “rules” for building the program is necessary. For RPM packages that are file with *.spec* format. For building, I used Packit. It triggers and uses the *.spec* file to create a new RPM package. Firstly I added *make* in *rear.spec* requirements. It allows a build system to use the *make* command. The second commit contained a change that adds the following code into *rear.spec*:

```
%check
%{__make} validate
```

It creates a new *check* stage for the building and uses macro *%make* to run *make validate*.

To test it, I create my own repository 2 PRs. One of them was with an incorrect file name. On GitHub, it is imaged as an error in section *Checks*. Example in Figure 5.1

The example with correct file names you can see in the Figure 5.2

The suggested changes have been integrated into the source codes.

5. File name validation

The screenshot shows a failed check for 'incorrect filename' with ID '6cf7722'. The check is associated with 'Packit-as-a-Service' and the job 'rpm-build:fedora-rawhide-x86_64', which failed on Jun 4, 2022. The message states 'RPMs failed to be built.' A table provides a link to the dashboard for results.

Name/Job	URL
Dashboard	https://dashboard.packit.dev/results/copr-builds/362146

Figure 5.1. make validate failed

The screenshot shows a successful check for 'correctness of make validate' with ID 'c6e43b5'. The check is associated with 'Packit-as-a-Service' and the job 'rpm-build:fedora-rawhide-x86_64', which succeeded on Jun 4, 2022. The message states 'RPMs were built successfully.' A table provides a link to the dashboard for results.

Name/Job	URL
Dashboard	https://dashboard.packit.dev/results/copr-builds/362145

Figure 5.2. make validate passed

Chapter 6

Static code analysis

In this chapter, I described the possible solution for static program analysis. Static code analysis - it is code analysis of a program without execution of the program. It was not a part of the work to introduce a new static analyzer or write own. There I tested already existing solutions for it; I showed the advantages and disadvantages of them for ReaR. In the first section of this chapter, I described what ShellCheck is and why it is useful for the project. And then, I described “Differential ShellCheck” and tested it.

6.1 ShellCheck

“ShellCheck - A shell script static analysis tool.”¹ ShellCheck is an open source project for static analysis for different versions of shell. It is written in Haskell. The program is licensed under the GNU General Public License version 3. Such type of licence allows to use the program in others open source projects. Notice: ReaR uses the same licence “GNU General Public License v3.0.”

Most of the code ReaR is written in specific Shell-bash (GNU Bourne-Again SHell). That means we can use the ShellCheck to verify a newly introduce code.

ShellCheck supports two very important features:

- We can specify a Shell we want to use. “Specify Bourne shell dialect. Valid values are sh, bash, dash and ksh. The default is to deduce the shell from the file’s shell directive, shebang, or .bash/.bats/.dash/.ksh extension, in that order. sh refers to POSIX sh (not the system’s), and will warn of portability issues.”
- Severity. “Specify minimum severity of errors to consider. Valid values in order of severity are error, warning, info and style. The default is style.”

ReaR uses BASH as a default language. So, we can specify the needed configuration that ShellCheck prints errors according to the standard.

Let’s try to install ShellCheck on Fedora 37 and try to execute it on a source file of ReaR:

```
# dnf install ShellCheck.x86_64 -y
# # shellcheck --version
ShellCheck - shell script analysis tool
version: 0.7.2
license: GNU General Public License, version 3
website: https://www.shellcheck.net

# git clone https://github.com/rear/rear.git
# cd rear
# ls usr/share/rear/init/default/030_update_recovery_system.sh
```

¹ <https://github.com/koalaman/shellcheck>

```
usr/share/rear/init/default/030_update_recovery_system.sh

# shellcheck usr/share/rear/init/default/030_update_recovery_system.sh
```

The output is too large, and I attached it as Figure 6.1.

```
[root@fedora rear]# shellcheck usr/share/rear/init/default/030_update_recovery_system.sh

In usr/share/rear/init/default/030_update_recovery_system.sh line 1:
^-- SC2148: Tips depend on target shell and yours is unknown. Add a shebang or a 'shell' directive.

In usr/share/rear/init/default/030_update_recovery_system.sh line 33:
local update_archive_filename="recovery-update.tar.gz"
^----^ SC2168: 'local' is only valid in functions.

In usr/share/rear/init/default/030_update_recovery_system.sh line 38:
local http_response_code=$( curl $verbose -f -s -S -w "%{http_code}" -o /$update_archive_filename $RECOVERY_UPDATE_UR
L )
^----^ SC2168: 'local' is only valid in functions.
      ^-----^ SC2155: Declare and assign separately to avoid masking return values.
                ^-----^ SC2154: verbose is referenced but not assigned.
                ^-----^ SC2086: Double quote to prevent globbing and word splitting.
^ SC2086: Double quote to prevent globbing and word splitting.

Did you mean:
local http_response_code=$( curl "$verbose" -f -s -S -w "%{http_code}" -o /$update_archive_filename "$RECOVERY_UPDATE
_URL" )

In usr/share/rear/init/default/030_update_recovery_system.sh line 45:
tar $verbose -xf $update_archive_filename || Error "Updating recovery system via 'tar -xf /$update_archive_filename'
failed."
      ^-----^ SC2086: Double quote to prevent globbing and word splitting.

Did you mean:
tar "$verbose" -xf $update_archive_filename || Error "Updating recovery system via 'tar -xf /$update_archive_filename
' failed."

In usr/share/rear/init/default/030_update_recovery_system.sh line 46:
popd
^--^ SC2164: Use 'popd ... || exit' or 'popd ... || return' in case popd fails.

Did you mean:
popd || exit

For more information:
https://www.shellcheck.net/wiki/SC2148 -- Tips depend on target shell and y...
https://www.shellcheck.net/wiki/SC2168 -- 'local' is only valid in functions.
https://www.shellcheck.net/wiki/SC2154 -- verbose is referenced but not ass...
```

Figure 6.1. ShellCheck example

From the Figure 6.1, you can see some errors the ShellCheck has found. It writes a code of the error and place.

It is possible to set standard shell:

```
shellcheck --shell=bash\
usr/share/rear/init/default/030_update_recovery_system.sh
```

And, as I wrote above, ShellCheck supports different severities. For example, we can execute it with the lowest rank of errors:

```
shellcheck --severity=error\
usr/share/rear/init/default/030_update_recovery_system.sh
```

6.2 Criteria for static analyzer

I have discussed the problematic of static analysis with developers of ReaR. They had few main criteria that are based on own experience with ShellCheck.

- To notify about introduced problems only. As we saw in some examples with an output of ShellCheck, it prints many errors and warnings. It is irrational to fix all of them. Some of the code has existed for a long time. Fixing such issues may cause the introduction of new errors. Or, for example, there is a pull request with 35 modified files.² It would be challenging to go through all files and verify if the errors were added or existed before.
- Minimize the number of false positive errors. It is a very subjective term “false positive”. But in some cases, the static analyzer may incorrectly mark it as problematic code. But it may be specific to ReaR code style. There are some examples. If we run the `shellcheck` command for file in ReaR code `usr/share/rear/init/default/030_update_recovery_system.sh` it prints errors like “SC2168: 'local' is only valid in functions”. In the file is used variable that is marked as “local” but is used outside of a function. But in ReaR code are files that are used inside other files and in functions. That means the local variable will be used in the scope of any function. As I noticed before, ReaR uses by default `BASH`. And, it does not declare by `#!/bin/bash` type of shell in all scripts. Since the main script is `usr/bin/rear` there is a declared type `BASH`. Other scripts are imported via `source` into it. There is no need to declare the type of shell. But ShellCheck cannot recognize and prints the following error “SC2148: Tips depend on the target shell, and yours is unknown. Add a shebang or a 'shell' directive.”
- One of the requirements was “nice to have UI feature”.³ Any UI interface may simplify reviewing code.

ReaR code functions are used in the `usr/sbin/rear/lib/` directory only. So, it is necessary to check for local variables only in the directory. And we know that `BASH` is a standard type of shell for ReaR. To summarize above, we need exceptions for the rules. ShellCheck supports exceptions. If we want to disable any checking, we can place it in the `.shellcheckrc` file. For example, the following code disables local variable checking and set `BASH` by default:

```
disable=SC2168
shell=bash
```

So, we can place the file in the root directory of the ReaR code. It allows for avoiding unnecessary warnings about incorrect usage of a local variable and lack of “shell” directive. So, to enable the rules for `lib` directory, we need to place another file:

```
enable=SC2168
shell=bash
```

Notice that we need again set `BASH` by default because the `.shellcheckrc` rewrite the previous configuration of `.shellcheckrc`.

² <https://github.com/rear/rear/pull/2625>

³ <https://github.com/rear/rear/issues/1040>

6.3 Differential ShellCheck

Differential Shellcheck is an open-source project developed by Jan Macku.⁴ That is a GitHub Action that uses ShellCheck for shell script checking and uses *csdiff* utility to match introduced errors only. “*csdiff* tool for comparing code scan defect lists in order to find out added or fixed defects, and the *csgrep* utility for filtering defect lists using various filtering predicates.”⁵

Differential ShellCheck uses very simple principles. It is possible to configure Differential ShellCheck to trigger every pull request made in a GitHub repository. It gets the files that had been changed and executes ShellCheck on them according to severity before the changes applied and after. The output of the 2-nd ShellCheck’s executions is stored in different files. At this time, the files have all errors from the ShellCheck warnings. Then it uses *csdiff* command to find differences and print them.

Jan Macku suggested such a solution.⁶

To add it in the ReaR repository is needed to create a workflow for GitHub in the directory *rear/.github/workflows/differential-shellcheck.yml*:

```
name: Differential ShellCheck
on: [pull_request]
permissions:
  contents: read
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Repository checkout
        uses: actions/checkout@1f9a0c22da41e6bfa534300ef656657ea2c6707
        with:
          fetch-depth: 0
      - name: Differential ShellCheck
        uses: redhat-plumbers-in-action/differential\
        -shellcheck@574cfd79f7317593a0a361cf50fec62d744b3c8e
        with:
          severity: error
          token: ${ secrets.GITHUB_TOKEN }
```

Now we need to create a PR against the branch where we have configured the Differential ShellCheck. For example, I attended the URL for such PR:⁷ We have three options to check the output.

- (i) To check a conversation with the PR. Differential ShellCheck prints it to the comments.
- (ii) In the menu tab “Checks” we can open the results of execution. Example 6.2
- (iii) In menu tab “Files changed”. Differential ShellCheck marks the code with errors and prints how it is possible to solve. 6.3

⁴ <https://github.com/redhat-plumbers-in-action/differential-shellcheck>

⁵ <https://github.com/csutils/csdiff>

⁶ <https://github.com/rear/rear/pull/2847>

⁷ <https://github.com/antonvoznia/rear/pull/353>

Then ShellCheck will print the following messages: “Couldn’t find ‘fi’ for this ‘if’. Couldn’t parse this if expression. Fix to allow more checks. ” And it fails in this step. It does not continue checking. If our file contains more errors followed by the incorrect condition, it won’t be found.

Differential ShellCheck cannot correctly recognize the line in which the code was wrong. I added in the file `usr/share/rear/init/default/030_update_recovery_system.sh` the following local variable usage (Figure 6.4).

```

--- 030_update_recovery_system.sh      2022-12-30 14:32:48.234744778 +0100
+++ 030_update_recovery_system.sh.1    2022-12-30 14:33:23.542975940 +0100
@@ -5,7 +5,8 @@

# Without a RECOVERY_UPDATE_URL there is nothing to do:
test "$RECOVERY_UPDATE_URL" || return 0

+local new_local_variable="A new local variable will be used in a function only."
+echo "$new_local_variable"
# With a RECOVERY_UPDATE_URL ensure 'curl' is actually there
# because that 'curl' was added to the default PROGS array
# (see https://github.com/rear/rear/issues/1156)
@@ -23,6 +24,9 @@
# via this special built-in ReaR functionality because in the normal system
# one can manually update ReaR as anything else.
test "$WORKFLOW" != "recover" && return
+local the_second_local_variable="The second local variable will be used in a function only. Close to another local\
+variable."
+echo "$the_second_local_variable"

# The actual work:

@@ -38,6 +42,8 @@
local http_response_code=$( curl $verbose -f -s -S -w "%{http_code}" -o /$update_archive_filename $RECOVERY_UPDATE_URL )
# Only HTTP response code 200 "OK" is what we want (cf. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes):
test "200" = "$http_response_code" || Error "curl '$RECOVERY_UPDATE_URL' failed with HTTP response code '$http_response_code'."
+local the_third_local_variable="The third local variable will be used in a function only."
+echo "$the_third_local_variable"

# Install the downloaded tar.gz at the root directory '/' of the recovery system-
# "tar --verbose" messages go to stdout so that they appear on the terminal where "rear recover" was started:
@@ -47,4 +53,3 @@

# Tell the user that recovery system update is done:
LogPrint "Updated recovery system."

```

Figure 6.4. Add new lines into the file

I ran the ShellCheck on the original file and then on the modified file to match the “local” keyword. Flag `-format=gcc` will force ShellCheck to print the output in gcc format, which is used by `csdiff`. Original | grep local:

```

# shellcheck --format=gcc 030_update_recovery_system.sh | grep local
030_update_recovery_system.sh:33:1: \
error: 'local' is only valid in functions. [SC2168]

030_update_recovery_system.sh:38:1: \
error: 'local' is only valid in functions. [SC2168]

```

Modified:

```

# shellcheck --format=gcc 030_update_recovery_system.sh | grep local
030_update_recovery_system.sh.1:8:1:\
error: 'local' is only valid in functions. [SC2168]

030_update_recovery_system.sh.1:27:1:\
error: 'local' is only valid in functions. [SC2168]

```

```
030_update_recovery_system.sh.1:37:1:\
error: 'local' is only valid in functions. [SC2168]

030_update_recovery_system.sh.1:42:1:\
error: 'local' is only valid in functions. [SC2168]

030_update_recovery_system.sh.1:45:1:\
error: 'local' is only valid in functions. [SC2168]
```

So, ShellCheck correctly found the local variable we have added. But in such an output format, finding lines that were modified and not added is impossible. By adding the new local variable, we shift the other code. The output of csdiff is the following:

```
Error: SHELLCHECK_WARNING:
./030_update_recovery_system.sh.1:8:1:\
error[SC2168]: 'local' is only valid in functions.

Error: SHELLCHECK_WARNING:
./030_update_recovery_system.sh.1:27:1:\
error[SC2168]: 'local' is only valid in functions.

Error: SHELLCHECK_WARNING:
./030_update_recovery_system.sh.1:37:1:\
error[SC2168]: 'local' is only valid in functions.

Error: SHELLCHECK_WARNING:
./030_update_recovery_system.sh.1:42:1:\
error[SC2168]: 'local' is only valid in functions.

Error: SHELLCHECK_WARNING:
./030_update_recovery_system.sh.1:45:1:\
error[SC2168]: 'local' is only valid in functions.
```

It just prints all lines again.

6.4 Script for automatic PRs

So, running the Differential ShellCheck on different use cases is necessary. And compare the behavior of the static analysis. I asked my thesis adviser to send me real examples where in the code of ReaR were fixed problematic spots code. I got a list of PRs from GitHub which were approved by the ReaR development team and merged with the master branch. Using real examples is better than manually adding errors because it demonstrates how Differential ShellCheck may act for the project. The idea of such testing is to run the Differential ShellCheck on the PRs. Differential ShellCheck uses GitHub workflow. That is why it requires creating two branches (1 enabled Differential ShellCheck and the second is with applying commits from the PR). I want to simulate a case when developers add the problematic code and verify how our setup is capable of finding it out. To simplify the testing on the list of PR, I wrote a script which is following steps:

- (i) get all commits from the specific PR

It is possible to get all commits that belong to a PR from the GitHub website and parse them. The link with commits would be in the following format

```
https://github.com/rear/rear/pull/pr_number/commits/
```

Where `pr_number` - is a number for a specific PR. I used `wget` command to download the web-page with commits from the PR, and `grep` them with string and regex

```
/rear/rear/pull/$pr_number/commits/[A-Za-z0-9]
```

All commits have their calculated hash with SHA1 checksum. The hash of length 160 bits is composed of characters and digits. That is the reason why I add such “[A-Za-z0-9]” regex at the end of the string. Now we have only an array of hashes of commits. The full link looks like

```
https://github.com/rear/rear/commit/[A-Za-z0-9].patch
```

If we add “.patch” at the end of the link, which corresponds to the GitHub link, it will satisfy the commit patch.

- (ii) Create a new branch before applying the commits in step 1

In this step, we need to create a new branch and allow Differential ShellCheck to check our code. In step 7. we are going to create a new PR with changes from step 1. against to branch with enabled Differential ShellCheck. The idea is to simulate errors fixing on real PRs that have done it. I made it by the following line:

```
git checkout --force -b \  
"reverse-severity-$severity-$pr_number-no-fix" ${commits[-1]}
```

severity - variable with 4 possible values: error, warning, info, style.

pr_number - variable I used to mark and distinguish certain PR from the others.

commits[-1] - a value corresponds to last commit in the array before changes applies.

- (iii) enable usage of Differential shellcheck for ReaR

There I created a small git patch enabling Differential ShellCheck. Having the patch allows me to modify it and switch the value of the severity parameter. In the following steps, I can apply the patch.

- (iv) Create a new branch with already enabled Differential ShellCheck

For the branch, we are going to apply to commit by commit from step 1. In this step I create a branch from the branch with enabled Differential ShellCheck:

```
git checkout --force -b "reverse-severity-$severity-$pr_number-fix"
```

- (v) apply the commits from step 1.

We are not interested in committing messages or considering every commit separately. Because the static analysis triggers the whole PR. The static analyzer doesn’t distinguish which commit an error fixed or added.

- (vi) push the two branches (from steps 2. and 4.) into the GitHub repo

Before creating a new PR, we need to have the two branches we want to compare on GitHub.

- (vii) generate a PR with a branch from step 4. against a branch from step 2.

We can do it manually from GitHub via the web interface, or it is possible to use `gh` tool. “GitHub CLI, or `gh`, is a command-line interface to GitHub for use in your terminal or your scripts.”⁸

⁸ <https://cli.github.com/manual/>

gh helps us to make PR from our script automatically. I used the following script to make a PR:

```
gh pr create -R antonvoznia/rear\  
-B "severity-$severity-$pr_number-no-fix"\  
--title "$full_comment"\  
--body "$full_comment"
```

Where: *gh pr create* - create a PR.

-R *antonvoznia/rear* - specify the repository we want to make PR into.

-B *"severity-\$severity-\$pr_number-no-fix"* - specify a branch we want to make PR against. It is a branch with activated Differential ShellCheck but with no the changes.

-title *"\$full_comment"* - a title of PR. *full_comment* I generated in the script.

-body *"\$full_comment"* - a body text with the string value *full_comment*.

6.5 Script modification

While working with the script I described above, it found fixed problems in most cases, and only a few errors were added. We cannot exactly mark them as an error because it depends on severity and false-positive mistakes, and what exactly for ReaR style code is error is not defined. To test on, let's say "real" example of error introducing, we can reverse patches and apply them. The patches fix some errors which, whether Differential ShellCheck found or not. If we apply the patch in the reverse way, that means we add the errors again, Differential ShellCheck mark them. It would be the same errors that were recognized as fixed. But for our assurance, we need to test it. Git supports reverse-applying patches.

```
git apply -R patch-name.patch
```

Where argument *-R* turns on the option to reverse patch file "patch-name.patch" and apply it. In my script, I loaded from GitHub web-site patches by curl command. On the output of the curl I used *pip |* of bash and applied it by the git command

```
curl "$GET_COMMIT_URL/${commits[$counter]}.patch" \  
curl "$GET_COMMIT_URL/${commits[$counter]}.patch" \  
| git apply -R -v --index
```

Logically it doesn't make sense to apply a patch and reproduce the error. But in this way, we can test Differential ShellCheck on real examples and on actual code written by ReaR developers.

The script may not work correctly if some interface of web part of GitHub changes. But it was enough to test Differential ShellCheck with different severities, PRs, and commits. It is not a part of the work to write good enough script which will be supported for years.

The final version of the script is available by the link <https://github.com/antonvoznia/diff-shellcheck-test-script>

Chapter 7

Conclusion

This project introduced and described several possible solutions for automatic testing of the open-source program ReaR.

The ReaR project was described in chapter 2. There was demonstrated example of how it is possible to use the program. And based on this, I wrote requirements for automatic testing. The two main approaches were introduced in the thesis: testing of the basic functionality of ReaR and static code analysis.

Chapter 3 describes different infrastructures that I considered. I explained there three infrastructures I tested and wrote both advantages and disadvantages of each other. Referring to requirements for automatic ReaR testing, I chose Packit as a building platform in conjunction with Testing Farm infrastructure. These two infrastructures, in conjunction, allow us to build a project with changes suggested in a PR on GitHub and test on the built package. Such an approach makes it possible to find on-time regressions. GitHub Actions were enough for static analysis because of the lack of dependencies on specific architectures and environments (like OS, CPU architecture, type of filesystem, possible reboot system, etc.).

I introduced TMT to drive the testing process. I showed how to configure TMT and VMs for local testing, and then I demonstrated an example of test execution with the chosen infrastructures (Packit and Testing Farm). Although it is not possible to use a multi-host approach to make a backup over NFS, I showed a solution that may be integrated in the near future. Then I extended the tests to make executing the chosen infrastructures possible. I described the problems I encountered and how it is possible to debug in cloud systems in case of solid access restrictions. But a very important part was with backup in the ISO image because it introduces a non-trivial solution that may already be used in this project.

Chapter 5 contains short descriptions of validating source and configuration files' correctness. The file validating has already been integrated into the ReaR project.

In the last chapter of the practice part, I described existing solutions for static code analysis. I showed their disadvantages, and I introduced solutions for some of the issues. Because of the complexity of the static code analysis sphere and the subjective understanding of the problem, there may not be a quickly found solution that will satisfy all requirements and imagines of developers. I wrote a script that allows simplified testing of the Differential ShellCheck on real examples that fixed some code problems. The script is useful and may be used in further experiments with static code analysis.



References

- [1] David Farley Jez Humble. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))*. 2010.
- [2] *System BIOS for IBM PC/XT/AT computers and compatibles: the complete guide to ROM-based system software*. Reading, Mass: Addison-Wesley Pub. Co, 1989. ISBN 9780201518061.
- [3] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. *Design and Implementation of the Sun Network Filesystem*. 1985.
- [4] Jim Mauro Richard McDougall. *Solaris Internals. Solaris 10 and OpenSolaris kernel architecture..* 2006.
- [5] Hal Stern. *Managing NFS and NIS (Nutshell Handbooks)*. 2001.
- [6] Brendon Perry Dave Taylor. *Wicked Cool Shell Scripts, 2nd Edition: 101 Scripts for Linux, OS X, and UNIX Systems*. 2016.
- [7] William E. Shotts Jr. *The Linux Command Line: A Complete Introduction* . 2012.
- [8] Prasad Mukhedkar Vedran Dakic, Humble Devassy Chirammal. *Mastering KVM Virtualization: Design expert data center virtualization solutions with the power of Linux KVM, 2nd Edition*. 2020.
- [9] Curtis Gedak. *Manage Partitions with GParted How-to*. 2012.
- [10] Pranoday Pramod Dingare. *CI/CD Pipeline Using Jenkins Unleashed: Solutions While Setting Up CI/CD Processes*. 2022.