BINARY PACKAGING FOR THE ROBOT CONSTRUCTION KIT

*Thomas M. Roehr[1], Pierre Willenbrock[1]

[1]DFKI GmbH Robotics Innovation Center, Germany, E-mail: {thomas.roehr,pierre.willenbrock}@dfki.de

ABSTRACT

This paper introduces a binary software packing mechanism for the Robot Construction Kit (Rock). The use of Rock with a variety of single-robotic systems and the use in multi-robot scenarios has triggered the development of a repeatable and manageable installation and upgrading process. The mechanism is based on the widespread Debian packaging format and takes advantage of the available tooling. The high degree of standardization within the Rock framework, and the application of a meta-build tool have contributed to a build process which allows to automatically package hundreds of associated packages. A transparent integration into the meta-build tool allows to maintain the typical development workflow, and enables users to switch between custom source package variants and binary software packages. This paper will detail the architecture of the packaging approach, illustrate features and show the validation, and lessons learned from daily use.

1 INTRODUCTION

This paper introduces a binary software packaging mechanism for the Robot Construction Kit (Rock) [3]. Rock is of particular interest for the space community since it follows a model-based software development approach, and is used on a number of space related robotic systems such as SherpaTT [9], Coyote III [9], CREX [8] and ESA's ExoMars Test Rover (ExoTeR) [4]. The use of Rock with a variety of single-robotic systems and the use in multi-robot scenarios such as TransTerrA [10], RIMRES [8] and VIPE [7] has increased the need and triggered the development of a repeatable and manageable installation and upgrading process. The support of a component database and streamlined, user-friendly development workflow by the project D-Rock [2] added additional requirements for the software packaging solution. The motivation for the development of a binary packaging mechanism originates from the set of application scenarios and user demands which have been identified especially in the context of operating heterogeneous, multi-robot systems for space exploration [7, 10]. A large amount of software packages can be shared in a multi-robot system, but in practice this does not necessarily lead to a homogeneous software basis: multiple architectures and distributions have to be supported and in the case of Rock the flexibility of the meta-build system autoproj comes with the risk of an equally heterogeneous landscape of build configurations in a multi-robot system, if not managed with strict user policies.

The typical development process for robotics systems often relies on a source-based build, i.e. requires to compile a number of source packages for individual robotic systems and for all systems used for development. This approach wastes time, stretches the patience of developers as well as maintainers, and remains prone to errors. While the time for performing a full source-based build can be reduced for Linux-based systems with tools such as *ccache*[1] and *icecc*[2], a better option is offered by a binary packaging system. For Linux-based systems the Debian distribution has set a widespread standard and due to the distribution's stability serves as basis for many other Linux distributions including the popular distribution Ubuntu. The Debian ecosystem and its packaging format allows for a robust, repeatable and standardized process for software installation as well as upgrading.

Debian packages are also available for the popular Robot Operating System (ROS) [6], in contrast to Rock [3], which has long lacked the provision of Debian packages. The binary packages that can be generated with the approach described in this paper close this gap for Rock. But even more, the general approach introduces a local packaging system and hierarchical releases, which facilitate the creation and distribution of customized releases. The packaging solution is based on Rock's meta-build tool *autoproj* [5] and, while the approach has been implemented for Rock, the concept is generic enough to be applicable for other frameworks outside the scope of Rock.

The paper will describe the related work in Section 2. The general architecture will be described in Section 3 followed by a description of the practical application in Section 4. The general lessons learned will be collected in Section 5, while the paper concludes with a brief discussion and an outlook in Section 6.

2 RELATED WORK

A number of different packaging mechanisms exists, notably used by various Linux distributions. These distri-

[1]A fast C/C++ compiler cache - `https://ccache.samba.org`

[2]Distributed compiler - `https://github.com/icecc`

butions offer software in the form of packages, which can be installed into a system, and packages are distributed in binary form, i.e compiled and ready for installation, or in source form, requiring compilation before installation. Packages come with some meta data, e.g., to describe dependencies, allowing the packaging mechanism to account for dependencies for building and installing packages.

All packages need to be compiled at some point; either at the user's or at the distributor's side, and all binary packages will depend upon a source package format. Source based packages often include build recipes which rely on known build systems, such as *CMake* or *autotools*, and only provide additional configuration mechanisms, e.g., to customize installation directories and apply patches. The build process for the standard package format is also standardized, and allows to ease maintenance, create packages in a reproducible way, and build packages automatically and unattended.

This standardized process is supported by package management toolchain on the maintainer and user site, to organize the build, distribution and installation process, e.g., the DebianLinux Distribution uses *dpkg* and *apt* to manage packages at the user side, and the Common Debian Build Sytem (CDBS) [1] for package creators. A Debian package build description can handle arbitrary tasks for automation, so there is no limitation on the build system. While Debian packages respositories can be managed with *reprepro*[3], the *apt* system handles the task of downloading packages, and their dependencies as described in source and binary packages.

The *rpm*[4] related packaging toolchain represents another popular approach which is also used by major Linux distributions such as Red Hat and openSUSE. In general, the processes used for managing and distributing *rpm*-based package are similar to the ones used by Debian.

In contrast to other distributions, Gentoo[5] distributes only source packages. It uses the *emerge* utility to handle both build and installation for requested packages, and their dependencies at the user's side. Meta data and source are separated, the source may be downloaded from either one of Gentoo caches or directly from their original web site. As source, an archive holding the source files can be used as well as direct download of source files from version control systems like *subversion* or *git*.

While all previously mentioned package install into system directories, package managers such as *gem*, *rosinstall*, and *autoproj* exist for managing package installations in local (user) directories. The package manager *gem* handles installing and building of packages for Ruby using the package format "Ruby gem" or just "gem". All gems can be retrieved from a central respository, and the package manager *gem* handles both installation to system and user directories. The tool *gem2deb* can be used to convert gems to Debian source packages, although these packages often need some manual changes to make them build.

ROS [6] uses *rosinstall*[6] to maintain workspaces in user controlled directories. Workspaces contain a number of packages and may depend on packages from other workspaces, or from the base system. While ROS packages are typcially installed from source, ROS also provides Debian packages and Gentoo build instructions; Debian and Gentoo packages are generally produced and maintained manually.

Autoproj is the package manager for Rock [3], and like *rosinstall*, it manages an installation in user controlled directories. It downloads source packages, either using *git*, *subversion*, or an archive from a webserver, and building relies on the use of known buildsystems , e.g., *CMake*, *autootools* (Make), or *rake* for Ruby packages. *autoproj* accounts for source packages' dependencies, and downloads, build and installs them if required. Rock package dependencies can either be operating system dependencies, i.e. Debian's binary packages, Ruby gems, or other Rock packages. To download, build or install custom packages, users can dynamically extend *autoproj*. Package sets in *autoproj* define how a set of packages can be retrieved and built, i.e. specify the location as repository URL, the type of version control system , e.g., git or svn, and the release, tag, branch or particular commit which defines the version to use for a package. In addition, packages are types according to the used build systems; if no build system shall be used packages will simply be retrieved. While autoproj allows to compile packages, it cannot create binary packages.

3 BINARY PACKAGING ARCHITECTURE

Popular Linux distributions are a good source for identifying software practices which are suitable for large scale development. In particular the Debian distribution is the foundation for many other Linux distributions such as the popular Ubuntu, Kali Linux and Purism PureOS to name only a few. Debian has developed and standardized the packaging and upgrading process, and comes with a number of tools for this process. Since the target systems of the Rock framework are mostly Debian-based systems, a binary packaging architecture has been developed which builds upon the expertise and tooling of the Debian community.

[3] Reprepro `https://mirrorer.alioth.debian.org`

[4] RPM Package Manager: `http://rpm.org`

[5] Gentoo Homepage: `https://gentoo.org/`

[6] rosinstall documentation: `http://docs.ros.org/independent/api/rosinstall/html/`

The typical Rock workflow relies heavily on the meta-build tool *autoproj* [5], which is capable of creating and maintaining Rock installation with hundreds of source packages. To achieve scalability, *autoproj* provides an interface and thus standardization layer to manage heterogeneous package distributions in a very flexible way. The binary packaging approach reuses the package database which *autoproj* maintains - especially the dependency tree. The following subsections describe the elements of the architecture of a binary packaging system implemented on top of *autoproj* in order to achieve the intended workflow standardization.

The general architecture has to account for two major roles of users: (a) the Rock maintainer or package creator, and (b) the Rock user and thus main user of the created packages. The architecture description will first focus on the needs of a Rock maintainer, and subsequently on the needs of end users of packages.

The Debian package generation workflow consists of the following general steps: (a) bootstrap a source package and apply custom patches, (b) create source tarball (**.orig.tar.gz*), (c) create Debian build instructions (**.debian.tar.xz*) and package specification (**.dsc*), (d) use build instructions to compile binary package, and (e) register and host compiled binary packages in a repository

The first three steps are covered by our tool *deb_package*, whereas the building and registering of packages will be covered by our tool *deb_local*. To trigger the build of a Debian source package, a source tar archive (**.orig.tar.gz*) and the control files need to be provided. The control files consist of the specification file (**.dsc*) and the build instructions (**.debian.tar.xz*); among other meta information they contain the package name, version, dependency information, and the changelog. The Debian community has automated the build process based on these three artifacts using the *pbuilder* and *cowbuilder* utilities, so that the most important step is the complete and correct generation of the control files. Figure 1 shows the core components for the tool *deb_package*, which performs the generation of the control files and the correct packaging of the original source files.

3.1 Prerequisites

In order to setup a general architecture for building Rock packages a minimal level of standardization is required. Autoproj already provides a significant level of standardization, and thereby allows to maintain a local installation with hundreds of packages. Nevertheless, to use this capability for building Debian packages, an adaption and harmonization layer is required, since Debian packaging relies on its own strict set of policies.

Naming schema Packages in autoproj follow a path-based naming schema, using the slash as separator between namespaces, e.g., base/types will be a package *types* located in the folder *base*. Debian requires a canonization of packages name, so that the autoproj based name base/types will be translated in a dash-based naming schema: base-types. The addition of further distinguishing namespaces is required in order to account for the target framework, here: *rock* as default, a particular release name, here following the pattern *FLAVOUR-YY.MM* to be understood as the main package branch of Rock to be used, year and month, e.g., rock-master-18.01-base-types. Additionally, Debian requires to add a package version and a suffix for the target release name, e.g., 0.20171118-1~xenial.

Release & Installation target Binary packaging of a given set of Rock packages leads to a consistent, distribution-like set of packages that can be used for an installation. Each such set of packages will be called a Rock Debian package release or just release. Many Rock users use multiple workspaces for development and require support for multiple releases in parallel, so that each release uses a separate installation folder: */opt/rock/<release-name>*. The possibility for the parallel installation of releases is a fundamental requirement for the use of hierarchical releases, which will be detailed in Section 3.3.

Versioning & Timestamping Rock consists of heterogeneous collection of packages, where no general assumption can be made about the versioning schema of packages. Hence, the version is derived from the time of the last commit for all packages which are accessed through a supported version control system. For all packages which are imported as tar balls, the time of last modification is used. The extracted timestamp is enforced on the later generated source tarball (**.orig.tar.gz*), so that a repeated generation of a source tarball has equal checksums for the same version.

Dependency management Autoproj allows to compute recursively all dependencies for a package including other Rock packages, Ruby gems and Debian packages that are available for a specified (or typically the current host's automatically identified) operating system. Rock binary packages can be built for multiple architectures and distributions, but autoproj currently does not consider the architecture in its dependency definitions. Therefore, an availability check for operating system dependencies is essential in order to verify the correctness of dependency information. The existing tool *dcontrol*[7] allows to query this information for Debian distributions, but not for derived ones such as Ubuntu. Therefore, a mechanism has been introduced to query this information for Ubuntu based systems from the online collaboration platform Launchpad[8]. An additional challenge has been the set of Ruby

[7] dcontrol is part of the Debian devscripts packages `https://github.com/Debian/devscripts`

[8] Launchpad collaboration platform: `https://launchpad.net`

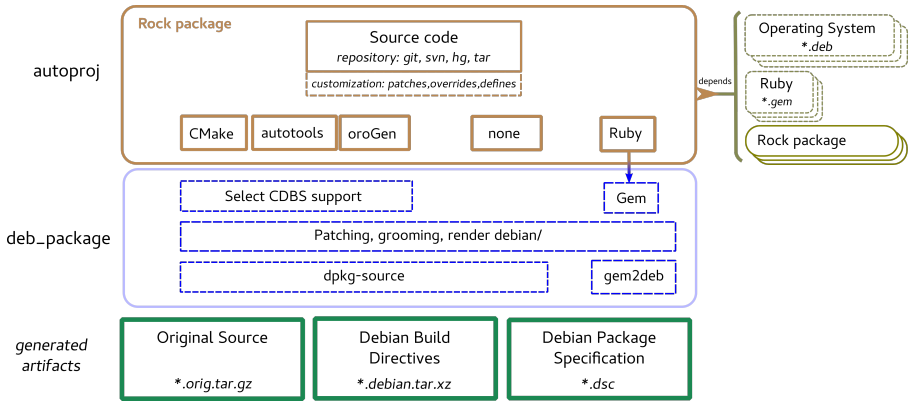

Figure 1. Core components of the tool `deb_package` to generate the required artifacts for a Debian package built.

gems used in Rock installations and identification of dependencies. The program *gem* allows to manage Ruby packages aka gems and also allows to query information about dependencies to other gems. The default assumption when building Rock binary packages is a full bootstrap including all required gems, so that dependency information can easily be queried from the local cache.

3.2 Preparing Debian source packages

Autoproj supports retrieving repositories from various version control systems and tar archives. It supports building using CMake, oroGen (which uses CMake), autotools and Ruby gems using Rake. As already mentioned, autoproj manages the dependencies between packages and, most of the time, packages provide information about the required dependencies through a manifest file. In addition, a user can inject additional information or packaging control directives into autoproj, e.g., for packages that are located outside the administrative realm of the user or maintainer.

Patching & overlay The tool `deb_package` uses autoproj to retrieve the package source, information about direct dependencies, and other meta information about the package. For most packages this information is sufficient to generate the Debian control files, since the CDBS [1] provides a parametrizable set of Makefile fragments which can be reused, according to the source packages build system.

In all cases, the binary packing system allows to apply patches to the source to correct inconsistencies or fix build instructions. These patches come in form of an overlay over the existing source packages. In practice, fixing existing build instructions is often required for Ruby packages, since these were either incomplete or too narrowly focused on a single architecture or distribution.

Ruby packages A significant portion of Rock packages are extra Ruby gems and Ruby packages the framework provides by itself. Although the CDBS provides support for other interpreter languages such as Python or Perl, it does not for Ruby. Therefore, a central element of this packaging architecture is dedicated to automate the binary package generation for Ruby packages. Unfortunately, Ruby packages do not follow a strict standardization, and require a special treatment in many cases. Still, the Debian workflow uses a tool called *gem2deb*[9] which converts gems which follow a particular standardized layout into the required Debian source artifacts. Therefore, the general workflow aims at converting any Ruby package into a *gem*, thereby enforcing a level of standardization. As a result, the packing process can easily also account for external gems which need to be packaged as well, e.g., when they are not provided by the package management system of the operating system. This means, that all external Ruby gems that are required by a package in the Rock framework automatically become part of the generated binary Rock package distribution. Still, the customization of the target installation folder, e.g., to the mentioned */opt/rock/<release-name>*, required an adaptation of the *gem2deb* tool to allow parametrization.

Since gems also come in a variety of versions, the packaging supports version pinning for the built Ruby gems. Typically however, if a gem is already available for a particular Linux distribution, this package is preferred over (re)packaging it; this policy will likely change in the future, since it can lead to conflicts when a release requires a more recent version of these gems.

Multiple architectures and distributions The compilation of Rock for the ARM architecture has been another reason to develop the binary packaging system. Although

[9] Debian Ruby packaging suite `https://packages.debian.org/sid/gem2deb`

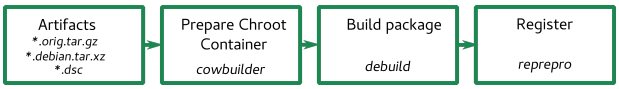

Figure 2. Workflow of `deb_local`

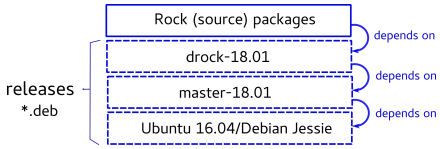

Figure 3. Illustration of the dependencies for a hierarchical release drock-18.01, which reuses the packages of the release master-18.01

a dedicated cross compilation workflow has been considered, numerous hindrances especially regarding Ruby packages have been identified. Therefore, the packaging assumes building directly on a system with the given architecture; other architectures can be built for using platform emulation or container-based building.

Generating the Debian source artifacts for various architectures and distributions is based on feeding a user-defined setting of the operating system into autoproj instead of using the auto-detected one. Controlling the setting allows to generate the Debian source artifacts for all target platforms with a single build system.

Special treatment is required for a subset of Ruby gems, e.g. concurrent-ruby, to remove any architecture specific prebuilt extensions.

3.3 Building Debian packages

Once the artifacts have been created few steps remain in order to build the final Debian package. An initial base image for an operating system and architecture combination is constructed using *pbuilder* for that purpose. Our tool *deb_local* uses a standard approach with *cowbuilder* and *pbuilder* to create the build environment: the combined use of these tools allows the repeated building of Debian packages within clean environments, i.e. each build copies a default base image and then populates it with the dependencies of each package. This approach allows to quickly identify missing build-dependencies.

Parallel build `deb_local` takes a number of autoproj package names or autoproj meta packages (a collection of autoproj packages) and generates Debian binary packages for them and all of their dependencies, as required. Since installing all required dependencies, and building binary packages can take a long time, the build jobs are run in parallel, if configured. Figure 2 illustrates the workflow of the package building using `deb_local`.

Hierarchical release structures The creation of binary package releases requires an initial investment in maintenance and standardization of packages. To avoid the unnecessary repetition of build and maintenance activity for binary packages that are already available in a particular binary Rock package release, hierarchical release structures have been enabled. While a release allows to label a consistent and compatible set of packages, the concept of hierarchical releases allows to relate releases by a parent child relationship and thus share software packages through inheritance. Each build can be configured, so that it depends upon another binary Rock package release. This effectively adds a custom package repository and before creating a new Debian source package, this package is searched for in the known ancestor releases. A new release might reuse an existing release, but still needs to override a subset of packages with customized version. For that purpose, a Rock maintainer who created a new release can provide a list of packages (or name patterns) which will be ignored when found in ancestor releases; all their reverse dependencies will be ignored as well.

3.4 Integration with Rock

In order to allow an existing installation of Rock to use the Debian packaging, a mechanism had to be found to transparently integrate the set of generated packages. Firstly, packages are hosted via the use of *reprepro* - these repositories can be seamlessly integrated into the apt-based package management. Secondly, the user is required to only add a particular package set to the manifest of the autoproj based installation. The package set contains the information about all available Debian packages as part of a list of so-called osdeps files, and by default the definition of binary packages will override existing source package definitions; one osdeps file exists per Rock Debian package release and per architecture.

An automated procedure has been embedded into the package set in order to: (a) detect the operating system and the architecture, (b) set necessary environment variables, (c) automatically add the required repositories to */etc/apt/sources.list.d/* based on the Rock release name, which has to be selected by the user, and (d) render a single osdeps file under consideration of blacklisted packages and multiple ancestor releases.

Blacklisting binary packages refers to disallowing the use existing binary packages for a workspace. Blacklisting enables a Rock user to follow the classical development workflow while at the same time allowing the use of Debian packages in a Rock installation, i.e. while Debian packages will be preferably used in any existing installation they can be overridden by local packages when blacklisted.

Overall, the Debian package integration has been designed to require minimal Rock user interaction. This ap-

proach avoids the compilation all packages, which are not actively developed by a Rock user and can thereby significantly speed up of the installation process as well as the subsequent builds.

4 EVALUATION

The development and implementation of the binary packaging architecture has been a continuous process, where numerous shortcoming and additional requirements have been identified and fixed in a iterative manner.

4.1 Rock maintainer workflow

The general maintainer workflow has been outlined in the main part of this paper, and the latest created binary Rock package release comes with approximately 310 packages, proving the capability of the current approach. Maintaining such large number of packages is only possible with a high degree of standardization, and the main issue for adding packages is the reproduction and interpretation of errors requiring extensive logging capabilities. Due to the large number of packages, the build process can take a long time, and in order to account for the priority of core package sets and to facilitate the maintenance on the level of package sets, these are build in order of their inter-dependency.

4.2 Rock user workflow

The project VIPE [7] serves as platform to verify the full set of features of the binary packaging for Rock users. The use of additional packages, e.g., such as the simulation, required additional verification of build procedures, and identified incorrect definitions in oroGen components. The application of Debian packages in the context of VIPE showed that a large number of Rock packages come with the implicit assumption that they are installed into the same installation directory. However, this assumption breaks as soon as a dependency exists only as Debian package. This requires fixing for both: (a) using the Debian packages in contexts of such packages, and (b) creating binary packages for this set of packages. Additionally, erroneous include directives are exposed, since no standard include path is assumed. The verification in the project has shown that the usage of the Debian packages can substantially increase the quality and robustness of the Rock software stack in general.

The requirement for blacklisting packages in order to use a custom version of a source Rock package instead of the binary version is an essential feature. The same holds for the generation of new binary package releases, since projects such as VIPE branch packages, but still want to exploit the benefit of the binary Rock packages. This shows, that the future development has to account for the fact that the role of a Rock user overlaps more and more with the one of a Rock maintainer in order to provide a custom, project-related binary package release. Hence, the addition and consideration of a blacklisting of packages for the generation of packages is currently under development.

4.3 Field Testing

A major testcase was given through the Field Trials of FT-UTAH [9], which required the in-field operation of a multi-robot team. Customization and parametrization of the software stack during the field trip was necessary in order to tackle newly identified requirements or bugs in the software stack. In order to minimize the necessary compilation of packages, three different binary Rock package releases were accounted for: master-16.07, master-16.08, and master-16.09. The use of these three releases exposed one of the biggest issues of the current workflows namely the API compatibility between releases. The Rock release master-16.09 embeds a major revision of one of the core packages namely *base/types* which led to binary incompatibilities between the releases master-16.08 and master-16.09. Hence, a consistent application of a single release had to be maintained during the operation.

To allow for reinstalling packages, all Debian package repositories including the Rock package repositories where mirrored for all required architecture[10] and available on-site. Satellite-based internet connection was limited and only usable for the remote operation of the system, thus not available for maintenance. The general setup of the repository mirrors allowed to maintain the standard workflow. Nevertheless, a handful packages were missing from the mirroring process, and the API incompatibilities required an update of ARM-based systems in use to master-16.09. The release master-16.09 had not been prepared for ARM-based system, such that a manual built had to be triggered on site. Unfortunately, the local building of binary packages for this architecture was not possible, since the corresponding base image was not available. All Rock core packages were used as binary packages, and no compilation issues (of dependant packages) and build inconsistencies were identified. This allowed to focus on the essential tasks of the test campaign.

4.4 Architectures and distributions

The support for multiple architecture and distributions has been verified and extended throughout the general development. The set of Rock core packages has been sucessfully build for (a) distributions such as Ubuntu 14.04,15.10,16.04 and Debian Wheezy and Jessie, and (b) for architectures: amd64, i386, armel and armhf. The architecture support for armel and armhf relies on building the packaging in the emulation environment qemu. Al-

[10]The tool *apt-mirror* `http://apt-mirror.github.io` was used for this purpose

though the process of building packages within a qemu environment is most likely slower than a pure cross-compilation approach, it supports a homogeneous workflow for all architectures. The generated packages for the ARM platform have been successfully deployed on the so-called payload items in the project TransTerrA and serve as basis for operating the camera which is required for the visual servoing processes.

4.5 Package deployment

To speed up the deployment of Rock onto remote machines, the project D-Rock [2] has fostered an ansible[11] based Rock bootstrap. The automation with *ansible* in combination with the usage of Rock's binary packages, allows to prepare operative system and development environment within minutes, and is suitable for a parallel deployment of multiple-systems.

5 LESSONS LEARNED

The following section will elaborate on the general lessons learned throughout the development of the binary packaging system for Rock including: (a) preferring a local build approach over an external build farm, (b) preferring standards over custom solutions, and (c) the importance of small features for user acceptance.

External Infrastructure The general development has started by anticipating the use of external build infrastructure and more specifically OpenSUSE. Although SUSE is known for the packaging format *rpm*, the infrastructure is also capable of building Debian packages. This initial approach triggered the implementation of the *deb_package* tool for Rock, which generates the initial artifacts *.orig.tar.gz, *.debian.xz, and *.dsc. The major drawback arising from developing the initial packaging approach in combination with the openSUSE infrastructure[12] was the dependency upon external servers, which made debugging tedious and time consuming. At the current stage the use of an external infrastructure can be reconsidered, since the build recipes of packages have been verified and fixed, which was not the case in the early development stage.

Jenkins Jenkins[13] is a Java-based solution typically used in the context of Continuous Integration (CI). Jenkins comes with a wide range of supporting plugins, e.g., a matrix-based control scheme to build for multiple architectures and multiple operating systems. The initial approach was to construct a single job to package, and build each Rock package and its defined dependencies accordingly. Unfortunately, operating with the Jenkins frontend for development purposes was slow, and still required customization scripts to trigger building software packages. In effect, Jenkins was an overhead to control the packaging and build process, and worse led to code and logic fragmentation. The use of Jenkins helped to structure the initial approach of standardization, but the direct use of the existing Debian toolchain in combination with autoproj has more advantages. The use of Jenkins can be reconsidered, when the existing set of source packages is relatively stable, and does not require significant maintenance as in the given case.

Standardization as road to success The generation of binary packages and usage has initially required substantial work in order to fulfill the policies of the Debian Build System. The lack of strict standardization of Ruby packages required adaption to allow each package to be converted to a Ruby gem as prerequisite for the use of *gem2deb*. An additional requirement arose through embedded C-extension, and the Ruby gem *rake-compiler* has been selected as means for standardization and to facilitate the building of C-extensions for different architectures. To reach a sufficient level of standardization, additional adaptation for a number of Ruby gems was required, these adaptions included fixing specification files and correcting build instructions. The large set of CMake based packages required only very few adaptation, and most requirements for change were related to building bindings for interpreter languages such as Ruby or Python. Generally, the ability to build a large set of CMake based packages shows the benefit of standards, which allow large-scale automation and improve robustness of a toolchain in general through ease of maintenance. The use of non-standardized packages comes with high maintenance cost, so that the benefit of a particular dependency should be well evaluated. Same holds for the standardization of workflows, which should be kept simple to maintain and simple to use at the same time.

User interaction and experience Developing the general architecture would not have been possible without early adopters, which dealt with some of the early restrictions of the approach. Only the early and continuous use of the generated binary packages has led to a transparent integration into the standard Rock workflow. Although the use of binary packages offers significant benefits, this is not sufficient in order to attract a user base. Only an easy to use, smooth, while still transparent and robust workflow for Rock users will lead to the required user acceptance of the overall approach. The integration of binary Rock packages into autoproj therefore tried to minimize the changes to the existing development workflow. Additional features were required to enable users to identify and report errors, e.g., by providing an access to meta-information for each binary package.

[11] Ansible `http:/www.ansible.com`
[12] openSUSE Homepage: `https://www.opensuse.org`
[13] Jenkins Homepage: `https://jenkins.io`

Internet dependency The initial setup of a host that can build binary Rock packages requires a connection to the Internet in order to retrieve images of the target distributions and to retrieve installation packages. For offline operation it is therefore necessary to anticipate all required architectures and prepare the base images accordingly; likewise all required package repositories have to be mirrored and made accessible offline. Again, Ruby gems require a special treatment and all required gems have to be bootstrapped. Only these preparatory steps will allow for a build process, which will support full offline operations.

6 CONCLUSION & OUTLOOK

The Debian community has paved the way for most elements of the binary Rock packaging system. In combination with the meta-build system autoproj a tool has been developed which allows for the generation of binary releases and which allows to maximise the reuse of these packages for derivative works. Facilitating the flexible reuse of binary packages using hierarchical release structures is expected to lead to a significant productivity gain for robotic software development and maintenance of robots in general.

Since the Debian packaging system is used, the installation process hooks into the common apt-based system. Therefore, new users require no special training, and the approach almost transparently hooks into existing autoproj-based installations. The result is a system which leverages the stability of existing workspaces and the quality of the software stack. At the same time it applies minimal constraints to developers to use Rock's traditional workflow and the Rock's Debian packages for existing installations.

Future work will focus on improving the workflow for hierarchical releases for Rock users, and evaluating the application of the build infrastructure for Python libraries.

Acknowledgment

The development of the Binary Packaging Architecture has been supported by the German Space Agency (DLR Agentur) with federal funds of the Federal Ministry of Economic Affairs and Energy for the project TransTerrA under grant agreement 50RA1301 to design the basic architecture, and for the project VIPE under grant agreement 50NA1516 to use and evaluate the application of the binary packaging. The development of the local build infrastructure in the project D-Rock has been supported by the Federal Ministry of Education and Research under grant agreement 01IW15001.

References

[1] Mark Dequènes et al. *The Common Debian Build System*. Available at: https://build-common.alioth.debian.org/cdbs-doc.html, (Accessed: 9 April 2018). 2018.

[2] DFKI GmbH Robotics Innovation Center. *D-Rock: Model, methods and tools for the model based software development of robots*. Available at: https://robotik.dfki-bremen.de/en/research/projects/d-rock.html, (Accessed: 3 January 2018). 2018.

[3] DFKI GmbH Robotics Innovation Center. *The Robot Construction Kit*. Available at: http://www.rock-robotics.org, (Accessed: 22 May 2011). 2011.

[4] ESA. *ESA Lab Rover Prototypes*. Available at: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Automation_and_Robotics/Lab_Rover_prototypes, (Accessed: 5 January 2018). 2018.

[5] Sylvain Joyeux. *autoproj*. Available at: https://github.com/rock-core/autoproj, (Accessed: 9 April 2018). 2018.

[6] Anis Koubaa, ed. *Robot Operating System (ROS)*. Vol. 707. Studies in Computational Intelligence. Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-54927-9.

[7] Daniel Kuehn, Alexander Dettmann, and Kirchner Frank. "Analysis of Using an Active Artificial Spine in a Quadruped Robot". In: *Proceedings of the 4th International Conference on Control, Automation and Robotics (ICCAR)*. Auckland, NZ: IEEE, 2018.

[8] Thomas M. Roehr, Florian Cordes, and Frank Kirchner. "Reconfigurable Integrated Multirobot Exploration System (RIMRES): Heterogeneous Modular Reconfigurable Robots for Space Exploration". In: *Journal of Field Robotics* 31.1 (Jan. 2014), pp. 3–34. DOI: 10.1002/rob.21477.

[9] Roland Sonsalla et al. "Field Testing of a Cooperative Multi-Robot Sample Return Mission in Mars Analogue Environment". In: *Proceedings of the 14th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA 2017)*. Leiden, NL, 2017.

[10] Roland Sonsalla et al. "Towards a Heterogeneous Modular Robotic Team in a Logistic Chain for Extraterrestrial Exploration". In: *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*. Montreal, Canada: International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2014.