ROS Discover Evaluation

This is the replication package for the paper, **ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems,** which has been accepted at the International Conference on Software Architecture (ICSA), 2021. A preprint of the paper is included in this replication package (`paper.pdf`).

This artifact is archived on Zenodo with the following DOI: https://doi.org/10.5281/zenodo.5834633

The study associated with this artifact was carried out by the following investigators:

- Christopher S. Timperley (Carnegie Mellon University)
- Tobias Dürschmid (Carnegie Mellon University)
- Bradley Schmerl (Carnegie Mellon University)
- David Garlan (Carnegie Mellon University)
- Claire Le Goues (Carnegie Mellon University)

If you have any questions regarding the research or the replication package, you should contact Christopher, Tobias, or Bradley.

Contents

This replication package is structured as follows:

```
- architecture-style/    The definition of the ROS architeture style used for analysis.
- deps/                  Contains the code that the evaluation pacakges uses.
  |- rosdiscover/        The code for the implementation of the rosdiscover system
  |                      evaluated in the paper
  |- roswire/            The code for the layer used for interacting with ROS
  |                      Docker images
  |- rosdiscover-cxx-extract/
  |                      The code for static analysis
- docker/                Files used for building Docker images used in the experiments
- experiments/           Data for setting up the experiments, and the results we got
  |- detection/          Experiments that we used in RQ3
  |- recovery/           Experiments we used in RQ1 and RQ2
```

```
- results/               Contains raw and processed results from running the scripts
  |- DataAnalysis.ipynb  A Jupyter notebook used for processing the results
  |- data/               Various CSV files used for summarizing data for the paper
  |  |- RosTopicBugs - Bug Data Set.csv
  |                      The misconfiguration bug dataset contributed in the paper
  |- detection/          Results for the detection experiments
  |- recovery/           Results for the recovery experiments
- scripts/               Python scripts for running and analyzing the experiments
                         (see more below)
- INSTALL.rst            Installation instructions for this replication package
- STATUS.rst             A document stating the claims made about this replication package
- LICENSE                The license under which this replication package is made available
- Pipfile                Used to describe the dependencies of the Python virtual environment
- Pipfile.lock           Used to provide the exact versions of all packages (i.e., transiti
- paper.pdf              The final version of the paper **ROSDiscover: Statically Detecting
```

Overview of the ROSDiscover Toolchain

This replication package contains, in addition to scripts for replicating the result, the source code for the complete ROSDiscover toolchain.

- **ROSDiscover**: This is the tool that is described in the paper. It is designed to, among other purposes, recover run-time architectures from ROS applications provided in the form of a Docker image and an accompanying configuration file. Further instructions on the general use of ROSDiscover can be found in its README file, available either in its archival form in the deps/rosdiscover directory of this artifact, or, preferably, in its up-to-date form on GitHub at: https://github.com/rosqual/rosdiscover. ROSDiscover has commands for recovering component models, observing running systems to produce observed architectures, and statically assembling architetures to form recovered architectures.
- **ROSWire**: This is a standalone Python library, used as part of the ROSDiscover toolchain, that provides extensive functionality for building static and dynamic tools for ROS that accept Docker images as their input (rather than assuming that those tools are located on the same machine as the subject of the analysis). More details about ROSWire can be found in its README file, available either in its archival form in the deps/roswire directory of this artifact, or, preferably, in its up-to-date form on GitHub at: https://github.com/rosqual/roswire.
- **CXX-Extract**: Provides the implementation of the static component model recovery of ROS nodes from source code written in C++.

When ROSDiscover is invoked to recover an architecture, it uses ROSWire to locate packages, launch files, etc. ROSDiscover subsequently invokes CXX-Extract when it encounters a node in a launch file it is processing, to parse the source, identify ROS API calls, and produce a component model. ROSDiscover then combines the component models according to the launch files being processed and resolves any parameters, arguments, unbound topics, etc. that may be in the component models to produce an architecture model.

Replicating results for the paper

To aid in replicating the results of the research, we have provided a set of scripts that ease the reproduction of each step in a research question, along with an experiment definition for each experiment. The definition uses YAML, and provides all the information for building containers, recovering nodes, extracting and checking architectures, and detecting misconfigurations. In these instructions (except for misconfiguration bug detection) we will use `autorally` as an example, with the experiment defined in `experiments/recovery/subjects/autorally/experiment.yml`.

You may run the experiments either natively on the host machine via Python, as described in Section 2.2 of INSTALL.rst, or, preferably, via the Docker-based evaluation toolchain, described in Section 2.1 of INSTALL.rst. **NOTE: In order for this to work, the container will need to connect to the Docker socket that is running on the host.** In the instructions below, we give two versions of each command. One, prefixed by `(native)$` is how to run the command from the host; the other `(container)$` is how to run the command using the provided helper script that connects to the evaluation Docker container.

To obtain a list of commands that can be executed inside the replication package, execute the `help` command as shown below from the root of the replication package.

```
(docker) $ docker/run.sh help
(native) $ pipenv run help
```

You can also obtain a list of all of the experiments using the `list` command:

```
(native) $ docker/run.sh list
(native) $ pipenv run list
```

Run recovery of all nodes in images for RQ1

To run the component model recovery experiments described in RQ1, you should use the `recover-node-models.py` script provided in the experimental scripts directory. The script simply takes the name of a subject system for RQ1 and emits a set of component models (in JSON) form, along with a summary of the success of the overall process (recovered-models.csv), describing the number of API calls that were found and successfully resolved for each individual node in that subject system.

```
(docker)$ docker/run.sh recover-node-models autorally
```

```
(docker)$ docker/run.sh recover-node-models autoware
(docker)$ docker/run.sh recover-node-models fetch
(docker)$ docker/run.sh recover-node-models husky
(docker)$ docker/run.sh recover-node-models turtlebot

(native)$ pipenv run scripts/recover-node-models.py autorally
(native)$ pipenv run scripts/recover-node-models.py autoware
(native)$ pipenv run scripts/recover-node-models.py fetch
(native)$ pipenv run scripts/recover-node-models.py husky
(native)$ pipenv run scripts/recover-node-models.py turtlebot
```

The results for each system are written to its corresponding `results/recovery/subjects/autorally`. The files that are produced are:

- a `models` directory that contains JSON formatted information for the component models of each node that was analyzed by the system. The filename is of the form `{package}__{node}.json`.
- a `recovered-models.csv` that records, for each node and package, its entrypoint, the time it took to do the static analysis, whether it crashed or produced an error message, the number of statements, functions, and relevant API calls encountered, and then information about unresolved (unknown) and unreachable code.

To reproduce the analysis used in the paper, the CSV file for each system should copied into `results/data/` directory and given the name `RQ1 node model recovery results - <system>.csv`.

Derive and check architecture for RQ2

The experimental setups for RQ2 are in the `experiments/recovery/subjects` directories. We currently report results for recovery in `turtlebot`, `autorally`, and `husky`. RQ2 consists of two phases followed by checking and comparison of results. All the examples will be given or `autorally` but should be the same for the other subjects. All commands are executed in the root directory of this package.

Note that, for convenience, we provide a shell script that automates all the steps below. It assumes that all the images have been prebuilt as described above. To run this:

```
(docker)$ docker/run.sh rq2 [autorally | husky | turtlebot]
(native)$ scripts/rq2.sh [autorally | husky | turtlebot]
```

If no arguments are given, the script will run through all three cases. After running the steps for reproducing RQ2, a human readable form of the comparison will be in `results/recovery/subject/<system>/compare.observed-recovered.txt`, where `<system>` is one of `autorally | husky | turtlebot`. A side-by-side comparison of the architectures, and the metrics calculated, are in the last to sections of this file.

The rest of this section describes how to reproduce RQ2 step by step.

Step 1: Derive the ground truth by observing the running system.

```
(docker)$ docker/run.sh observe autorally
(native)$ pipenv run scripts/observe-system.py autorally
```

This will take a while to run because it needs to start the robot, start a mission, and then observe the architecture multiple times. In the end, a YML representation of the architecture will be placed in `experiments/recovery/subjects/autorally/observed.architecture.yml`.

Step 2: Run ROSDiscover to statically recover the system.

```
(docker)$ docker/run.sh recover recovery autorally
(native)$ pipenv run scripts/recover-system.py recovery autorally

INFO: reconstructing architecture for image [rosdiscover-experiments/autorally:c2692f2]
...
INFO: applying remapping from [/camera/left/camera_info] to [/left_camera/camera_info]
INFO: applying remapping from [/camera/right/camera_info] to [/right_camera/camera_info]
INFO: statically recovered system architecture for image [rosdiscover-experiments/autorally:
```

This will process the launch files supplied in the `experiment.yml` and produce the architecture in `experiments/recovery/subjects/autorally/recovered.architecture.yml`. The first time this is run it may take some time because ROSDiscover needs to statically analyze the source for the nodes mentioned in the launch files, but thereafter those results are cached and the analysis will run more quickly.

Step 3: Check and compare the architectures of the observed and recovered systems. This involves three steps.

Step 3a: Produce and check the architecture of the observed system.

```
(docker)$ docker/run.sh check observed recovery autorally
(native)$ pipenv run scripts/check-architecture.py observed experiments/recovery/subjects/au

INFO: Writing Acme to /code/experiments/recovery/subjects/autorally/recovered.architecture.a
INFO: Writing Acme to /code/experiments/recovery/subjects/autorally/recovered.architecture.a
INFO: Checking architecture...
Checking architecture...
...
ground_truth_republisher  publishes to an unsubscribed topic: '/ground_truth/state'. But the
with a similar name that subscribes to a similar message type. ground_truth_republisher was
```

The result is placed in `experiments/recovery/subjects/autorally/observed.architecture.acme`

Step 3b: Produce and check the architecture of the recovered system.

```
(docker)$ docker/run.sh check recovered recovery autorally
(native)$ pipenv run scripts/check-architecture.py recovered experiments/recovery/subjects/a
```

```
INFO: Writing Acme to /code/experiments/recovery/subjects/autorally/recovered.architecture.a
INFO: Writing Acme to /code/experiments/recovery/subjects/autorally/recovered.architecture.a
INFO: Checking architecture...
Checking architecture...
...
ground_truth_republisher  publishes to an unsubscribed topic: '/ground_truth/state'. But the
with a similar name that subscribes to a similar message type. ground_truth_republisher was
/autoRallyTrackGazeboSim.launch.
```

The result is placed in `experiments/recovery/subjects/autorally/recovered.architecture.acme`

Step 3c: Compare the architectures.

```
(docker)$ docker/run.sh compare autorally
(native)$ pipenv run scripts/compare-recovered-observed.py autorally
```

The comparison output is placed in `experiments/recovery/subjects/autorally/compare.observed-recove`
The analyzed results used in the paper are in `experiments/recovery/subjects/autorally/observed.recove`

If you look at the file `experiments/recovery/subjects/autorally/observed.recovered.compare.csv`, it is divided into five sections.

1. Observed architecture summary. This summarizes the observed architcture. It is a summarization of `experiments/recovery/subjects/autorally/observed.architecture.a`
2. Recovered architecture summary. This summarizes the recovered architecture. It is a summarization of `experiments/recovery/subjects/autorally/recovered.architecture`
3. Provenance information. This summarizes the component models used in recovery that were handwritten and recovered.
4. Side-by-side comparison: This gives a side by side comparison of the details of the architecture, giving topics etc that were observed for a node, those that were recovered. Upper case elements are those that appear in both the observed and recovered architectures, those in lower case only appear in one.
5. Differences: A summary of the statistics for over-approximation/under-approximation for the whole system (not that in `observed.recovered.compare.csv` we divide these numbers into handwritten and recovered, and only use the recovered metrics in the paper.

Run configuration mismatch bug detection for RQ3

To run configuration mismatch bugs for RQ3 involves building another set of Docker images for each robot system at the time the misconfiguration was extant and the time at which it was fixd. Like the other RQs, we use the same scripts for building these images. We will use the example of the `autorally-01` bug which is an error that was introduced into the `autorally_core/launch/stateEstimator.launch` file that incorrectly remapped a topic. The format of the experiment definition for detection replication is different to the other experiment definitions, containing information on how to build the buggy and fixed Docker images, the errors that are expected to be found, and definition of a reproducer node that guarantees use of the broken connector. We provide the pre-built images. See INSTALL.rst.

To reproduce the results for RQ3, we have provided a script that automates the process above for the detection experiment. The script:

1. Recovers the architectures of both the buggy and fixed versions, as described in the corresponding experiment.yml.
2. Applies architectural rule checking to both architectures and outputs any found errors
3. Summarizes the results. The results first print any architecture errors found in the buggy version of the system, followed by any architecture errors in the fixed version. If the buggy version contains errors, but the fixed version prints out **NO RELEVANT RESULTS** this means we have succcessfully detected the bug.

To run RQ3 reproduction on all the systems where we successfully detected the misconfiguration:

```
(docker)$ docker/run.sh rq3
(native)$ pipenv run rq3
```

This will run RQ3 on all the images that we were successful in detecting: autorally-01, autorally-03, autorally-04, autoware-01, autoware-11 husky-02 husky-04 husky-06. To run on an individual example:

```
(docker)$ docker/run.sh rq3 autorally-01
(native)$ pipen run rq3 autorally-01
```

Results Data

Raw results

The replication package also provides results that we used in the paper. Data for each detection case is in

```
results/detection/subjects/[autorally-N, autoware-N, ...]
```

For each case where we could duplicate the misconfiguration, there is a `buggy.architecture.[yml,acme]`, `fixed.architecture/[yml,acme]` that define the architecture recovered and an `error-report.csv` that reports whether we captured the misconfiguration error or not.

The results for the recovery case is in:

```
results/recovery/subjects/[autorally, husky, ...]
```

Each case has the following files:

```
[recovered,obeserved].architecture.[yml,acme]   - recovered and observed architectures
compare.observed-recovered.txt                  - a human readable summary of the compariso
```

```
observed.recovered.[compare,errors].csv         - a CSV version of the comparison results,
                                                  with errors detected
recovery.rosdiscover.yml                        - a script generated config file passed to r
recovered-models.csv                            - a list of models recovered for RQ1 and the
                                                  metrics
```

Processed Results and Data Analysis

In order to produce the results presented in the paper, we combined the results into various files that can be analyzed by a Jupyter notebook. These can be reproduced.

The data collected for the experiments of RQ1 are in these files:

- results/data/RQ1 node model recovery results - autorally.csv
- results/data/RQ1 node model recovery results - autoware.csv
- results/data/RQ1 node model recovery results - fetch.csv
- results/data/RQ1 node model recovery results - husky.csv
- results/data/RQ1 node model recovery results - turtlebot.csv

The data collected for the experiments of RQ2 are in these files:

- results/data/RQ2 Observed Architecture - Comparison.csv
- results/data/RQ2 Observed Architecture - Models.csv
- results/data/RQ2 Observed Architecture - Node-Level Comparision.csv
- results/data/RQ2 Observed Architecture - Summary.csv

To reproduce the comparison files, you can run:

```
(native)$ pipenv scripts/gather-rq2-results.py
(container)$ docker/run.sh gather-rq2
```

This pulls information out of the `compare.observed.recovered.csv` files into the Comparison CSVs mentioned above. They can the be analyzed like mentioned below.

The data collected for the experiments of RQ3 is in: `results/data/RosTopicBugs - RQ3 - Results Table.csv`

The Jupyter Notebook in `results/DataAnalysis.ipynb` uses these results to produce the numbers in the paper. To run this analysis, you can run the following command:

```
(native)$ pipenv run jupyter notebook --ip=0.0.0.0 --port=8080 --no-browser results/DataAnal
(container)$ docker/run.sh jupyter notebook --ip=0.0.0.0 --port=8080 --no-browser results/Da
```

This will start the Jupyter notebook, which can be accessed by opening a browser to the address: 192.168.0.1:8080

Results Format

The Jupter notebook writes the results into these files:

- results/RQ1.csv (which includes the nodel-level accuracy results shown in Table III in the paper)
- results/RQ1_unreachable.csv (which includes the nodel-level static analysis results of unreachable statements and functions)
- results/RQ2.csv (which includes the system-level static analysis accurary results shown in Table IV in the paper)
- results/RQ2_architectural_element.csv (which includes the system-level static analysis accurary results per architectural element shown in Table V in the paper)
- results/RQ2_handwritten.csv (which includes the system-level accurary of handwritten models discusssed in Section IV.B. RQ2 – System Architecture Recovery - Results of the paper)
- results/RQ2_handwritten_architectural_element.csv (which includes the system-level accurary of handwritten models discusssed in Section IV.B. RQ2 – System Architecture Recovery - Results of the paper per architectural element)
- results/RQ3.csv (which includes the data shown in Table VI of the paper)

Furthermore, results/modelSizes.csv lists the lines of code for each handwritten model of the corresponding file in deps/rosdiscover/src/rosdiscover/models.

Running different experiments

The experiment pipeline is designed for flexible modification to run different experiments (e.g., other bugs, or bugs in other systems).

Experiment Configuration File Format

Each experiment is set up in a configuration YAML file (such as in /experiments/detection/subjects/husky-01/experiment.yml).

```yaml
type: detection
subject: husky
distro: kinetic
build_command: catkin_make -DCMAKE_EXPORT_COMPILE_COMMANDS=1
missing_ros_packages:
- yaml-cpp
exclude_ros_packages:
- lms1xx
- orocos_kdl
- python_orocos_kdl
- opencv3
- diagnostics
- diagnostic_updater
- diagnostic_aggregator
- diagnostic_msgs
- std_srvs
- tf
- tf2_eigen
- tf2_geometry_msgs
- tf2_kdl
- tf2_msgs
- tf2_py
- tf2_ros
- tf2_sensor_msgs
- message_relay
apt_packages:
- ros-kinetic-orocos-kdl
- libyaml-cpp-dev
- ros-kinetic-tf
- ros-kinetic-tf2-sensor-msgs
- ros-kinetic-control-msgs
- ros-kinetic-message-relay
buggy:
  docker:
    type: templated
    image: rosdiscover-experiments/husky:dc8169b6b7b9cfe37497f222adbe5f20bb83495a
  repositories:
  - name: husky
    url: https://github.com/husky/husky.git
    version: dc8169b6b7b9cfe37497f222adbe5f20bb83495a
fixed:
  docker:
    type: templated
    image: rosdiscover-experiments/husky:97c5280b151665704f8f8e3beecb3e6e89ea14ae
  repositories:
  - name: husky
    url: https://github.com/husky/husky.git
    version: 97c5280b151665704f8f8e3beecb3e6e89ea14ae
sources:
- /opt/ros/kinetic/setup.bash
- /ros_ws/devel/setup.bash
launches:
- /ros_ws/src/husky/husky_gazebo/launch/spawn_husky.launch
- /ros_ws/src/husky/husky_navigation/launch/amcl_demo.launch
- /ros_ws/src/husky/husky_gazebo/launch/husky_playpen.launch
```

The `subject` tag describes the name of the system (e.g. husky, autoware, or turtlebot). The `type` tag can either be `detection` (with a buggy and fixed version for RQ3) or `recovery` for a single-version experiment for RQ2. This tag defines what format the experiment is described. For detection experiments, the project sources are be specified for buggy and fixed versions separately:

```
buggy:
  docker:
    type: templated
    image: rosdiscover-experiments/husky:dc8169b6b7b9cfe37497f222adbe5f20bb83495a
  repositories:
  - name: husky
    url: https://github.com/husky/husky.git
    version: dc8169b6b7b9cfe37497f222adbe5f20bb83495a
fixed:
  docker:
    type: templated
    image: rosdiscover-experiments/husky:97c5280b151665704f8f8e3beecb3e6e89ea14ae
  repositories:
  - name: husky
    url: https://github.com/husky/husky.git
    version: 97c5280b151665704f8f8e3beecb3e6e89ea14ae
```

The `repositories` tag describes a list of repositories to be included according to the following specification. The `url` specifies the URL to the git repository that should be cloned for analysis. The `version` specifies the commit ID or tag that should be checked out for analysis. The `image` tag specifies the name that the Docker image should have, which will be used when running the experiment as well. The `type` tag specifies the Docker image type and can be `templated` for generated an image based on a generic approach that uses a parameterized Dockerfile (see section ”Parameterized Dockerfile” below), or `custom` for separately provided Dockerfiles (e.g., for forwardporting). If custom is used, the Docker tag needs an `filename` child-tag specifying the file name of the custom Dockerfile (with a path relative to the experiment.yml file and the path to the context used by Docker to create the image) to be used to build the image, such as for the Autoware recovery image:

```
docker:
  type: custom
  image: rosdiscover-experiments/autoware:static
  filename: ../../../../docker/Dockerfile.autoware
  context: ../../../../docker
```

The `errors` tag lists the topic names for which an error is expected.

For recovery experiments the buggy content of the buggy / fixed tag is included in the root XML tag, since there is only one version. For each version of the system, the ROS package dependencies are determined by analyzing all package.xml files that can be found recursively in the listed repositories. All dependencies includes as ”depend”, ”build_depend”, ”build_export_depend”, or ”run_depend” will be added to the image. The corresponding historically accurate versions are determined using https://github.com/rosin-project/rosinstall_generator_time_machine based on date of the specified commit in the version tag of the repository. If multiple repositories are included and therefore multiple versions are provided the image construction process uses the most recent one among them.

The rest of the format is identical for both experiment types.

The `distro` is the name of ROS distribution in which the bug is supposed to be replicated. Examples include indigo, kinetic, lunar, and melodic. The experiment infrastructure will use the corresponding ROS distribution as a basis and install the system and its corresponding dependencies in the stated ROS distribution. The `missing_ros_packages` tag specifies as list of additional ROS packages that should be installed in the image, additionally to those listed in the package.xml files that can be found recursively in the project directories. The `exclude_ros_packages` tag specifies a list of ROS packages that are includes int the project’s package.xml files but should not be installed in the image. Packages can be excluded here either if they result in build errors, if they are installed manually, or if the package.xml is incorrect and those packages should not be installed. The `apt_packages` tag specifies a list of Linux packages that should be installed using `apt-get install <packages>` before the system is built. Those can include dependencies, libraries, or build tools used by the project. The `build_command` tag specifies the Linux command used to build the project from source (e.g., `catkin_make -DCMAKE_EXPORT_COMPILE_COMMANDS=1` or `catkin build -DCMAKE_EXPORT_COMPILE_COMMANDS=on`). Since rosdiscover analyzes the compiler commands used to build the project, the build command must include the corresponding CMake flags to export compiler commands. The `sources` tag specifies the bash scripts that should be sourced before building the project. This includes the ROS distribution and the catkin workspace but may also include custom other source files. The `cuda_version` tag specifies the CUDA version that should be installed, if any (e.g., 6-5). The `launches` tag includes the file names of the launch files to be launched by the experiments and optionally launch file arguments specified as key-value dictionary with keys being argument names and values being the values to which the arguments should be set, such as in autoware-01:

```
launches:
  - filename: /ros_ws/src/autoware/ros/src/util/packages/runtime_manager/scripts/launch_file
  - filename: /ros_ws/src/autoware/ros/src/util/packages/runtime_manager/scripts/launch_file
    arguments:
      tf_launch: /.autoware/data/tf/tf.launch
      pmap_param: noupdate
      pcd_files: /.autoware/data/map/pointcloud_map/bin_Laser-00147_-00846.pcd /.autoware/da
      csv_files: /.autoware/data/map/vector_map/road_surface_mark.csv /.autoware/data/map/ve
```

Parameterized Dockerfile

Most images that are needed to analyze and/or reproduce bugs have require the same steps to install all required content. Therefore, we use a generic Dockerfile (located in `docker/Dockerfile`) that can be parameterized to construct a replication environment for historic versions of ROS systems. This has the advantage that it the specification of what is installed for each project version is very small and structured systematically. Furthermore, since many versions will share identical installation steps, Docker automatically reuses existing layers, which reduces the image build time and the required storage for the resulting images.

The Dockerfile uses the Docker image of the corresponding ROS version (e.g., indigo, kinetic, melodic) as a parent, installs common tools to interact with Docker containers, such as VNC, build tools for Python and ROS, and common libraries. Then it installs the specific versions of the dependencies listed in the experiment config. Finally, it compiles the source code of the project.

To customize the build process, the Dockerfile is configured to execute optional preinstall, prebuild, and/or postbuild scripts located in the Docker folder of the corresponding experiment:

- The preinstall script (preinstall.sh) runs before the ROS dependencies are installed and can be used to, for example, configure the Python installation in cases in which the ROS dependencies do not install correctly.
- The prebuild script (prebuild.sh) runs directly before the project is compiled and can be user to install additional dependencies that cannot simply be installed as an apt-get package or ROS package (for example because it needs to be built from source or because it needs to be downloaded from a custom location). The prebuild script can also be used to perform small changes to the source code (for example if the current version has a compiler error that can be fixed very easily, or if the CMake.list is missing dependencies).
- The postbuild script (postbuild.sh) runs as the final step during image creation can be used to, for example, make changes to the launch files of a system.

The generic Dockerfile has the following arguments that are initialized based on the information provided in the experiment configuration file or will be automatically determined by the infrastructure in `scripts/build-image.py`:

- `DISTRO`: The ROS distribution (e.g., indigo, kinetic, melodic). This parameter is taken from the experiment configuration YAML file.
- `COMMON_ROOTFS`: The directory on the host machine that is copied into the root directory of the Docker image. This parameter is automatically set.
- `CUDA_VERSION`: The CUDA version number to be installed, 0 if none is needed. This parameter is taken from the experiment configuration YAML file.

- `APT_PACKAGES`: The list of packages to be installed using apt-get install represented as string with spaces as separators. This parameter is taken from the experiment configuration YAML file.
- `DIRECTORY`: The "docker" subdirectory of the experiment directory that includes the preinstall, prebuild, and postbuild scripts as well as their dependent files to be copied to the Docker container for custom image building configuration steps. This parameter is automatically determined based on the location of the experiment folder.
- `ROSINSTALL_FILENAME`: The file name of the .rosinstall file that should be used to install ROS packages. This parameter is automatically determined based whether the buggy, fixed, or single version of the project should be built. The rosinstall file has been created using the https://github.com/rosin-project/rosinstall_generator_time_machine as described above.
- `BUILD_COMMAND`: The build command to be executed to compile the system. This parameter is taken from the experiment configuration YAML file.