Robot Runner: A Tool for Automatically Executing Experiments on Robotics Software

Stan Swanborn, Ivano Malavolta
Vrije Universiteit Amsterdam, The Netherlands
s.o.swanborn@vu.nl, i.malavolta@vu.nl

***Abstract*—Software is becoming *the* core aspect in robotics development and it is growing in terms of complexity and size. However, roboticists and researchers are struggling in ensuring and even *measuring* the quality of their software with respect to run-time properties such as energy efficiency and performance.**

This paper presents Robot Runner, a tool for streamlining the execution of measurement-based experiments involving robotics software. The tool is able to automatically setup, start, resume, and fully replicate user-defined experiments. Thanks to its plugin-based architecture, the tool is fully independent of the number, type, and complexity of the used robots (both real and simulated). GitHub repository – https://github.com/S2-group/robot-runner Youtube video – https://youtu.be/le-SAXI2k1E

I. Introduction

Context. The intensive usage of robots is becoming commonplace and successful in several industrial sectors like manufacturing, logistics, delivery, transportation, healthcare, security, entertainment [1]. Despite the advances in electronics and mechanics, the main barrier in robotics is *software*. As robot capabilities increase, so does the complexity of their controlling software, which is shifting towards an intricate network of interdependent components running millions of lines of code [6]. Roboticists are struggling in ensuring and even *measuring* the quality of their software with respect to run-time properties such as energy efficiency and performance [8]. When this fails, the result is faulty and inefficient robots, unpredictable project delays, and general lack of trust by users.

The tool. This paper presents Robot Runner (RR), a tool to automatically execute measurement-based experiments on robotics software. Starting from a self-contained Python-based description of the design of the experiment (and experiment-specific business logic), RR automatically executes several runs of a robotic system (while collecting measurement data) and packages collected measures in a ready-to-analyze format.

RR is designed as a plugin-based tool. Run-time profilers and other third-party components can be reused across experiments, thus streamlining the conduction of experiments in the area of robotics software. Plugins can produce different measures and multiple plugins can be used within a single experiment. Currently, we provide plugins for measuring the (i) energy consumption of battery-powered robots and (ii) CPU and memory utilization of UNIX-based robots. We preliminarily evaluated the tool via a series of experiments using a ROBOTIS TurtleBot3 robot as subject.

RR is based on the Robot Operating System (ROS) [7]. ROS is the de-facto standard for robotics software. It supports more than 140 types of robots and has a vibrant open-source ecosystem with several GitHub repositories containing ROS-based software, 4,152 publicly-available ROS packages, 7,696 ROS Wiki users, and 36,229 ROS Answers users [3].

Intended Usage. Firstly, researchers can use RR to conduct *new empirical studies on the run-time quality of robotics software*; in this context, once the design of the experiment is defined, the researcher can straightforwardly encode it into a (Python-based) configuration, launch it, and collect the ready-to-analyse measurement data. Secondly, researchers can easily *replicate already-conducted empirical studies* by reusing/customizing an already-existing experiment configuration. Thirdly, researchers can use RR to *benchmark run-time estimation models*. For example, a researcher proposing a new simulation-based energy model can use RR to rigorously compare it against the energy consumption collected from real robots by reusing exactly the same experiment definition and robotics missions. Practitioners can use RR to *evaluate alternative implementations/architectures/packages* over a common benchmark or a representative robotic mission. This helps practitioners in taking better informed decisions on their systems and lowering the risks encountered when reusing third-party software. This is specially valid in the ROS ecosystem, where several third-party packages exist for common robotic capabilities like object recognition, planning, Simultaneous Localization and Mapping (SLAM), etc. [6], [3]. In this context, having an objective assessment of which ROS packages are best suited for the system under development is fundamental, specially because there is evidence that in the ROS ecosystem third-party packages can suffer from unpredicted performance issues, outdated implementations, bugs [3].

II. Robot Runner

As shown in Figure 1, RR experiments involve two main subsystems: (i) RR acting as experiment orchestrator and (ii) the robotic system being measured. In the figure the robotic system is depicted as a Turtlebot3, but RR is fully independent of the number, type, and complexity of the robots used in the experiment. RR is implemented as a lightweight Python package and as such it can run on any machine able to run Python code, typically a laptop. The implementation of RR is fully independent from the robotic system being measured, provided that the latter runs on ROS and it receives commands and data as ROS messages.

This clear separation of roles gives us several advantages: (i) RR can run on any machine able to run Python code independently of the measured robotic system, (ii) the experiment does not affect the measured robotic system, mitigating potential internal threats to validity [9], (iii) the measured robotic system can be reused *as is* during the experiment, without needing to adapt it for making it compatible with the experiment being executed, and (iv) the same experiment can be easily run on either real or simulated robots, thus reducing future experiment modification/replication costs and speeding up experiments execution. RR is independent from the

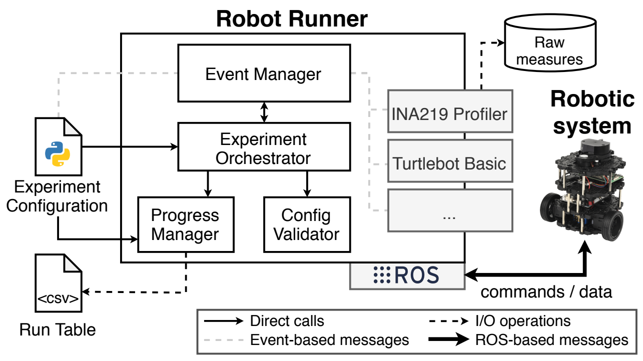


Fig. 1: Architecture of Robot Runner

A. *Experiment configuration*

The main input of RR is the *experiment configuration*. It provides a Python-based representation of the whole experiment. It allows the user to automatically *setup, start, resume, and replicate* an experiment, while treating the internal details of both the robotic system and RR itself as black boxes. The rationale for having a Python-based representation of the experiment (*e.g.,* instead of a static configuration file) is to allow users to flexibly define the business logic of their experiments in a self-contained manner. This is aligned with the ROS ecosystem, where launch configuration files are defined in Python in ROS2, instead of XML in ROS1.

```
1  # Import plugins
2  import plugins.systems.BasicTurtlebot3
3  import plugins.profilers.Ina219
4  import plugins.profilers.PsUtil
5
6  class RobotRunnerConfig:
7    # Static parameters
8    experiment_name: str = 'basic_experiment'
9    required_ros_version: int = 1
10   required_ros_distro: str = "melodic"
11   operation_type: OperationType = OperationType.SEMI
12   results_output_path: Path = Path('~/example_mission_results/')
13
14   def __init__(self):
15     # Events subscription
16     EventSubscriptionController.subscribe_to_multiple_events([
17       (RobotRunnerEvents.LAUNCH_MISSION, self.launch_mission),
18       # ...
19     ])
20
21   # Event callbacks
22   def launch_mission(self, context: RobotRunnerContext) -> None:
23     # ... code for launching the robotic mission
24   # Creation of the run table
25   def create_run_table(self) -> List[Dict]:
26     run_table = RunTableModel(
27       factors = [
28         Factor('Task', ['Computation', 'Video', 'Network']),
29         Factor('Repetition', range(1, 6))],
30       data_columns=['CPU', 'Memory', 'Energy']
31     )
32     run_table.create_experiment_run_table()
33     run_table.randomize_runs_order()
34     return run_table
```

Listing 1: Excerpt of basic experiment configuration

Listing 1 shows an excerpt of a basic experiment configuration[1]. The relevant information about the experiment is encapsulated into a dedicated class, which can be roughly divided into four main parts:

- *Import plugins* (lines 1-4): the user imports the plugins required for running the experiment (see Section III).
- *Static parameters* (lines 7-12): in addition to the name of the experiment (line 8), the user can define the required_ros_version and required_ros_distribution (lines 9-10) for ensuring that the experiment is executed on an expected environment. The operation_type allows users to execute the experiment in two different modes: (i) AUTO – on the completion of each run, the next run is started consecutively, without interruption, and (ii) SEMI – on the completion of each run, the next run is only started if the user signals to continue the experiment (this is useful in case a manual intervention is needed after each run, *e.g.,* substituting the battery of a robot). The results_output_path is used to specify the location for persisting the output of the experiment.
- *Event subscriptions* (lines 15-23): RR triggers a global event at specific points during the execution of the experiment (*e.g.,* before every run of the experiment, to start/stop the measurement, to launch a mission, etc.). Users can subscribe to those events (*e.g.,* lines 16-19) and implement their own callbacks containing the business logic for managing them (lines 21-23), *e.g.,* launching the robotic mission. The events currently supported by RR are shown in Figure 2.

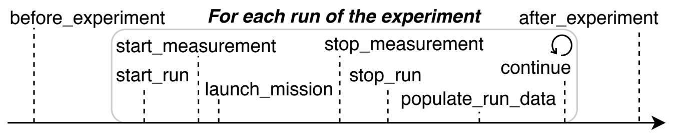


Fig. 2: Timeline of events triggered during an experiment

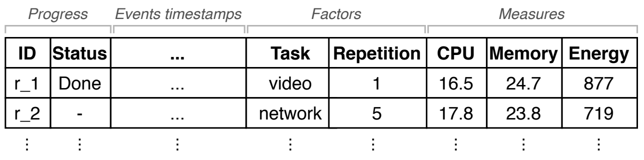
- *Creation of the run table* (lines 24-34): the run table contains the plan of all the runs of the experiment, together with the measures collected in each run (see Section II-B). The create_run_table function is meant to create the structure of the run table of the experiment. RR provides a dedicated API to concisely create a run table starting from the factors (and treatments) of the experiment, the dependent variables, and their (possibly randomized – line 33) combination. By following the principle of inversion of control, the create_run_table function is automatically called by RR.

B. *Output Produced by Robot Runner*

The main output of RR is a populated **run table**. Figure 3 shows a fragment of the run table created in Listing 1. The first two columns are used for tracking the *progress* of

[1]The full list of the parameters, events, and additional functionalities supported by RR is available in its GitHub repository.

the experiment, then we have a group of columns with the timestamp of each event show in Figure 2 (useful in case of problematic runs), then we have one column for the Task factor (which can have the 3 treatments shown in line 28), one column for $\mathrm{Repetition}$, which is used for repeating each task 5 times so to account for possible fluctuations of the collected measures, and finally we have the 3 columns for CPU, memory, and energy. In our example, the run table has 15 randomly-ordered rows (3 tasks, each with 5 repetitions).



Progress		Events timestamps	Factors		Measures		
ID	Status	...	Task	Repetition	CPU	Memory	Energy
r_1	Done	...	video	1	16.5	24.7	877
r_2	-	...	network	5	17.8	23.8	719
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 3: Fragment of run table

If required, users can subscribe to the $\mathrm{populate\_run\_data}$ event (see Figure 2) to intercept the measures collected after each run and apply their own experiment-specific aggregation policy. In this way, plugin developers can focus on producing the measures and providing a default aggregation policy, instead of trying to cover all possible use cases for their plugin.

RR reserves a dedicated location in the file system for each plugin; this location can be used by the plugin developer to store the **raw measures** collected during the experiment. The format of the persisted raw data is plugin-dependent and is decided by the developer of each plugin according to their specific needs/constraints.

C. Internal Components and Plugins

Experiment Orchestrator. Given the run table of the experiment, the Experiment Orchestrator is responsible for executing each run of the experiment. Before the actual execution of the experiment, the orchestrator interacts with the Config Validator for checking if the experiment configuration is well formed, otherwise the experiment is aborted with clear error messages and suggested solutions. Then, the orchestrator iterates over each row of the run table and executes it accordingly. During the experiment, the orchestrator (i) calls the Event Manager in order to trigger the key events shown in figure 2 and (ii) informs the Progress Manager about the status of the run being executed (*i.e.,* whether it is successfully completed or aborted). Finally, at the end of the experiment, it informs the user about the final outcome of the experiment, where to find the results, and other diagnostics information.

Event Manager The Event Manager acts as message broker between the publishers and subscribers of the key events of the experiment. At the time of writing, plugins are the publishers and the experiment configuration is the subscriber (with a dedicated callback function for each relevant event). This mechanism allows third-party plugin developers to be independent both from the internals of RR and the experiment-specific business logic in the experiment configuration. Each event triggered by RR is enriched with (i) contextual information about the current run being executed (*i.e.,* the current combination of the treatments) and (ii) any plugin-specific information represented as a set of key-value pairs.

Config Validator. It performs diagnostic checks on the experiment configuration provided by the user (*e.g.,* check if all the static parameters fall within acceptable values) and provides user friendly warning/error messages.

Progress Manager. The first responsibility of this component is to build an in-memory representation of the run table. Here RR distinguishes between two cases: (i) if the CSV file of the run table does not exist, then the Progress Manager executes the user-provided $\mathrm{create\_experiment\_run\_table}$ function, otherwise (ii) it parses the contents of the CSV file. The latter case can be exploited by RR users in three ways: (i) to manually build the file of the run table for planning experiments with non-canonical trials or imbalanced designs, (ii) to resume the execution of experiments carried out in batches, or (iii) to restore the execution of an experiment after an unexpected error without incurring in data loss. During the experiment execution, the orchestrator visits each row of the run table, executes it, and informs the Progress manager about the completion of the run. Finally, this component persists the timestamps and collected measures of the current run.

Plugins. RR follows a plugin-based architecture where third-party software is developed independently of the experiment orchestration logic. In order to avoid clashes or data loss, the only requirements for plugin developers are: (i) plugins can persist raw data only in their reserved file system subtree and (ii) plugins can populate the run table with new measures only via the Event Manager. The rationale behind this design decision is to avoid having silos-like experiments where an ad-hoc software pipeline is developed for each experiment. Rather, we expect (and hope) that over time different teams and research groups focus their efforts on self-contained RR plugins, which can be opportunistically reused by the community. At the time of writing, three plugins are available for RR:

• *Turtlebot Basic*: implements a minimal robotic mission for benchmarking measurement-based experiments involving a Turtlebot3 robot. The mission is composed of the steps shown in Figure 4, where $< task >$ is always the same and, depending on the assigned treatment (see Listing 1), it can be either: (i) $\mathrm{computation}$: computing the Fibonacci sequence for 10 seconds; (ii) video: recording a video via 5-megapixel camera for 10 seconds; and (iii) $\mathrm{network}$: continuously issuing HTTP GET requests to the same remote address for 10 seconds. The mission is designed so that (i) it explores different potential sources of energy consumption, (ii) it is circular, *i.e.,* there is no need to reposition the robot at each run.

• *INA219 Profiler*: measures the energy consumed by battery-operated robots. Specifically, an Arduino NANO onboard the robot measures the power drawn from the battery via a INA219 high-side sensor (with a 200Hz sampling rate); at the end of the mission, the plugin (i) applies the $E = P \times t$ formula, where P = measured power and t = the duration of the mission and (ii) emits an RR event containing the value of E.

• *PsUtil Profiler*: encapsulates the ps UNIX utility and runs it onboard the Turtlebot3 robot at 10Hz, while retrieving information about the system memory and CPU utilization, both as percentage.

III. Robot Runner in Action

In this section we report the results of our example experiment described in Listing 1. The experiment uses the Turtlebot Basic plugin as benchmark and the two INA219 Profiler and PsUtil Profiler plugins for collecting run-time measures. For the sake of space, in this section we report the results on energy consumption only.

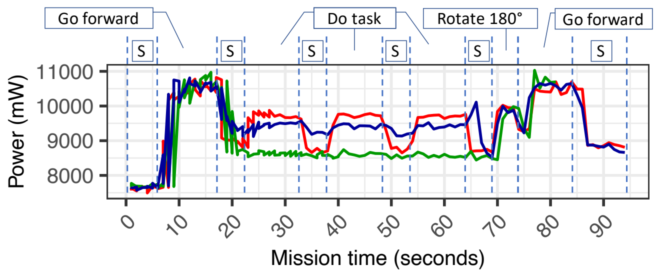


Fig. 4: Examples of power measures (computation, network, video, S= sleep for 5 seconds)

Figure 4 shows the power measurements collected during one randomly-chosen run for each treatment of the Task factor. RR and its INA219 plugin are able to clearly distinguish the various phases of the robotic mission just by looking at their power consumption, where we can observe some interesting phenomena: (i) the movement of the robot is the operation that consumes more power ($\sim$10.5W), (ii) the three types of tasks are clearly distinguishable, where computation and video recording tend to consume more power ($\sim$9.5W) than networking ($\sim$8W)[2], (iii) even when sleeping, after the initial movement the robot has a higher power consumption w.r.t. the first sleep operation, hinting that some hardware components of the robot are optimistically kept alive for subsequent usages.

For replicability and independent verification, the experiment definition, complete raw data, and full ROS environment are available as a self-contained Docker image in our replication package (this further exemplifies the versatility of RR).

IV. Related Work

To the best of our knowledge, RR is the first tool for the automatic execution of measurement-based experiments on robotics software. Similar tools exist in other domains, specially in energy-efficient mobile development. For example, we are maintaining Android Runner [5], a framework for executing experiments targeting Android native and web apps. As a matter of fact, the design of RR is based on the experience we gained from conducting more than 30 experiments via Android Runner. Specifically, RR inherits the customizability of experiments via external business logic, the plugin-based architecture, and the start/resume mechanism implemented in the Progress Manager. In addition to the technical differences due to the completely difference design space of the systems under measurement (*i.e.,* robots vs Android apps), the main difference between the two tools is that Android Runner is platform-specific and always assumes to be interacting with a physical and always-available device, whereas RR is completely independent from the type and number of used robots (either simulated or real). Other Android-specific tools include GreenMiner and PETrA. GreenMiner [4] is a hardware/software tool that physically measures the power used by an Android device, while automatically mining and executing mobile apps. PETrA [2] is another tool for estimating the energy consumption of Android apps via a pure software-based energy profiling technique.

[2]We speculate that networking exhibits an almost constant power consumption since its effect is masked by the high-frequency exchange of ROS messages performed by the sensors of the robot

V. Conclusion and Future Work

In this paper we presented Robot Runner, a tool for the automatic execution of measurement-based experiments on robotics software. The tool is one-of-its-kind since it masks the complexity of orchestrating experiments on robotics software; this allows roboticists and researchers to focus on what they are best at, *i.e.,* robotics software development, the design and data analysis of empirical studies targeting robots.

Robot Runner is one of the building blocks of a wider research effort targeting the energy efficiency of robotics software. Specifically, we are in the process of mining open-source software repositories and developer discussion platforms for building a catalog of architectural tactics for improving the energy efficiency of ROS-based systems; we will design and conduct several experiment for providing evidence about the run-time impact of each identified tactic in terms of quantitative metrics like tasks execution times, energy consumption of the robots, communication overhead, etc. This line of research is a step forward towards achieving guarantees on the quality of robotics software, allowing robots and the software that controls them to be an integral part of our daily life.

References

[1] From Internet to robotics: A roadmap for US robotics: 2020 Edition. http://www.hichristensen.com/pdf/roadmap-2020.pdf, Oct 2020. [Online; accessed 29. Oct. 2020].

[2] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *International conference on software analysis, evolution and reengineering (SANER)*, pages 103–114. IEEE, 2017.

[3] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry. The robot operating system: Package reuse and community dynamics. *Journal of Systems and Software*, 151:226–242, 2019.

[4] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Working Conference on Mining Software Repositories (MSR)s*, pages 12–21, 2014.

[5] I. Malavolta, E. Grua, C.-Y. Lam, R. de Vries, F. Tan, E. Zielinski, M. Peters, and L. Kaandorp. A framework for the automatic execution of measurement-based experiments on android devices. In *Conference on Automated Software Engineering Workshops*, pages 61–66. ACM, 2020.

[6] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan. How do you architect your robots? state of the practice and guidelines for ROS-based systems. In *International Conference on Software Engineering*, 2020.

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[8] S. Swanborn and I. Malavolta. Energy efficiency in robotics software: A systematic literature review. In *Conference on Automated Software Engineering Workshops*, pages 137–144. ACM, 2020.

[9] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.