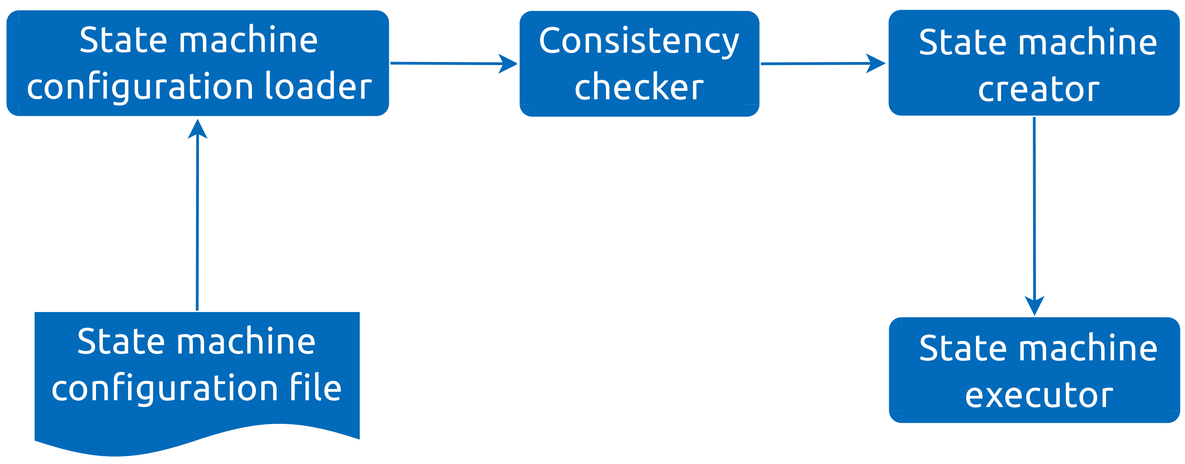
Reusable Specification of State Machines for Rapid Robot Functionality Prototyping

Alex Mitrevski and Paul G. Plöger
Autonomous Systems Group, Hochschule Bonn-Rhein-Sieg, Germany

Introduction

Rapid prototyping of robot functionalities often involves the use of *state machines*, which model the execution by a set of states and transitions between them. This is particularly the case when designing robot experiments, whose *reproducibility* is of utmost importance. Due to the simplicity with which state machines can be created, it is often the case that the specification of a state machine is interleaved together with the implementation; this however affects the reusability and transparency of the state machine. We address this problem with the help of a *Python-oriented domain-specific language for specifying state machines* and a small *Python library that allows state machines to be dynamically created*.

State Machine Library Diagram



State Machine Definition Language

We use a TOML-based language for specifying state machines.

```
sm_id = <string>
states = <list[string]>
outcomes = <list[string]>

[state_descriptions]
    [state_descriptions.STATE_NAME]
        state_module_name = <string>
        state_class_name = <string>
        initial_state = <bool>
        [state_descriptions.STATE_NAME.transitions]
            transition_1_name = <string>
            ...
            transition_n_name = <string>
        [state_descriptions.STATE_NAME.arguments]
            argument_1 = argument_1_value
            ...
            argument_n = argument_n_value
    ...

[arguments]
    argument_1 = argument_1_value
    ...
    argument_n = argument_n_value
```

Example 1: Pick and Place Experiment for a Domestic Robot

The first example we consider is one in which a domestic robot finds objects on a table, picks one of them, and then places it back on the table at a potentially different position.

```
sm_id = "simple_pick_and_place"
states = ["GO_TO_TABLE", "SCAN_TABLE", "PICK", "PLACE"]
outcomes = ["DONE", "FAILED"]

[state_descriptions]
    [state_descriptions.GO_TO_TABLE]
        state_module_name = "mdr_navigation_behaviours.move_base"
        state_class_name = "MoveBase"
        initial_state = true
        [state_descriptions.GO_TO_TABLE.transitions]
            succeeded = "SCAN_TABLE"
            failed = "GO_TO_TABLE"
            failed_after_retrying = "FAILED"
        [state_descriptions.GO_TO_TABLE.arguments]
            destination_locations = ["TABLE"]
            number_of_retries = 3

    [state_descriptions.SCAN_TABLE]
        state_module_name = "mdr_perception_behaviours.
          perceive_planes"
        state_class_name = "PerceivePlanes"
        [state_descriptions.SCAN_TABLE.transitions]
            succeeded = "PICK"
            failed = "SCAN_TABLE"
            failed_after_retrying = "FAILED"
        [state_descriptions.SCAN_TABLE.arguments]
            plane_prefix = "table"
            number_of_retries = 3

    [state_descriptions.PICK]
        state_module_name = "mdr_manipulation_behaviours.
          pick_closest_from_surface"
        state_class_name = "PickClosestFromSurface"
        [state_descriptions.PICK.transitions]
            succeeded = "PLACE"
            failed = "PICK"
            failed_after_retrying = "FAILED"
            find_objects_before_picking = "SCAN_TABLE"
        [state_descriptions.PICK.arguments]
            picking_surface_prefix = "table"
            number_of_retries = 3

    [state_descriptions.PLACE]
        state_module_name = "mdr_manipulation_behaviours.place"
        state_class_name = "Place"
        [state_descriptions.PLACE.transitions]
            succeeded = "DONE"
            failed = "PLACE"
            failed_after_retrying = "FAILED"
        [state_descriptions.PLACE.arguments]
            placing_surface_prefix = "table"
            number_of_retries = 3
```

This specification allows easy state machine transfer between robots - a consequence of the fact that state machines are loaded dynamically.

Example 2: Docking to a Cart and Entering an Elevator for a Logistics Robot

A second example we consider involves an experiment in which a logistics robot needs to dock to a cart and then enter an elevator immediately after docking. This experiment thus involves two states: docking to the cart and then entering an elevator (where both states are state machines themselves).

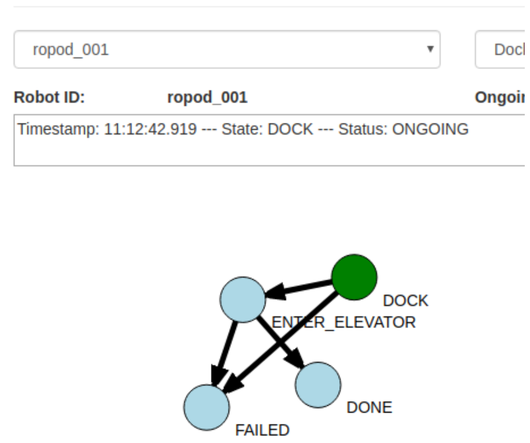
The state machine specification of this experiment is given below.

```
sm_id = "dock_and_enter_elevator"
states = ["DOCK", "ENTER_ELEVATOR"]
outcomes = ["DONE", "FAILED"]

[state_descriptions]
    [state_descriptions.DOCK]
        state_module_name = "ropod_experiment_executor.commands.dock"
        state_class_name = "Dock"
        initial_state = true
        [state_descriptions.DOCK.transitions]
            done = "ENTER_ELEVATOR"
            failed = "FAILED"
        [state_descriptions.DOCK.arguments]
            area_id = "Area1"
            area_name = "CartArea1"
            dock_action_topic = "/ropod_task_executor/DOCK"
            dock_progress_topic = "/task_progress/dock"
            timeout_s = 120.0

    [state_descriptions.ENTER_ELEVATOR]
        state_module_name = "ropod_experiment_executor.commands.enter_elevator"
        state_class_name = "EnterElevator"
        [state_descriptions.ENTER_ELEVATOR.transitions]
            done = "DONE"
            failed = "FAILED"
        [state_descriptions.ENTER_ELEVATOR.arguments]
            area_floor = 0
            elevator_id = 4
            elevator_door_id = 88
            wait_for_elevator_action_topic = "/ropod_task_executor/WAIT_FOR_ELEVATOR"
            enter_elevator_action_topic = "/ropod_task_executor/ENTER_ELEVATOR"
            elevator_progress_topic = "/task_progress/elevator"
            timeout_s = 120.0
```

In our application, we invoke experiments specified in this manner through a remote monitoring interface, such that the status of an experiment is continuously monitored.



Future Work

- Support for *multi-level hierarchical state machines*
- Allow the specification of *concurrent states*
- Using the specification language for generating *transparent automated tests*

Open Source Repositories

[1] MAS Execution Manager. https://github.com/b-it-bots/mas_execution_manager.
[2] ROPOD Experiment Executor. https://github.com/ropod-project/ropod_experiment_executor.
[3] ROPOD Remote Monitoring. https://github.com/ropod-project/remote-monitoring.

Acknowledgement

We gratefully acknowledge the support by the b-it International Center for Information Technology. We thank Argentina Ortega and Minh Nguyen for various suggestions about the library, the b-it-bots@Home RoboCup team and the ROPOD team for its early adoption, and Dharmin Bakaraniya for actively contributing to the development.

Contact

Alex Mitrevski
Hochschule Bonn-Rhein-Sieg
Email: aleksandar.mitrevski@h-brs.de
Grantham-Allee 20
53757 Sankt Augustin
Germany

b-it Bonn-Aachen International Center for Information Technology

RoboCup2019 2-8 July 2019 Sydney | Australia

Hochschule Bonn-Rhein-Sieg University of Applied Sciences