Reusable Specification of State Machines for Rapid Robot Functionality Prototyping

Alex Mitrevski and Paul G. Plöger

Hochschule Bonn-Rhein-Sieg, Sankt Augustin, Germany
{aleksandar.mitrevski, paul.ploeger}@h-brs.de

Abstract. When developing robot functionalities, finite state machines are commonly used due to their straightforward semantics and simple implementation. State machines are also a natural implementation choice when designing robot experiments, as they generally lead to reproducible program execution. In practice, the implementation of state machines can lead to significant code repetition and may necessitate unnecessary code interaction when reparameterisation is required. In this paper, we present a small Python library that allows state machines to be specified, configured, and dynamically created using a minimal domain-specific language. We illustrate the use of the library in three different use cases - scenario definition in the context of the RoboCup@Home competition, experiment design in the context of the ROPOD project[1], as well as specification transfer between robots.

Keywords: State machines · Rapid prototyping · Experiment design.

1 Introduction

The development of robot programs requires the integration of multiple functionalities that together allow a robot to perform useful tasks. Particularly for rapid prototyping, the development process often involves the use of finite state machines, which model the program execution by a set of states and transitions between them. State machines are attractive for robot program creation due to various properties, such as the fact that they can be used to make the program execution transparent and reproducible. This is particularly important for robot experiments, which need to be designed in a manner that allows different experimenters to obtain the same results under similar experimental conditions.

The development of state machines differs depending on the programming language, but most languages have dedicated libraries for defining and creating automata[2]. 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

[1] ROPOD is an Innovation Action funded by the European Commission under grant no. 731848 within the Horizon 2020 framework program.

[2] In Python, one such example is SMACH: https://github.com/ros/executive_smach

with the implementation[3]. Mixing up the specification and implementation of a state machine has various limitations however. First of all, keeping the specification together with the implementation leads to non-reusable specifications that are committed to a specific implementation. Furthermore, states can only be made reusable if they can be reconfigured, but this means that direct code changes are needed for reconfiguration, which is particularly problematic when reusing functionalities over different robots. Having the ability to load state machines dynamically is also more reasonable in certain cases, for instance when a robot operator needs to trigger an experiment remotely[4].

In this paper, we present a Python-oriented domain-specific language for specifying state machines as well as a small Python library that allows state machines to be dynamically created. We then illustrate the use of the library in three different use cases, namely (i) a pick-and-place experiment in the context of domestic robots, (ii) a docking and elevator entering experiment for a logistic robot, and (iii) reusing the pick-and-place state machine, but redefining a particular state for a specific robot. The limitations and possible extensions of the library are then discussed.

2 Related Work

In the context of full robot autonomy, state machine-driven development has various limitations, such as the lack of flexibility [12], but nevertheless, a state machine remains an invaluable tool for rapid functionality testing. For instance, the Amazon States Language [1] is a JSON-based state machine specification language that, in addition to including basic specification constructs, allows specifying conditional transitions, parallel state execution, as well as predefined error recovery behaviours. SCXML [13] is a similar XML-based language that also defines advanced constructs for complex state machine behaviours based on Harel statecharts [5]. The main aspect that distinguishes our library is that we consider the reusability of state machines and the redefinition of states. In contrast to language-based specifications, rcommander [11] can be used for creating state machines graphically; however, the library is SMACH-specific and it also does not address the reusability aspect. Bardaro and Matteucci [2] discuss the use of the Architecture Analysis and Design Language (AADL) for robot component modelling in the context of the Robot Operating System (ROS), though the use of the presented framework with other middlewares is also considered. Similarly, Li et al. [7] apply the RoboChart state machine framework [9] for modelling robot programs, such that, just as in [2], the specification can be used for code generation. Gogolla and Vallecillo [4] apply the Unified Modelling Language

[3] Various such examples can be found in the main repository of the b-it-bots@Work RoboCup team: https://github.com/b-it-bots/mas_industrial_robotics/tree/kinetic/mir_scenarios

[4] This is for example necessary in the case of the ROPOD project, where robots need to be deployed to a hospital [8].

(UML) and the Object Constraint Language (OCL) for robot program description. The Lotos New Technology (LNT) language can also be used for specifying program behaviour and, additionally, fault diagnosis [6]. A Petri net [3] is an alternative formalism that is particularly suited for concurrent execution. All of these representations are formally rich, which can however make it difficult to apply them without dedicated training. In contrast, our state machine library is minimal and has the purpose of simplifying the process of rapid prototyping, but it should be noted that our intention is to supplement the more powerful frameworks, which remain essential in the context of architectural modelling and formal verification.

3 Use Cases

To motivate our state machine specification language and library, we will consider three different use cases: (i) definition and execution of a simple pick and place scenario in the context of the RoboCup@Home competition, (ii) experiment definition in the context of the ROPOD project, and (iii) state machine transfer between different robots.

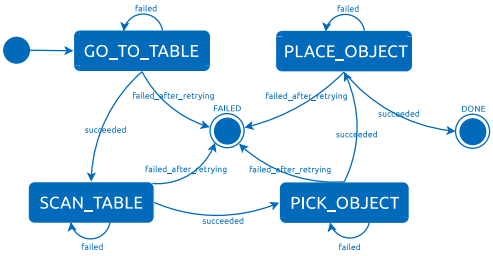
In RoboCup@Home, a common task in different scenarios is picking and placing everyday objects. In practice, both picking and placing are error-prone activities due to the variation of objects in domestic environments, which is why it can be useful to investigate a robot's performance on both tasks experimentally, for instance on a common dining table. Such an experiment could involve going to the table and scanning all objects on it, picking one of the objects, and then placing it back at a different location. This would lead to the state machine shown in Fig. 1a.

In the ROPOD project, robots need to transport hospital items, such as carts and beds, between different parts of a hospital, possibly over multiple floors. Docking to a cart and navigating into an elevator with a cart attached are particularly interesting to consider since both actions can result in execution failures. An experiment that verifies the operation of both actions would be one in which a robot first has to dock to a cart and then enter an elevator, where the assumption is that the robot docks the cart right next to the elevator. This would be represented by the state machine shown in Fig. 1b.

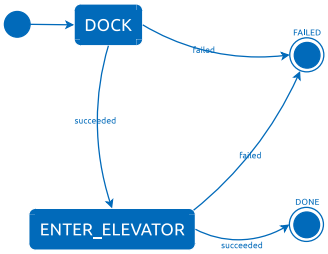
The third use case we consider is that of reusing a state machine that is either robot-independent or specified for one particular robot on a different robot. As a concrete example, we will suppose that all states in the above pick-and-place state machine can be implemented in a robot-independent manner, except for the table scanning state, which needs to be reimplemented for different robots[5]. The objective in this case is to avoid redefining the complete state machine multiple times.

In the following section, we show how we specify, create, and execute state machines such as these.

[5] For instance, some robots may require taking the manipulator out of the camera's way before the table is scanned, while others may not.



(a) A state machine for a simple pick and place experiment



(b) A state machine for a cart docking and elevator entering experiment

Fig. 1: State machines in the context of RoboCup@Home and ROPOD

4 State Machine Specification and Configuration

The general design of our Python library is illustrated in Fig. 2.

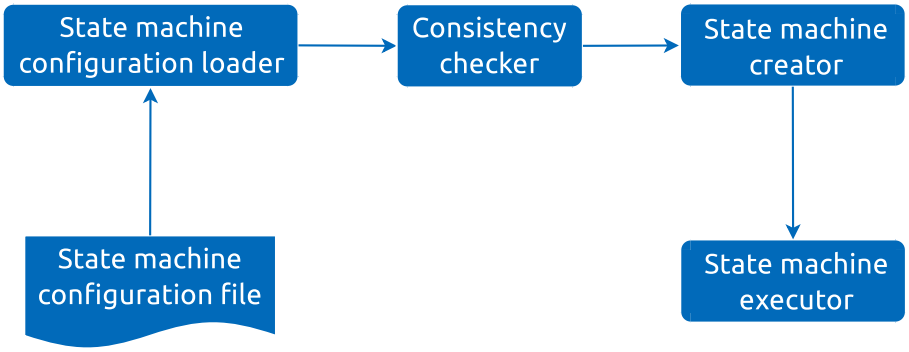


Fig. 2: Overview of the state machine specification library. A state machine is defined in a configuration file, such that state specifications include the names of the modules in which individual states are implemented. After a consistency check, the state machine is dynamically created from the specification and subsequently executed.

The state machine specification is at the core of the library, such that state machines are specified in a language embedded in TOML[6,7]. The language allows specifying the states in a state machine and the transitions between them, as well as passing arguments to the individual states and the state machine as a whole. In addition, our specification assumes that each state is implemented as

[6] https://github.com/toml-lang/toml

[7] An earlier prototype, which we still actively use, was based on a YAML-based specification.

a separate class; the names of the modules in which states are implemented and the names of the state classes are also specified in the configuration. A generic state machine specification template is shown below.

Listing 1.1: Generic state machine specification

```
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
```

By specifying state modules in the specification file, a state machine can be assembled and loaded on the fly[8]. This is accomplished in a three-step process, namely (i) a configuration loader reads the specification, (ii) a consistency check on the specification is performed (in particular, whether all defined states have been declared in the state list and whether all state transitions lead to declared states or terminal outcomes), and (iii) if the specification passes the consistency check, it is passed on to a state machine creator, which creates the state machine based on the provided configuration. The state machine can then be appropriately executed.

To create and execute state machines, we use the SMACH library, namely all states are implemented as SMACH states and the state machine creator initialises an appropriate SMACH container. It should however be noted that our library does not make any specific assumptions about the implementation of individual states or the actual state machine - other than the requirement that each state needs to be an individual class - which simplifies switching between state machine libraries without changing the state machine specification. In other words, switching between state machine libraries requires a change in the *implementation*, but not the *specification* of a state machine.

We can now illustrate how state machines can be described using our minimal description language in the context of the previously described use cases. The specification of the pick and place experiment[9] would be given as follows:

[8] In our Python library, this is achieved with the help of the *importlib* package.

[9] An implementation using the earlier YAML-based specification can be found at https://github.com/b-it-bots/mas_execution_manager; this is actively used for spec-

Listing 1.2: State machine for a simple pick and place experiment

```
sm_id = "simple_pick_and_place"
states = ["GO_TO_TABLE", "SCAN_TABLE", "PICK_OBJECT", "PLACE_OBJECT"]
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_OBJECT"
            failed = "SCAN_TABLE"
            failed_after_retrying = "FAILED"
        [state_descriptions.SCAN_TABLE.arguments]
            plane_prefix = "table"
            number_of_retries = 3

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

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

As can be seen in this specification, different arguments can be passed to the states depending on the needs; for example, the *GO_TO_TABLE* state takes a list of named locations to which the robot should go, while the *PICK_OBJECT* state receives the name of a surface from which the robot should grasp an object. This state machine also has some fault tolerance in its design, such that all states can be retried a predefined number of times before a global failure is reported.

ifying RoboCup@Home scenarios in our domestic robotics repository: https://github.com/b-it-bots/mas_domestic_robotics

The specification of the docking and elevator entering experiment in the ROPOD context is similarly shown below[10]:

Listing 1.3: State machine for a cart docking and elevator entering experiment

```
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
```

The state machine describing this experiment is slightly simpler than the one for the pick-and-place case. What is worth noting is that the states require information from an OpenStreetMap environment description [10], which the robot is using for both navigation and docking. The complete workflow in ROPOD includes triggering such experiments through a web application[11]; because of this, dynamic state machine loading is particularly useful since an operator may want to run different experiments one after the other. In addition, the separation of the specification from the implementation makes it rather simple to visualise the state machine, as shown in Fig. 3.

Finally, to accomplish state machine transfer, we define hierarchical relations between state machines, such that the robot-independent state machine represents the parent and the robot-specific one the child[12]. When redefining a state,

[10] This specification, along with various other experiment definitions, can be found at https://github.com/ropod-project/ropod_experiment_executor.

[11] https://github.com/ropod-project/remote-monitoring

[12] Multi-level hierarchies may also be beneficial, but our current implementation does not consider those.

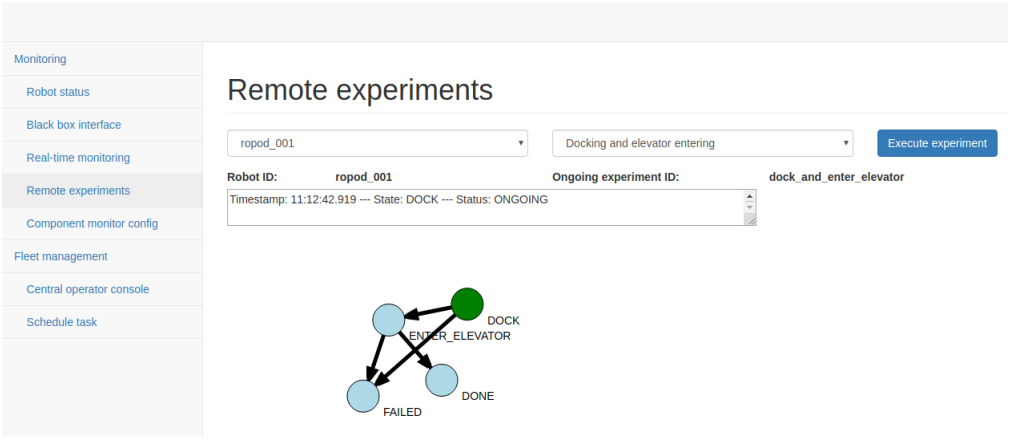


Fig. 3: Visualisation of the docking and elevator entering state machine. The green node denotes the currently active state, which is known due to the fact that the states continuously advertise their status.

in principle we only need to modify the module and class names of the robot-specific state implementation; redefining the state transitions and arguments is possible as well, but not mandatory. In addition, the global state machine parameters, such as the name and list of states, can be modified as well. A redefinition of the *SCAN_TABLE* state for a specific robot can be done as shown below.

Listing 1.4: Robot-specific state redefinition in the pick-and-place state machine

```
sm_id = "robot_specific_simple_pick_and_place"

[state_descriptions]
    [state_descriptions.SCAN_TABLE]
        state_module_name = "my_robot_perception_behaviours.perceive_planes"
        state_class_name = "PerceivePlanes"
        [state_descriptions.SCAN_TABLE.arguments]
            arm_position = "folded"
```

In the above redefinition, we specify a new implementation for the *SCAN_TABLE* state and add an additional input argument that is specific to the new state.

It should be noted that we currently define hierarchical relations between state machines by specifying the paths to the parent and child state machines as ROS node parameters; this is thus the only ROS-dependent aspect of our library.

5 Discussion and Future Work

As described above, the state machine library presented in this paper is actively used and maintained in different contexts, but there are various potential improvements that can be made. First of all, as mentioned before, the definition of hierarchical relations between state machines is currently ROS-dependent and

is performed by specifying the paths of the state machine configuration files as ROS node parameters; ideally, this should be dealt with in a ROS-independent manner, thus eliminating the dependency on ROS. A more significant issue is that the language does not allow specifying concurrent states, which are however necessary in practice since a robot should be able to perform multiple activities in parallel, such as perception and manipulation for active grasping or docking to a charging station. The ability to specify multi-level state machines hierarchies would also be a useful extension since that would further improve the reusability of state machines. Finally, an interesting use of the library would be applying the configuration language to an automated testing scenario, in particular using it as a means to generate test state machines that a robot needs to execute, potentially in a simulated environment.

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

References

1. Amazon.com: Amazon States Language, https://states-language.net/spec.html
2. Bardaro, G., Matteucci, M.: Using AADL to Model and Develop ROS-Based Robotic Application. In: 2017 1st IEEE Int. Conf. on Robotic Computing (IRC). pp. 204–207 (Apr 2017)
3. Costelha, H., Lima, P.: Modelling, analysis and execution of robotic tasks using petri nets. In: 2007 IEEE/RSJ Int. Conf. Intelligent Robots and Systems. pp. 1449–1454 (Oct 2007)
4. Gogolla, M., Vallecillo, A.: (an example for) formally modeling robot behavior with uml and ocl. In: Software Technologies: Applications and Foundations. pp. 232–246 (2018)
5. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987)
6. Hofer, B., Mateescu, R., Serwe, W., Wotawa, F.: Using LNT Formal Descriptions for Model-Based Diagnosis. In: 29th Int. Workshop Principles of Diagnosis DX'18 (2018)
7. Li, W., Miyazawa, A., Ribeiro, P., Cavalcanti, A., Woodcock, J., Timmis, J.: From Formalised State Machines to Implementations of Robotic Controllers. CoRR **abs/1702.01783** (2016)
8. Mitrevski, A., Thoduka, S., Ortega Sáinz, A., Schöbel, M., Nagel, P., Plöger, P.G., Prassler, E.: Deploying robots in everyday environments: Towards dependable and practical robotic systems. In: 29th Int. Workshop Principles of Diagnosis DX'18 (2018)
9. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling pp. 1–53 (Jan 2019)

10. Naik, L., Blumenthal, S., Huebel, N., Bruyninckx, H., Prassler, E.: Semantic mapping extension for OpenStreetMap applied to indoor robot navigation. In: IEEE Int. Conf. Robotics and Automation (ICRA) (2019)
11. Nguyen, H.: rcommander_core, http://wiki.ros.org/rcommander_core
12. Shpieva, E., Awaad, I.: Integrating Task Planning, Execution and Monitoring for a Domestic Service Robot. Information Technology **57**(2), 112–121 (Mar 2015)
13. World Wide Web Consortium (W3C): State Chart XML (SCXML): State Machine Notation for Control Abstraction, https://www.w3.org/TR/scxml/