SMACHA: An API for Rapid State Machine Assembly

Barry Ridge
Humanoid and Cognitive Robotics Laboratory
Department of Automatics, Biocybernetics and Robotics
Jožef Stefan Institute, Ljubljana, Slovenia
barry.ridge@ijs.si

ABSTRACT

Given the burgeoning complexity and diversity of both the hardware and software components of robotic systems, software libraries that use *state machines* as a basis for robot control by seamlessly connecting between low-level imperative task scripting and higher-level task planning have been in active development over the past decade or so. However, while they provide much in terms of power and flexibility, their overall task-level simplicity can often be obfuscated at the script-level by boilerplate code, intricate structure and lack of code reuse between state machine prototypes. To address these issues, we propose a code generation, templating and meta-scripting methodology for state machine assembly, as well as an accompanying application programming interface (API) for the rapid, modular development of robot control programs. The API has been developed within the ROS ecosystem to function effectively as either a front-end for concise scripting or a back-end for code generation for visual programming systems. Its capabilities are demonstrated in experiments using the Baxter robot simulator.

1. INTRODUCTION

The Robot Operating System (ROS) has, in recent years, become a popular choice of middleware for communication and control when designing robotic applications, and various packages within its ecosystem have come to the fore as being especially useful for dictating control flow. The SMACH high-level executive [3], in particular, has proven to be an exceptionally versatile and robust task-level architecture for state machine construction in ROS-based systems. It allows for the description of nested hierarchical state machines in Python in which parent container states contain child state sequences. State machines may describe lists of different possible outcomes and transitions are specified between states that depend on the outcomes in order to specify the control flow. These transitions are easily remapped across different depth levels in the hierarchy. Data may be passed between states as defined by a *userdata* object and the inputs and outputs of states may be remapped to userdata variables in order to control the flow of data.

While the ideas encapsulated by SMACH are conceptually simple, its usage still demands a significant degree of domain-specific expertise and prototyping time in order to define a functional state machine for a given robot control application. Another library that builds on the functionality of SMACH named FlexBE [5] aims at addressing this by providing a visual programming interface from which code may be generated. However, the generated code is language-specific and would therefore be brittle with respect to any significant changes to the programmatic approach.

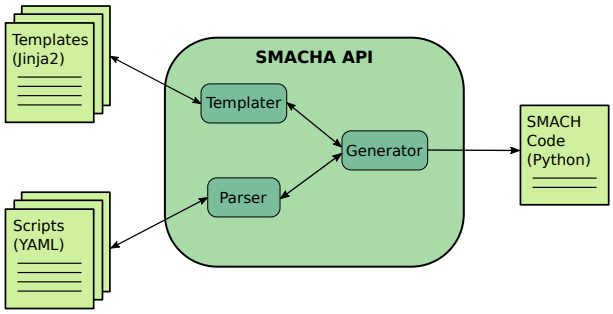

Figure 1: SMACHA API overview.

Here we present an application programming interface (API) named SMACHA[1] that aims at distilling the task-level simplicity of SMACH into compact scripts in the foreground, while retaining all of its power and flexibility in code templates and a custom code generation engine in the background. One of the major potential advantages of SMACHA is that it is is designed to be both language and framework agnostic. Although this has not yet been implemented, it would be possible, for example, to design templates to generate FlexBE code instead of SMACH code, or even state machine code written in a language other than Python, while maintaining the same scripting front-end.

2. SMACHA API OVERVIEW

The SMACHA API is composed of three main components as depicted in Fig. 1: a *parser*, a *templater* and a *generator*. The parser parses simple data-oriented scripts that describe the high-level arrangement of state machines to be constructed into operational program code by the generator and templater. We refer to this concept as *meta-scripting* and it is described below in Section 2.1. The templater retrieves and renders code templates as required by the generator in order to produce the code, and is described in Section 2.2. The generator recursively processes the parsed script and generates the final program code using the templater. It is described in Section 2.3. The relationship between the scripting and templating functionality, as well as the overall recursive code generation process, is depicted in Fig. 2.

[1]https://github.com/ReconCell/smacha

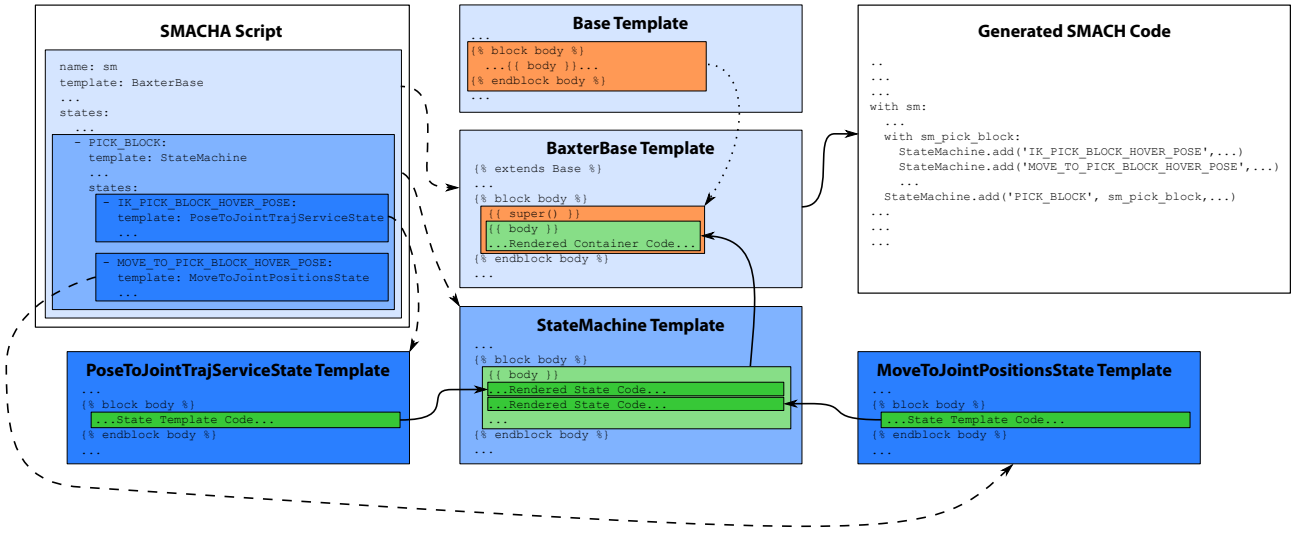

Figure 2: SMACHA recursive meta-scripting, templating and code generation pipeline example. Dashed arrows show nested state template selection from the SMACHA script and the blue shaded boxes indicate the depth level in the state hierarchy. Solid arrows and green shaded boxes show recursive template rendering flow, from child state templates at bottom-left and bottom-right, to a parent container *StateMachine* template at bottom-centre, to its parent *BaxterBase* template in the middle, to the final generated SMACH code on the right. Template inheritance is indicated by the dotted arrow and orange boxes.

2.1 Meta-Scripting

One of the core ideas behind the development of SMACHA is that state machines are essentially simple entities that can be almost entirely described via natural language constructs, perhaps augmented by some essential additional information necessary to describe how transitions should occur and how data should be passed between states. With this in mind, in order to transcribe the high-level logic of state machine description in as simple a manner as possible with a view towards offloading the more complex aspects to be processed by a code generation system working in the background we selected YAML (YAML Ain't Markup Language) as our scripting front-end [2]. YAML scripts are data-oriented and so are built around constructs such as lists and associative arrays that may be easily translated into corresponding machine code constructs and, more importantly for our purposes, can be used to represent both sequences of states and their individual data representations respectively. They can also represent data hierarchies very effectively, and are therefore well-suited to describing SMACH container states and nested state hierarchies. Thus, SMACHA scripts are YAML files that are used to describe how SMACHA should generate SMACH code. An example of a script that was written for a pick and place demonstration for the Baxter simulator can be see in Listing 1.

```
1  --- # Modular SMACHA pick and place test script for the Baxter simulator.
2  name: sm
3  template: BaxterBase
4  node_name: baxter_smach_pick_and_place_test
5  outcomes: [succeeded, aborted, preempted]
6  userdata:
7    hover_offset: [[0.0, 0.0, 0.15], [0.0, 0.0, 0.0, 1.0]]
8  states:
9    - LOAD_TABLE_MODEL:
10       template: LoadGazeboModelState
11       model_name: cafe_table
12       model_path: rospkg.RosPack().get_path('baxter_sim_examples') +
13                   '/models/cafe_table/model.sdf'
14       userdata:
15         table_model_pose_world: Pose(position=Point(x=1.0, y=0.0, z=0.0))
16         table_model_ref_frame: world
17       remapping: {pose: table_model_pose_world,
18                   reference_frame: table_model_ref_frame}
19       transitions: {succeeded: LOAD_BLOCK_MODEL}
20
21   - LOAD_BLOCK_MODEL:
22       template: LoadGazeboModelState
23       model_name: block
24       model_path: rospkg.RosPack().get_path('baxter_sim_examples') +
25                   '/models/block/model.urdf'
26       userdata:
27         block_model_pick_pose_world: [[0.6725, 0.1265, 0.7825],
28                                       [0.0, 0.0, 0.0, 1.0]]
29         block_model_pick_ref_frame: world
30         block_model_pick_pose: [[0.7, 0.15, -0.129],
31                                 [-0.02496, 0.99965, 0.00738, 0.00486]]
32         block_model_place_pose: [[0.75, 0.0, -0.129],
33                                  [-0.02496, 0.99965, 0.00738, 0.00486]]
34       remapping: {pose: block_model_pick_pose_world,
35                   reference_frame: block_model_pick_ref_frame}
36       transitions: {succeeded: MOVE_TO_START_POSITION}
37
38   - MOVE_TO_START_POSITION:
39       template: MoveToJointPositionsState
40       limb: left
41       userdata: {joint_start_positions:
42           [-0.08000, -0.99998, -1.18997, 1.94002, 0.67000, 1.03001, -0.50000]}
43       remapping: {positions: joint_start_positions}
44       transitions: {succeeded: PICK_BLOCK}
45
46   - PICK_BLOCK:
47       script: pick_block
48       remapping: {pick_pose: block_model_pick_pose,
49                   hover_offset: hover_offset}
50       transitions: {succeeded: PLACE_BLOCK}
51
52   - PLACE_BLOCK:
53       script: place_block
54       remapping: {place_pose: block_model_place_pose,
55                   hover_offset: hover_offset}
56       transitions: {succeeded: succeeded}
```

Listing 1: SMACHA pick and place demo script.

2.1.1 Base Variables

The base of a main SMACHA script file specifies the following variables: *name* (a name for the overall state machine), *template* (the name of its base template), *manifest* (an optional ROS manifest name), *node_name* (a name for its associated ROS node), *outcomes* (a list of its possible outcomes) and *states* (a list of its constituent states). Each of the states in the base script may, in turn, specify similar variables of their own, as discussed in the following.

2.1.2 States

Each state, including the base, must specify a template from which its respective code should be generated (see *e.g.* lines 3, 10, 22 and 39 of Listing 1). States may be specified as lists specifying their transition order (see *e.g.* lines 8, 9, 21, 38, 46

and 52 of Listing 1), and may also be nested as described in the SMACH documentation using appropriate combinations of template and state specifications. Possible state outcomes may be specified as a list in the base state machine and in each container state (see *e.g.* line 5 of Listing 1). Possible state transitions may be specified as an associative array in each state (see *e.g.* lines 19, 36, 44, 50 and 56 of Listing 1). Input and output remappings of user data may be specified as an associative array in each state (see *e.g.* lines 17, 34, 43, 48 and 54 of Listing 1).

2.1.3 Modularity

Modularity is achieved at the scripting level by allowing useful subroutines wrapped in container states to be saved as separate YAML script files called *sub-scripts* which can be included in a main script as states. Examples of this can be seen in lines 46–50 and 52–56 of Listing 1, where the sub-scripts "pick_block" and "place_block" are included in the main pick and place state machine script to define its sub-states. The input and output userdata keys expected by the container states in the sub-scripts may be remapped as appropriate in the main script along with their state transitions. The use of this functionality encourages low coupling and high cohesion, while allowing for extremely rapid and easily specified reuse of common patterns.

2.2 Templating

Code templating is implemented using the Jinja2 templating library [1]. Core templates are provided by default to support standard SMACH states and custom templates may be defined for particular use cases.

2.2.1 Core Templates

SMACHA provides default core templates for many of the SMACH states and containers, as well as for other useful constructs. At the time of writing, the following core templates are present and functional: *Base* (Python script skeleton), *State* (contains functionality common to all states, *e.g.* userdata specification), *StateMachine* (container), *Concurrence* (container), *ServiceState* (generic state), *SimpleActionState* (generic state), *ReadTopicState* (custom state used for reading messages from ROS topics) and *TF2ListenerState* (custom state used for reading TF2 transforms).

2.2.2 Code Generation Variables and Code Blocks

There are a number of core code generation variables and code blocks present in the core templates that enable SMACHA to produce code in the appropriate places. In most cases, a code block contains a variable of the same name within it to indicate where code from child state templates should be rendered into. The main code blocks are as follows: *base_header* (for code that must appear near the top of the program script), *defs* (for function definitions), *class_defs* (for class definitions), *main_def* (for the main function definition), *header* (for code that is to be rendered into the *header* variable the parent template), *body* (for code that is to be rendered into the *body* variable of the parent template), *footer* (for code that is to be rendered into the *footer* variable of the parent template), *execute* (for the code necessary for executing the state machine), *base_footer* (for any code that must appear near the bottom of the program script) and *main* (for the code necessary to execute the main function). The most important block for most state templates is the *body* block and its associated *body* variable, as it is where the state template should render the code necessary to add the state to the parent state machine, which will either be some container state or the base state machine itself. Note that all of the above code generation variables and code blocks may be either removed, modified or arbitrarily customized within the API for particular use cases. The code insertion order may also be specified within the API, i.e. code may be either prepended or appended to a variable. An example of how code generation variables work together with code blocks is depicted in Fig. 2.

2.2.3 Template Inheritance

Jinja2 provides powerful functionality, including the ability to extend templates via template inheritance, such that their constituent code blocks may be overridden or extended. SMACHA aims to incorporate as much of this functionality as possible, thus the core templates may be overridden or augmented by custom user-specified templates via the usual Jinja2 template inheritance mechanisms, with some caveats. This works in the usual way using the following Jinja2 variables and expressions: `{% extends "<template_name>" %}` (to inherit code blocks from the parent template specified by `<template_name>`), `{{ super() }}` (when this appears in a block, the code from the same block in the parent template as specified by `{{ extends }}` will be rendered at its position) and `{% include "<template_name>" %}` (to include all code from the template specified by `<template_name>`).

Regarding the aforementioned caveats, there is a behaviour that is specific to SMACHA that goes beyond the usual capabilities of Jinja2 and that was designed as a means of dealing with the recursive state machine processing required by this particular use case. If a state template contains blocks, but does not contain an `{{ extends }}` expression at the top of a template, it is implied that the code for the blocks will be rendered into variables and blocks with the same names as the blocks in the state template as dictated by the SMACHA script and as defined usually either by the base template or container templates. In the current implementation, only base templates use the `{% extends %}` inheritance mechanism, whereas state and container templates use the `{% include %}` mechanism to inherit code from other templates. This is partially illustrated in Fig. 2.

2.3 Code Generation

The SMACHA code generator is a custom-designed engine for recursively generating state machine code based on the scripts described in Section 2.1 and using the templates described in Section 2.2. Recursive processing was necessary given the potentially arbitrary depth levels of state machine nesting that are possible under the SMACH API. The basic operation scheme behind the code generator is thus to iterate through the data constructs of a parsed script, evaluate them based on their type, and determine whether they should be rendered as code using the appropriate templates, passed on for recursive processing, or some combination of both. When iteratively processing a script, data items that are encountered are either lists or associative arrays. When a list is encountered, it is assumed that it is a list of states and is passed on for further recursive processing. When processing an associative array, there are three main cases that

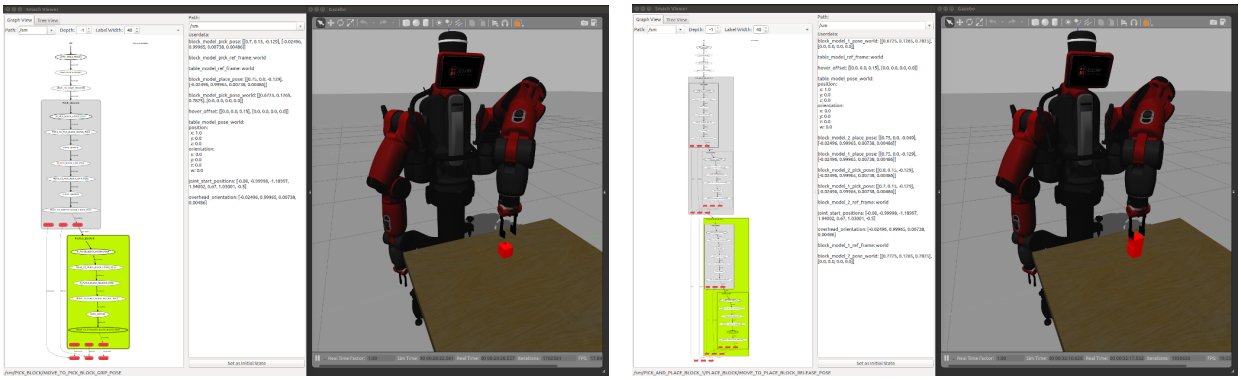

Figure 3: Pick and place (left) and stacking (right) tasks running on the Baxter simulator using SMACHA-generated code.

need to be handled separately: *container states*, *sub-script states* and *leaf states*. The recursive processing of container and leaf states is partially illustrated in Fig. 2.

3. EXPERIMENTS

For the experiments, we chose to use the Rethink Robotics Baxter robot [4] simulator which uses the Gazebo simulation system and comes equipped with extensive ROS support by default. Custom code templates were designed to facilitate the development of the necessary states required for the experiments. Two experiments were performed in total using these templates: a pick and place experiment and a block stacking experiment[2]. The first of these is a replication of the pick and place demo that comes as standard with the Baxter SDK. It was initially re-programmed from scratch in order to make use of SMACH and such that the control logic of the demo could be specified using a state machine. After that it was possible to design the necessary code templates and script the demo using SMACHA. Once the custom templates and the SMACHA script had been created for the first demo, it was possible to reuse both of them to very rapidly script the second experiment for block stacking. In both cases, it was possible to run the Python SMACH code generated by SMACHA without further modification with both experiments completing successfully.

The Baxter SMACHA package[3] includes the following custom code templates: *BaxterBase* (extends the core *Base* template), *LoadGazeboModelState* (allows allows a specified Gazebo model to be loaded into the simulator), *MoveToJointPositionsState* (moves a Baxter limb to a specified set of joint positions), *PoseToJointTrajServiceState* (uses inverse kinematics to calculate a set of joint positions from a specified end-point pose), and *GripperInterfaceState* (either opens or closes a specified gripper). In the inital states of the pick and place experiment state machine, as specified by the SMACHA script in Listing 1, a table model must be loaded into the simulator using the *LoadGazeboModelState* template, followed by a block model placed at a specified pose on the table, and the left limb of the robot must be moved to a starting position using the *MoveToJointPositionsState* template. Subsequently, the robot enters a "PICK_BLOCK" state as specified by the "pick_block" sub-script in order to pick the block from the table, followed by a "PLACE_BLOCK" state as specified by the "place_block" sub-script in order to place the block at a given placement pose. The stacking experiment initialises similarly to the pick and place experiment, only in this case, two block models are loaded instead of one, and the robot is tasked with stacking one on top of the other. This essentially involves two pick and place sequences, one for each block, so the "pick_block" and "place_block" sub-scripts used in the previous experiment are reused. The results of both experiments are depicted in Fig. 3.

[2]Video available at: https://youtu.be/WFp_keDsA6M
[3]https://github.com/abr-ijs/baxter_smacha

4. CONCLUSION

We have developed an API for the rapid assembly of state machines for modular robot control using a meta-scripting, code templating and code generation paradigm. It has been demonstrated on a simulated humanoid robot platform in two different experiments.

5. ACKNOWLEDGEMENTS

This work has been funded by the Horizon 2020 ICT-FoF Innovation Action no 680431, ReconCell (A Reconfigurable robot workCell for fast set-up of automated assembly processes in SMEs) and by the GOSTOP programme, contract no C3330-16-529000, co-financed by Slovenia and the EU under the ERDF.

6. REFERENCES

[1] Jinja2 (The Python Template Engine). `http://jinja.pocoo.org/`. (Last accessed 2017-07-24).

[2] YAML Ain't Markup Language (YAML™) Version 1.1. `http://yaml.org/spec/1.1/`. (Last accessed 2017-07-25).

[3] J. Bohren and S. Cousins. The SMACH High-Level Executive [ROS News]. *IEEE Robotics Automation Magazine*, 17(4):18–20, Dec. 2010.

[4] E. Guizzo and E. Ackerman. How rethink robotics built its new baxter robot worker. *IEEE spectrum*, page 18, 2012.

[5] P. Schillinger, S. Kohlbrecher, and O. von Stryk. Human-robot collaborative high-level control with application to rescue robotics. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2796–2802, May 2016.