Rapid State Machine Assembly for Modular Robot Control using Meta-Scripting, Templating and Code Generation

Barry Ridge[1], Timotej Gašpar[1] and Aleš Ude[1]

***Abstract*— As robotic systems have become more and more complex and difficult to manage, various software architectures, libraries and programming paradigms have been introduced aimed at high-level control and integration of their constituent parts. The Robot Operating System (ROS) has, for many, become the *de facto* software framework for communication standardisation and hardware interface abstraction, and various packages within its ecosystem have come to the fore as being reliable design choices for dictating control flow. ROS-based software packages that use *state machines* as their core methodology to bridge the gap between low-level imperative task scripting and higher-level task planning have proven particularly popular. 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. In this paper, we aim to address this deficit by proposing 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 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 an experiment using a simulated humanoid robot platform.**

I. INTRODUCTION

Recent years have seen an increase in the use of collaborative robots in industrial settings as well as for personal use. Many robot manufacturers have decided to invest into making robot programming more intuitive by adding adding functionalities such as kinesthetic guidance and improving the graphical user interfaces on teach pendants or other robot control devices. While kinesthetic teaching can be used to teach robot motions by directly guiding them by hand [1], building a sequence of actions and motions for the robot to perform still requires requires expert and manufacturer specific knowledge. To address these issues, various authors have been developing platform independent tools to ease the process of high-level programming of robot sequences, such as the State Controller Library (SC Library) and the Behavior Control Framework (BC Framework) [2].

The SMACH high-level executive [3], in particular, has proven to be an exceptionally useful and comprehensive task-level architecture for state machine construction in ROS-based systems and has seen widespread uptake. It allows for the description of nested hierarchical state machines 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.

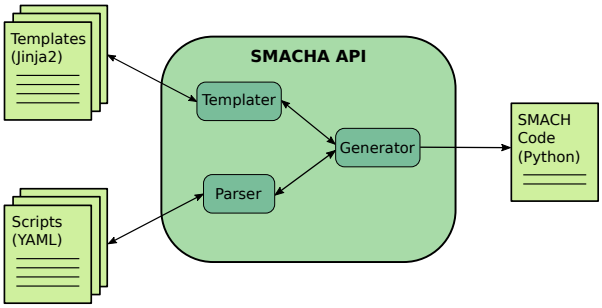


Fig. 1: SMACHA API overview.

While the ideas encapsulated by SMACH are conceptually simple, a significant degree of expertise is still required in order to program the state machines that it is capable of defining. Another library that builds on the functionality of SMACH named FlexBE [4] 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 potentially significant changes to the programmatic approach.

In this work, we have developed an application programming interface (API) named SMACHA[1] (short for "State Machine Assembler", pronounced /smæʃə/) 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 our proposed methodology is that it is is designed to be both language and framework agnostic. Although this has not yet been implemented under the current API, it would be possible, for example, to design templates to generate FlexBE state machine code instead of SMACH state machine code, or even state machine code written in a language other than Python, while maintaining the same scripting front-end.

[1]Humanoid and Cognitive Robotics Laboratory, Department of Automatics, Biocybernetics and Robotics, "Jožef Stefan" Institute, Jamova 39, 1000 Ljubljana, Slovenia {barry.ridge, timotej.gaspar, ales.ude}@ijs.si

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

A. Related Work

A systematic review performed by Domínguez *et al.* of the research in the field of object oriented code generation from state machine specifications has clearly illustrated the challenges that arise when designing frameworks for state machine design [5]. It has also shown the vast interest in this particular field. However, as the reviewed articles date from 1992 to 2010, it has lost some relevance in terms of the current available solutions. In our own literature review, we came across various tools and interfaces for intuitive robot sequence programming.

The authors of [6] present a scripting language that they developed for behaviour specification, which they called *b-script*, to describe hierarchical agent behaviours using generators. The developed language provides an interface to pre-written C++ modules which compose basic robot skills/motions. Scripts written in b-script can either be executed using an interpreter or be compiled to C++ code. Mateo *et al.* present an Android OS application called *Hammer* that provides a visual programming concept that allows a non-expert person to create robot programs. The application allows for running the designed program in simulation or for sending it to a real robot [7]. Niemüller *et al.* describe the *Fawkes* robot software framework that provides the infrastructure for running a number of plug-ins which fulfil specific tasks, where each plug-in consists of one or more threads [8]. Additionally, the authors developed a LUA scripting language plug-in for Fawkes in order to connect the Fawkes framework with the Nao robot.

CoSTAR combines a powerful GUI with grounded sensor abstractions produced by its distributed components [9]. Developed in ROS, it gives the user the ability to teach the robot kinesthetically. CoSTAR sensing capabilities expose only symbolic and qualitative information to the end user, meaning that the end user can formulate tasks in human terms. *FlexBE* provides an extensive user interface for both designing and executing various robot behaviours [4]. It also provides an interface that allows for the adjustment of behaviours during runtime. This makes it particularly useful for unstructured environments such as disaster scenes. *ROS commander - ROSCo* is a system built on top of ROS for expert users to develop, share and deploy robot behaviors [10]. ROSCo uses SMACH for saving and running the final result of the behaviour creation.

What makes SMACHA different from other related work is that it is not a standalone framework that provides a graphical interface or a library for a specific programming language. Instead, it is a tool that can be used to generate language-agnostic code from simple scripts by using templates. These templates can be easily customised to suit the needs of a particular use case. Additionally, it allows for the seamless inclusion and reuse of previously written SMACHA scripts into more complex scripts with novel behaviours.

II. SMACHA: A STATE MACHINE ASSEMBLY API

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* given its relationship to *metaprogramming* or *generative programming* [11] and it is described in more detail below in Section II-A. The templater is responsible for retrieving and rendering code templates as required by the generator in order to produce the operational state machine code. A popular library was employed for this purpose, alongside a custom rendering process developed specifically for this use case. This is described in Section II-B. The generator brings together the power of the script parser and the code templater by recursively processing the state machine script and generating the final operational state machine code. A custom engine was also developed for this purpose and is described in Section II-C. The relationship between the scripting and templating functionality, as well as the overall recursive code generation process, is depicted in Fig. 2.

A. 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 declarative constructs, similar to natural language, perhaps augmented by some essential additional information necessary to describe how transitions should occur and how data should be passed between states. Thus, a simple pick and place state machine might be partially described by a simple sentence such as "Move the arm to the start position, then pick up the block from here and place it over there." It is easy to see how such a sentence, which describes the *what* rather than the *how* of a pick and place task, might correspond to an actual state machine for its high-level robotic control, like the one depicted in Figure 3. To a large extent, this idea of simple state transitional description is already present in libraries like SMACH and FlexBE, indeed, it is SMACH itself that produces the elegant state machine charts seen in Figures 3 and 4, and FlexBE provides a similarly elegant visual programming interface for state machine design. However, in both cases, the code required to transcribe the state machine design at the scripting level is more imperative than it is declarative. A raft of boilerplate code must be in place, custom classes and functions must be either defined or imported, intricate language and library constructs must be adhered to, and as a result, the overall conceptual simplicity can sometimes get lost in the process.

With this in mind, as a first step towards developing our overall state machine assembly framework, we aimed at finding a means of transcribing 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. To achieve this, we selected YAML (YAML Ain't Markup Language) as our scripting front-end [12]. YAML scripts are data-oriented and so are built around constructs such as lists and associative arrays that may be

```yaml
--- # SMACHA block stacking demo script for the Baxter simulator.
name: sm
template: BaxterBase
node_name: baxter_smach_pick_and_place_test
outcomes: [succeeded, aborted, preempted]
userdata:
  limb: left
  hover_offset: [[0.0, 0.0, 0.15], [0.0, 0.0, 0.0, 0.0]]
states:
  - LOAD_TABLE_MODEL:
      template: LoadGazeboModelState
      model_name: cafe_table
      model_path: rospkg.RosPack().get_path('baxter_sim_examples') +
    ↪   '/models/cafe_table/model.sdf'
      userdata:
        table_pose_world: Pose(position=Point(x=1.0, y=0.0, z=0.0))
        table_ref_frame: world
      remapping: {pose: table_pose_world, reference_frame: table_ref_frame}
      transitions: {succeeded: LOAD_BLOCK_MODEL_1}

  - LOAD_BLOCK_MODEL_1:
      template: LoadGazeboModelState
      model_name: block_1
      model_path: rospkg.RosPack().get_path('baxter_sim_examples') +
    ↪   '/models/block/model.urdf'
      userdata:
        block_1_pose_world: [[0.67, 0.13, 0.78], [0.0, 0.0, 0.0, 0.0]]
        block_1_ref_frame: world
        block_1_pick_pose: [[0.7, 0.15, -0.13], [-0.02, 1.0, 0.01, 0.0]]
        block_1_place_pose: [[0.75, 0.0, -0.13], [-0.02, 1.0, 0.01, 0.0]]
      remapping: {pose: block_1_pose_world, reference_frame:
    ↪   block_1_ref_frame}
      transitions: {succeeded: LOAD_BLOCK_MODEL_2}

  - LOAD_BLOCK_MODEL_2:
      template: LoadGazeboModelState
      model_name: block_2
      model_path: rospkg.RosPack().get_path('baxter_sim_examples') +
    ↪   '/models/block/model.urdf'
      userdata:
        block_2_pose_world: [[0.77, 0.13, 0.78], [0.0, 0.0, 0.0, 0.0]]
        block_2_ref_frame: world
        block_2_pick_pose: [[0.8, 0.15, -0.13], [-0.02, 1.0, 0.01, 0.0]]
        block_2_place_pose: [[0.75, 0.0, -0.05], [-0.02, 1.0, 0.01, 0.0]]
      remapping: {pose: block_2_pose_world, reference_frame:
    ↪   block_2_ref_frame}
      transitions: {succeeded: MOVE_TO_START_POSITION}

  - MOVE_TO_START_POSITION:
      template: MoveToJointPositionsState
      userdata: {joint_start_positions: [-0.08, -1.0, -1.19, 1.94, 0.67,
    ↪   1.03, -0.50]}
      remapping: {limb: limb, positions: joint_start_positions}
      transitions: {succeeded: PICK_AND_PLACE_BLOCK_1}

  - PICK_AND_PLACE_BLOCK_1:
      template: StateMachine
      outcomes: [succeeded, aborted, preempted]
      input_keys: [limb, pick_pose, place_pose, hover_offset]
      remapping: {pick_pose: block_1_pick_pose, place_pose:
    ↪   block_1_place_pose, hover_offset: hover_offset}
      transitions: {succeeded: PICK_AND_PLACE_BLOCK_2}
      states:
        - PICK_BLOCK:
            script: pick_block
            transitions: {succeeded: PLACE_BLOCK}

        - PLACE_BLOCK:
            script: place_block
            transitions: {succeeded: succeeded}

  - PICK_AND_PLACE_BLOCK_2:
      template: StateMachine
      outcomes: [succeeded, aborted, preempted]
      input_keys: [limb, pick_pose, place_pose, hover_offset]
      remapping: {pick_pose: block_2_pick_pose, place_pose:
    ↪   block_2_place_pose, hover_offset: hover_offset}
      transitions: {succeeded: succeeded}
      states:
        - PICK_BLOCK:
            script: pick_block
            transitions: {succeeded: PLACE_BLOCK}

        - PLACE_BLOCK:
            script: place_block
            transitions: {succeeded: succeeded}
```

Listing 1: SMACHA block stacking demo script. Parameter values are rounded to two decimal places for brevity.

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. Examples of scripts that were written for demonstrations using the Baxter simulator can be see in Listings 1 and 2.

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* (optional): a ROS manifest name,
- *node_name*: a name for its associated ROS node,

```yaml
- PICK_BLOCK:
    template: StateMachine
    outcomes: [succeeded, aborted, preempted]
    input_keys: [pick_pose, hover_offset]
    transitions: {succeeded: succeeded}
    states:
      - IK_PICK_BLOCK_HOVER_POSE:
          template: PoseToJointTrajServiceState
          limb: left
          remapping: {poses: pick_pose, offsets: hover_offset, joints:
    ↪     ik_joint_response_block_pick_hover_pose}
          transitions: {succeeded: MOVE_TO_PICK_BLOCK_HOVER_POSE}

      - MOVE_TO_PICK_BLOCK_HOVER_POSE:
          template: MoveToJointPositionsState
          limb: left
          remapping: {positions: ik_joint_response_block_pick_hover_pose}
          transitions: {succeeded: OPEN_GRIPPER}

      - OPEN_GRIPPER: {template: GripperInterfaceState, limb: left, command:
    ↪     open, transitions: {succeeded: IK_PICK_BLOCK_GRIP_POSE}}

      - IK_PICK_BLOCK_GRIP_POSE:
          template: PoseToJointTrajServiceState
          limb: left
          remapping: {poses: pick_pose, joints:
    ↪     ik_joint_response_block_pick_pose}
          transitions: {succeeded: MOVE_TO_PICK_BLOCK_GRIP_POSE}

      - MOVE_TO_PICK_BLOCK_GRIP_POSE:
          template: MoveToJointPositionsState
          limb: left
          remapping: {positions: ik_joint_response_block_pick_pose}
          transitions: {succeeded: CLOSE_GRIPPER}

      - CLOSE_GRIPPER: {template: GripperInterfaceState, limb: left, command:
    ↪     close, transitions: {succeeded: MOVE_TO_GRIPPED_BLOCK_HOVER_POSE}}

      - MOVE_TO_GRIPPED_BLOCK_HOVER_POSE:
          template: MoveToJointPositionsState
          limb: left
          remapping: {positions: ik_joint_response_block_pick_hover_pose}
          transitions: {succeeded: succeeded}
```

Listing 2: SMACHA pick block sub-script.

- *outcomes*: a list of its possible outcomes,
- *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) States: Each state, including the base, must specify a template from which its respective code should be generated (see *e.g.* lines 3, 11 or 51 of Listing 1). States may be specified as lists specifying their transition order (see *e.g.* lines 9, 10, 20, 32, 44, 50 and 65 of Listing 1), and may also be nested as described in the SMACH documentation using appropriate combinations of template and state specifications (see how, *e.g.*, the "PICK_BLOCK" state of Listing 2 is specified using the StateMachine container state template and contains the states listed from line 6 onwards). 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 and line 3 of Listing 2). Possible state transitions may be specified as an associative array in each state (see *e.g.* lines 18, 55 or 59 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, 41 or 54 of Listing 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 57–59, 61–63, 72–74 and 76–78 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 contents of Listing 2 are thus included in the main pick and place state machine script in place of the "PICK_BLOCK" state, while a similar "place_block" sub-script (not listed) is included in place of the "PLACE_BLOCK" state. The input and output userdata keys expected by the container states in the sub-scripts may be remapped as appropriate in the main script

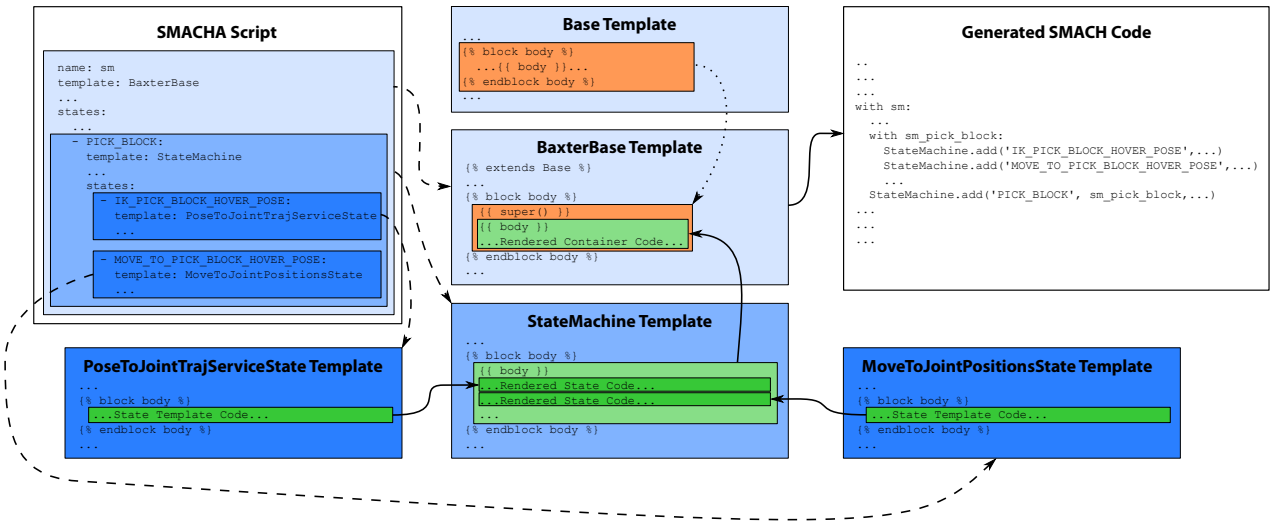


Fig. 2: SMACHA recursive meta-scripting, templating and code generation pipeline example. Dashed arrows show nested state template selection from the SMACHA script and the blue shaded boxes indicate the depth level in the state hierarchy. Solid arrows and green shaded boxes show recursive template rendering 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.

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, as shall be demonstrated in the experiments of Section III.

B. Templating

Code templating is implemented under the SMACHA API using the Jinja2 templating library [13] coupled with some custom modifications to its usual rendering behaviour, as detailed below, in order to achieve some of the functionality required by our proposed state machine code generation methodology. A number of core templates are provided by default to support standard SMACH states and common design patterns. Custom templates may be defined for particular use cases (see Section III for examples).

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, where the number of lines of code employed in each case is listed in parentheses:

- *Base*: renders a Python SMACH script skeleton (53).
- *State*: contains functionality common to all states, *e.g.* userdata specification (5).
- *StateMachine*: renders a SMACH *StateMachine* container for nested state machines (18).
- *Concurrence*: renders a SMACH *Concurrence* container for parallel state machines (20).
- *ServiceState*: renders a SMACH *ServiceState* for use with ROS services (37).
- *SimpleActionState*: renders a SMACH *SimpleActionState* for use with ROS action servers (41).
- *TF2ListenerState*: used for reading TF2 transforms (57).
- *Utils*: contains common utility code (21).

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 the code generation engine 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 where the state

template code from the `{% body %}` blocks of both the *PoseToJointTrajServiceState* and *MoveToJointPositionsState* templates are appended to the `{{ body }}` variable of their parent *StateMachine* template.

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>" %}`: When this expression appears at the top of a template, the template will inherit code blocks from the parent template specified by `<template_name>`.
- `{{ super() }}`: When this expression appears inside a block, the code from the same block in the parent template as specified by `{{ extends }}` will be rendered at its position.
- `{% include "<template_name>" %}`: When this expression appears at the top of a template, the template will 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.

C. Code Generation

The SMACHA code generator is a custom-designed engine for recursively generating state machine code based on the scripts described in Section II-A and using the templates described in Section II-B. 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 need to be handled separately:

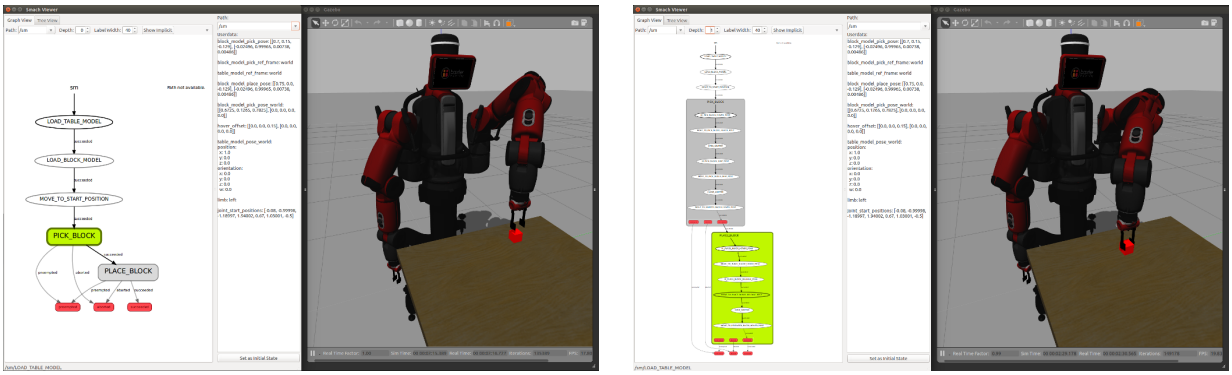
- *Container States*: if the associative array contains a "states" key, then it is assumed that it represents a container state. Its respective data and sub-states are recursively processed and once the results are returned, they are used to subsequently render the container state template.
- *Sub-Script States*: if the associative array contains a "script" key, then it is assumed that it represents a state that includes a sub-script. The sub-script is found and parsed by the script parser and any remapped input and output userdata keys or transitions are replaced in the parsed script before it is recursively processed.
- *Leaf States*: otherwise, the associative array is assumed to be a leaf state. Its template is rendered by the templater and its rendered code and variables are returned to the parent state for further use at higher processing levels.

The recursive processing of container and leaf states is partially illustrated in Fig. 2, where the *IK_PICK_BLOCK_HOVER_POSE* and *MOVE_TO_PICK_BLOCK_HOVER_POSE* states use *PoseToJointTrajServiceState* and *MoveToJointPositionsState* templates respectively to render code into a parent container *PICK_BLOCK* state as defined by the *StateMachine* template, which itself in turn is rendered into the base state machine as specified by a *BaxterBase* template.

III. EXPERIMENTS

For the experiments presented in this section, we chose to use the Rethink Robotics Baxter robot [14] 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. These are detailed below in Section III-A.1. Three experiments were performed in total using these templates: a pick and place experiment, a block stacking experiment and a dual-arm block stacking experiment, all of which are described below in more detail in Sections III-C, III-D and III-E respectively[2]. The first of these, the pick and place experiment, 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 and third experiments for block stacking and dual-arm block stacking. In all cases, it was possible to run the Python SMACH code generated by SMACHA without further modification with all three experiments completing successfully.

[2]Video available at: `https://youtu.be/KRjY0bd4dLg`



(a) In the "PICK_BLOCK" state.

(b) In the "MOVE_TO_PLACE_BLOCK_RELEASE_POSE" state of the parent "PLACE_BLOCK" state.

Fig. 3: A pick and place task running on the Baxter simulator using SMACHA-generated code.

A. Baxter Simulator

1) Custom Templates: The Baxter SMACHA package[3] currently contains the following custom code templates, where the number of lines of code employed in each case is listed in parentheses:

- *BaxterBase*: this extends the core *Base* template to add some necessary imports and code for starting the Baxter robot (35).
- *LoadGazeboModelState*: this state allows allows a specified Gazebo model to be loaded into the simulator at a specified pose in a specified reference frame (114).
- *MoveToJointPositionsState*: this state moves a specified Baxter limb to a specified set of joint positions using the Baxter interface (77).
- *FollowJointTrajActionState*: this state passes a specified set of joint configurations for a specified Baxter limb to the follow joint trajectory ROS action server provided by the Baxter SDK (19).
- *PoseToJointTrajServiceState*: this state uses the inverse kinematics ROS service provided by the Baxter SDK to calculate a set of joint positions from a specified end-point pose for a specified Baxter limb (177).
- *ReadEndpointPoseState*: this state reads the current end-point pose of a specified Baxter limb using the Baxter interface (63).
- *GripperInterfaceState*: this state either opens or closes the gripper of the specified limb using the Baxter interface (63).

B. Sub-Scripts

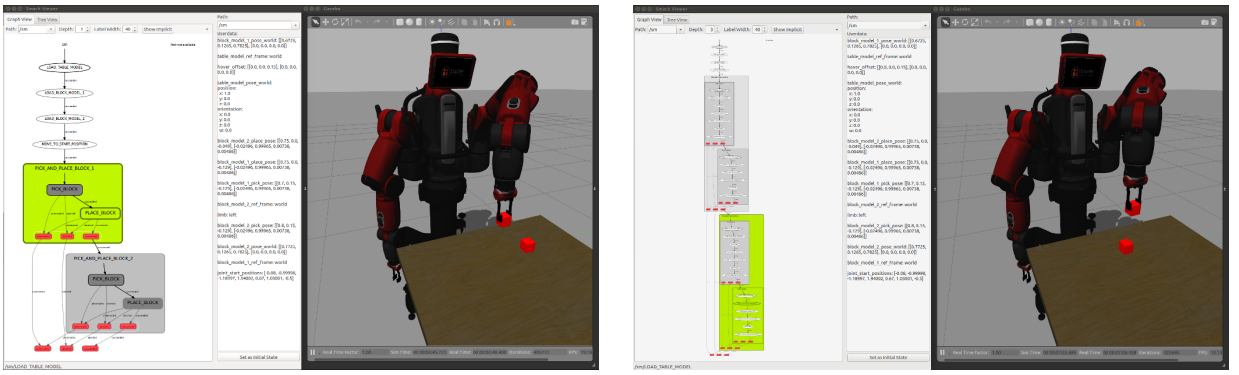
1) The "pick_block" Sub-Script: This sub-script, detailed in Listing 2, describes a series of states designed to instruct a Baxter limb to pick up a block object from a given pose. It specifies a container state that starts in a "IK_PICK_BLOCK_HOVER_POSE" state based on the *PoseToJointTrajServiceState* template which uses inverse kinematics to calculate appropriate joint positions for its left limb given the block pose and a hover offset such that the calculated joint positions leave the limb end-point hovering just above the block object (see lines 7–11 of Listing 2). The block pose is specified in the base reference frame due to the fact that the inverse kinematics solver requires such input, but the pose could just as easily be specified in the world frame with the addition of a *TF2ListenerState* that would transform the pose appropriately. Next, the robot enters into a "MOVE_TO_PICK_BLOCK_HOVER_POSE" state based on the *MoveToJointPositionsState* template using the calculated joint positions (lines 13–17), before opening the left gripper using a *GripperInterfaceState* template (line 19). After that, it enters into consecutive inverse kinematics and joint movement states once again in order to calculate and move to the block position (lines 21–31), before using another gripper interface state to close the gripper around the block object (line 33). Finally, it enters one last joint movement state in order to return to the previously calculated joint positions based on the hover pose (lines 35–39). The length of the "pick_block" sub-script was 35 lines of code.

2) The "place_block" Sub-Script: This sub-script is designed to instruct Baxter to place a block, assumed to be currently grasped by a given limb, at a given pose in the base frame. It first uses a *PoseToJointTrajServiceState* state to calculate the joint positions required to place the limb in a hover position above the desired block place pose, before using a *MoveToJointPositionsState* joint motion state to perform the actual motion. It follows these with another two states of the same type in order to move the grasped block to the placement pose. Finally, it uses a *GripperInterfaceState* to open the gripper in order to release the block, before using one last *MoveToJointPositionsState* to move the limb back to the previously calculated hover pose. The length of the "place_block" sub-script was 31 lines of code.

C. Pick and Place Experiment

In the inital states of the pick and place experiment state machine, a table model must be loaded into the simulator using the *LoadGazeboModelState* template, followed by a

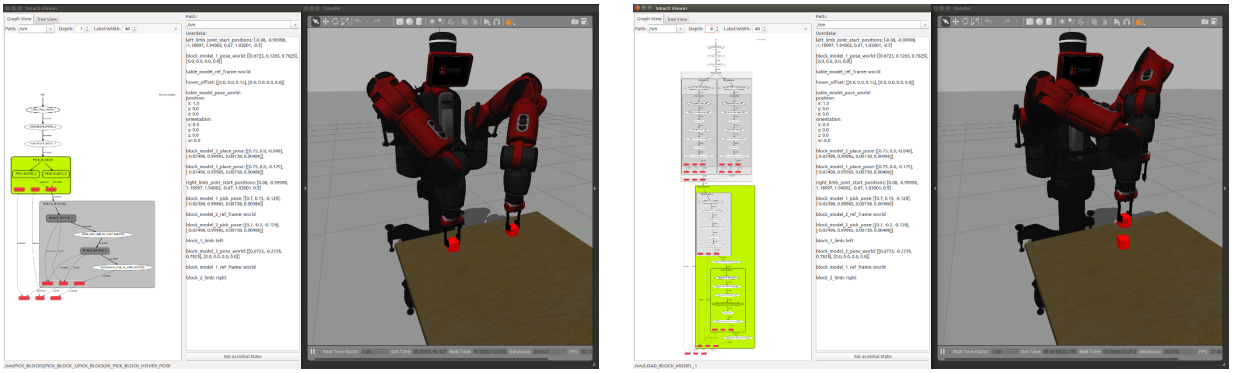
[3] https://github.com/abr-ijs/baxter_smacha



(a) In the "PLACE_BLOCK" state of the parent "PICK_AND_PLACE_BLOCK_1" state.

(b) In the "MOVE_TO_PLACE_BLOCK_HOVER_POSE" state of the parent "PLACE_BLOCK" state of the parent "PICK_AND_PLACE_BLOCK_2" state.

Fig. 4: A stacking task running on the Baxter simulator using SMACHA-generated code.



(a) In the "PICK_BLOCK_1" and "PICK_BLOCK_2" states of the parent "PICK_BLOCKS" concurrent state.

(b) In the "MOVE_TO_PLACE_BLOCK_RELEASE_POSE" state of the parent "PLACE_BLOCK_2" state of the parent "STACK_BLOCKS" state.

Fig. 5: A dual-arm stacking task running on the Baxter simulator using SMACHA-generated code.

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 length of the SMACHA script for the pick and place experiment was 45 lines of code. The results of the generated code for this experiment being run on the Baxter simulator are depicted in Fig. 3 along with visualisations of the associated state machine at various levels of introspection.

D. Stacking Experiment

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. The required SMACHA script is depicted in Listing 1. 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 here in two separate container states, "PICK_AND_PLACE_BLOCK_1" and "PICK_AND_PLACE_BLOCK_2" respectively. As is evident, the script is quite compact in its specification with a length of 71 lines of code, yet nevertheless results in the generation of quite an elaborate state machine, as is visualised in Fig. 4.

E. Dual-Arm Stacking Experiment

In the dual-arm stacking experiment, the objective was the same as in the stacking experiment, but both of the Baxter arms were employed to pick up each of the blocks simultaneously before stacking one on top of the other. In order to achieve this, a *Concurrence* template was employed (see Section II-B) such that two state machine sequences could be run in parallel, namely the "PICK_BLOCK_1" and "PICK_BLOCK_2" states for the left and right arm picking actions respectively. This is depicted in Fig. 5a. A regular *StateMachine* container was used to specify the stacking sequence, employing the "place_block" sub-script as before,

TABLE I: Code overhead progression from templates, sub-scripts and scripts versus generated code in files/lines.

	Templates	Sub-Scripts	Scripts	Total	Gen.
Pick & Place	5/466	2/66	1/45	8/577	1/372
Stacking	0/0	0/0	1/71	1/71	1/478
Dual-Arm	0/0	0/0	1/102	1/102	1/513

as depicted in Fig. 5b. The dual-arm stacking script was not much longer than the stacking script, containing 102 lines of code.

F. Results

Table I details the code overhead investment progression from templates, sub-scripts and scripts versus generated code when moving between the experiments in terms of the number of files and lines of code required for their implementation. It should be noted that the data presented in the table does not include the core SMACHA templates, which are assumed to be available for all usecases. As can be clearly seen, after the initial investment in the templates and sub-scripts, the amount of SMACHA script code required to generate the significant quantities of Python code necessary for running the subsequent experiments is minimal. Thus, state machine prototypes in the second two experiments were implemented relatively rapidly. While it could be argued that similar could be achieved by simply designing a bespoke Python library directly, this would forgo the significant declarative scripting advantages of SMACHA. Moreover, it would fail to take into account the potential advantages afforded by the SMACHA templating paradigm which, although yet untested, would allow for templates to be created for arbitrary programming languages or state machine frameworks, e.g. FlexBE instead of SMACH, without altering the declarative scripting interface.

IV. CONCLUSION AND FUTURE WORK

In conclusion, we have developed both a methodology and an API for the rapid assembly of state machines for modular robot control using a meta-scripting, code templating and code generation paradigm. The API has been demonstrated to function as described on a simulated humanoid robot platform in three different experiments. One of the main advantages of our contribution is that both code templates and sub-scripts may be reused as required depending on the particular robotic use case and the task at hand– once certain such basic templates and sub-scripts are in place, it is trivial to design a new SMACHA state machine script in order to accomplish a novel task. A second major potential advantage of our proposed methodology is that it is both language and framework agnostic– templates could be written for any language or framework whilst maintaining a common front-end for high-level state machine scripting.

With regard to future work, we are eager to write FlexBE code templates in order to prove the efficacy of our API with respect to the claim of language and framework agnosticism. More applicatively, we aim to incorporate SMACHA into the development of state machine-based visual programming control software for a rapidly reconfigurable robotic work-cell for small businesses in industry. We also aim to use SMACHA in order to program state machines for a real humanoid robot in future work, in particular, the Talos robot.

ACKNOWLEDGMENT

This work has been funded by the Horizon 2020 FoF Innovation Action no. 680431 ReconCell project and by the GOSTOP programme, contract no. C3330-16-529000, co-financed by Slovenia and the EU under the ERDF.

REFERENCES

[1] A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration," in *Springer Handbook of Robotics*. Springer, 2008, pp. 1371–1394.

[2] P. Allgeuer and S. Behnke, "Hierarchical and state-based architectures for robot behavior planning and control," in *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE-RAS Int. Conf. on Humanoid Robots, Atlanta, USA*, 2013, pp. 3–5.

[3] J. Bohren and S. Cousins, "The SMACH High-Level Executive [ROS News]," *IEEE Robotics Automation Magazine*, vol. 17, no. 4, pp. 18–20, Dec. 2010.

[4] 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)*, May 2016, pp. 2796–2802.

[5] E. Domínguez, B. Pérez, Á. L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," *Information and Software Technology*, vol. 54, no. 10, pp. 1045–1066, Oct. 2012.

[6] T. J. de Haas, T. Laue, and T. Röfer, "A scripting-based approach to robot behavior engineering using hierarchical generators," in *2012 IEEE International Conference on Robotics and Automation*, May 2012, pp. 4736–4741.

[7] C. Mateo, A. Brunete, E. Gambao, and M. Hernando, "Hammer: An Android based application for end-user industrial robot programming," in *2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, Sept. 2014, pp. 1–6.

[8] T. Niemüller, A. Ferrein, and G. Lakemeyer, "A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao." in *RoboCup*. Springer, 2009, pp. 240–251.

[9] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "CoSTAR: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 564–571.

[10] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp, "ROS commander (ROSCo): Behavior creation for home robots," in *2013 IEEE International Conference on Robotics and Automation (ICRA)*, May 2013, pp. 467–474.

[11] K. Czarnecki, U. W. Eisenecker, and K. Czarnecki, *Generative Programming: Methods, Tools, and Applications*. Addison Wesley Reading, 2000, vol. 16.

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

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

[14] E. Guizzo and E. Ackerman, "How Rethink Robotics Built Its New Baxter Robot Worker," *IEEE Spectrum*, p. 18, 2012.