Module name: SMACHA

- **Main functionalities**:

SMACHA is a meta-scripting, templating, and code generation engine for rapid prototyping of ***ROS SMACH*** state machines.

SMACH is an exceptionally useful and comprehensive task-level architecture for state machine construction in *ROS*-based robot control systems. However, while it provides much in terms of power and flexibility, its 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.

SMACHA (short for "State Machine Assembler", pronounced "smasha") aims at distilling the task-level simplicity of *SMACH* into compact *YAML* scripts in the foreground, while retaining all of its power and flexibility in *Jinja2*-based templates and a custom code generation engine in the background. Thus *SMACHA* does not aim to replace *SMACH*, but to augment it.

- **Technical specifications:**

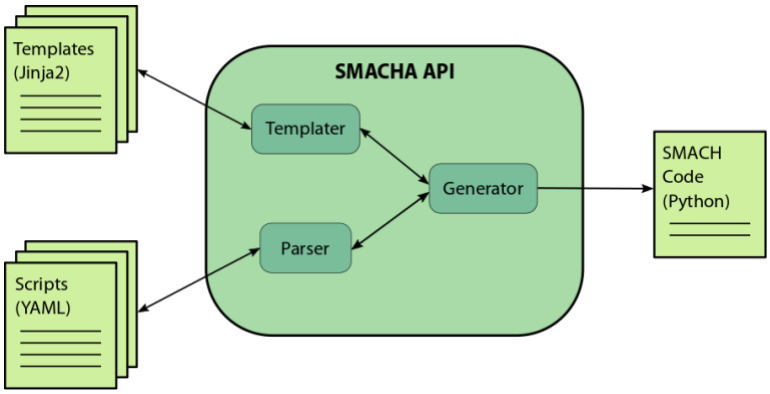

Figure 1: SMACHA API overview

Figure 1 contains a high-level illustration of the main components of the *SMACHA API*.

***SMACHA* scripts** are YAML files that describe how *SMACHA* should generate SMACH code. They are parsed by the *Parser module* and direct the *Generator module* on how to combine *SMACHA templates* using the *Templater module*.

Some of the benefits of meta-scripting in this way include:

- **quick, at-a-glance overviews** of state machine program intent,
- **easy script manipulation, reuse and restructuring** - *SMACHA* provides various utilities to automate some common script manipulation tasks, e.g.
    - the Contain Tool for automatic containerization of state sequences,
    - the Extract Tool for automatic conversion of commonly used container states to reusable sub-scripts,
- **streamlined *ROS* integration**, e.g.

- loading of scripts onto the *ROS* parameter server and
- executing them with service calls.

***SMACHA* templates** are used to specify how *SMACH Python* code should be generated from *SMACHA* scripts. They are effectively code skeletons with placeholder variables that are fleshed out with the contents of the scripts via the recursive code generation process. The *SMACHA API* renders these templates using the Jinja2 library, a powerful template engine for *Python*, coupled with some custom modifications to its usual rendering behaviour in order to suit this particular use case. They are filled out by the Templater module as directed by the Generator module and as determined by the structure of the SMACHA scripts parsed by the Parser module.

The use of templates comes with its own additional benefits:

- **intricate boilerplate code can be automatically filled**,
- **increased code reusability and modularity**, e.g.
  - common design patterns can be easily turned into state templates,
  - template macros make it easy to repeat common patterns between templates,
  - template inheritance helps avoid repetitive code blocks,
- **templates could be designed for use with frameworks other than *SMACH* and *Python***
  - although this has not yet been tested, in theory templates could be written for use with other frameworks, e.g. FlexBE, CoSTAR, and others.

The SMACHA Code Generator recursively processes SMACHA Scripts parsed by The Parser Module, manages the use of SMACHA Templates that are filled out by The Templater Module, and renders the final result to executable *Python SMACH* code. At its core, the code generator is a *Python API*, but a *ROS wrapper* is also provided that allows the generator to be launched as a *ROS* node that exposes services and generates *SMACH* code over *ROS topics*.

Thus, in order to install and run *SMACHA*, a *ROS*-based robot control system is required, as well as a *Linux* PC with the various prerequisite software libraries installed, e.g. *YAML, Jinja2, SMACH*, etc.

- **Inputs and outputs:**

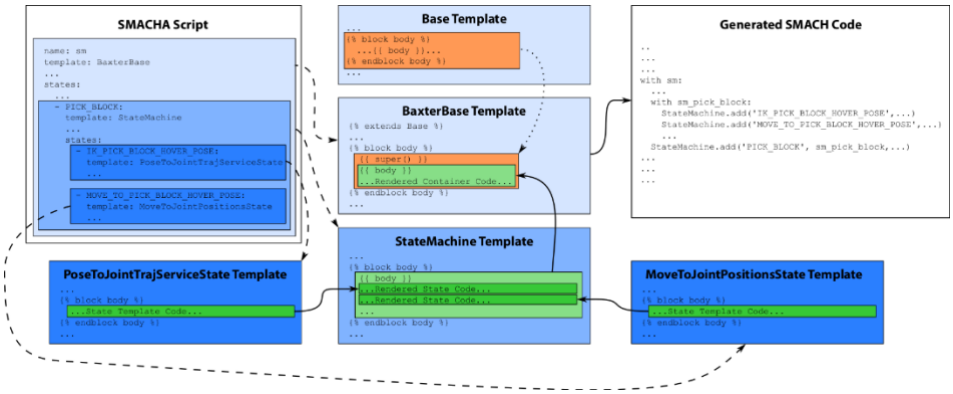

Figure 2: SMACHA recursive meta-scripting, templating and code generation pipeline example.

SMACHA requires the following inputs and outputs for operation:

- **Inputs:**
  - a template library, written in *Jinja2* format to produce *SMACH Python* code,
  - *YAML* scripts and/or sub-scripts
- **Outputs:**
  - a generated *SMACH Python* script.

Figure 2 illustrates how the scripts and templates are used in combination during the recursive code generation process in order to produce the final generated *SMACH* executable code.

Table 1: Example SMACHA YAML script.

```
--- # Nesting State Machines Tutorial SMACHA script.
name: sm_top
template: Base
manifest: smacha
node_name: smach_example_state_machine
outcomes: [outcome5]
states:
  - BAS: {template: Bas, transitions: {outcome3: SUB}}
  - SUB:
      template: StateMachine
      outcomes: [outcome4]
      transitions: {outcome4: outcome5}
      states:
        - FOO: {template: Foo, transitions: {outcome1: BAR, outcome2: outcome4}}
        - BAR: {template: Bar, transitions: {outcome1: FOO}}
```

Table 1 shows some example *SMACHA YAML* code for the "Nesting State Machines" example from the SMACH Tutorials.

- **Interface specification:**

The end-user will write *SMACHA YAML* scripts that are compiled into complex robot tasks. Expert users or developers may also write application specific *SMACHA* templates in order to provide additional functionality beyond the default templates.

- **Formats and standards used**:

YAML, Jinja2, Python, ROS MoveIt!, ROS topics, ROS services, ROS action servers, ROS bags, etc.

- **Availability:**

Open source, published as an official *ROS package*: http://wiki.ros.org/smacha

- **Application scenarios:**

Develop robot assembly sequences, robot data gathering sequences, etc.

- **Offered for internal / external use**

The *SMACHA packages* are available for both internal and external use.