Improvement of the TARDIS configuration system

Proposal by **Fotis Tsamis (ftsamis)**

Overview

TARDIS is a Monte Carlo radiative-transfer spectral synthesis code for 1D models of supernova ejecta. In order for TARDIS to carry out the desired calculations it requires as input a configuration file expressed in YAML which contains all the necessary information.

Currently, TARDIS uses the pyyaml library to parse the configuration file and a complex, custom-made validator to validate it. The issues with the configuration system in its current form are many:

- Many tasks that should be done in the parsing phase (e.g. converting to astropy Quantity objects) are done in the validator.
- The validator is practically reinventing the wheel, since it basically implements many simple or advanced concepts which are already implemented in existing validating libraries.
- The custom-made validator comes with a custom schema which adds nothing to already existing well-defined and broadly-used schemas.
- The code which processes the configuration after validation for the construction of the final Configuration object from different models is unmodular, it has many parts which are never used, or don't have any effect and it is officially described as "a perfect example for difficult to maintain code".

My project proposal is to improve the way the configuration system works by:

- Taking full advantage of the capabilities of pyyaml, by doing all the required type conversions in the parsing phase.
- Replacing both the validator and the schema with an existing, broadly-used library and format and using the library to the maximum extent in order to cover all the needs.
- Replacing the after-parsing code which processes different input models with a well-defined, easily extensible (with more input model formats) system.
- Documenting and testing all of the above.

These changes will be made with retaining as much compatibility with the current configuration syntax as possible in mind. This means that while the end-users of TARDIS may not notice any change, TARDIS' codebase will greatly benefit from replacing complex, unmaintainable, and undocumented code with modular, well-defined and documented code.

Detailed Approach

Each section below represents a sub-problem that I will be solving with this project.

Parsing the YAML configuration file

A valid TARDIS configuration, except from the default YAML types (int, float, etc), may also contain a few custom types. These are:

1. **quantity**, which is expressed as a value followed by a space and then a unit, and is ultimately converted to an astropy Quantity object
2. **quantity_range_sampled**, which is expressed as a mapping of (start, stop, num) key-values, and is ultimately converted to a list of [start Quantity, stop Quantity, integer] values
3. **container-property**, which is a container of other types, and does not need any extraordinary handling at the parsing phase.

All of the above types are currently handled in the validator. This is a problem firstly because the validator will be replaced in its entirety and secondly because, theoretically, a validator's task is to validate, not to do extra conversions.

My solution to this sub-problem will be to utilize and extend the custom YAML Loader I have written in my PR #515 which already handles the aforementioned custom types by defining a yaml constructor and an implicit resolver to parse quantities, and by replacing the default yaml mapping constructor to parse quantity ranges and return them as lists instead of dicts. For the third type, container-property, nothing needs to be done in the parsing phase.

The above solution for this sub-problem is proposed with keeping the configuration file syntax unchanged as a goal. It is also modular and elegant as it subclasses the default pyyaml.Loader and changes its behaviour as needed.

Validating the YAML configuration file

As mentioned in the Overview, one of the goals of this proposal is to completely remove the custom validator currently used and assign the task of validation to a broadly-used third-party library. One such library for validating a parsed YAML document against a schema is JSON Schema. Since JSON and YAML are fully compatible, JSON Schema is able to validate parsed YAML documents and the schema may be defined in YAML too, which means that JSON format is not a requirement.

JSON Schema has a Python implementation which I am going to use for this sub-problem. The main workload at this stage, will be to make the appropriate changes in the JSON Schema validator in order for TARDIS' custom types (quantity, quantity_range_sampled, container-property) to be recognized as valid types.

More specifically, the validator can accept custom types as parameters and the Python objects they should map to (e.g. `Draft4Validator(types={"quantity": Quantity})`. Care will be taken for supporting the quantity_range_sampled type as properties of this type must be a list containing 3 elements of a specific type and order (`[Quantity, Quantity, int]`). To support this type, adding it to the "types" parameter of the validator is not enough, since this will only check if it is of type list. Therefore, a custom validator will be required for this type to make sure all of the required constraints are met.

For the container-type type, most of the work needs to be done in the sub-problem of converting the current configuration schema to JSON Schema format, which is described below.

Converting the configuration schema

The current custom-made schema must be re-written with valid JSON Schema syntax (in YAML format) for the jsonschema library to be able to read it. Fortunately, JSON Schema's syntax is powerful enough to allow us to cover all of our current schema needs, with the most tricky one being the container-type type.

The minimum needed features that are used by the current TARDIS schema and therefore must be supported by the new schema are:

1. A property_type field which defines the valid type of each property.
2. A mandatory field which defines if a property is required or optional.
3. A default field which defines the default value of a property.
4. A help field which is a human-readable description of a property.
5. Custom property type definition by using the container-property type. The custom type name is then defined inside the container-declaration object by using a '_' prefix and a list of what properties this custom type should provide. Optional properties may be defined by using a '+' prefix and a list of the optional properties.

The equivalents of JSON Schema to those features are:

1. A type field.
2. A required field, which is a list of all the required properties.
3. A default field. Note that the python implementation of jsonschema ignores this field by default, and special care will be taken to support setting default values.
4. A description field.
5. Subschemas for custom types, which will also define what properties should be provided for each type. Taking advantage of the more advanced features of jsonschema like *$ref*, and *oneOf* fields we can define subschemas, one for each custom type, and require that a property must validate against one of our custom schemas by using the *oneOf* field and references to these schemas (*$ref* field).

Lastly, JSON Schema has many more features which may benefit TARDIS in the future such as: min/max constraints, regular expression pattern matching and others.

Reworking the input model handling

After the configuration file is parsed and validated, TARDIS does some post-processing which may include reading data from additional files that provide custom abundance and/or density profiles. Because that takes place in one long list of python commands with a lot of case checking, it is not easy to extend this part of the configuration system. There are also many functions that were doing part of this task but are not used anymore, as well as lines of code which do absolutely nothing (an example follows).

```
if plasma_section['helium_treatment'] == 'recomb-nlte':
   validated_config_dict['plasma']['helium_treatment'] == 'recomb-nlte'
else:
```

```
    validated_config_dict['plasma']['helium_treatment'] == 'dilute-lte'
```

A good first step for solving that will be to cleanup the config_reader module from unused code. Then, I plan to patternize the rest of the useful code (both in config_reader and in model_reader) and create a structure of handlers, each for a distinct task. The goal is to be able to connect a section or a type in the configuration file with a post-processing handler.

This may also be achieved directly in the configuration file by using YAML tags, which would allow us to specify in the file itself which handler we want to connect with what property or type, but that is probably not needed since the handler connection to a property will most likely be something constant, and not the user's choice. This would also introduce a new bit of syntax (tags), so it would break compatibility with older versions of TARDIS.

Deliverables

Below is a list of measurable and specific goals which consist solutions to the problems described in the Detailed Approach section.

1. The enhanced YAML parser implementation.
2. The new schema written in YAML and following the JSON Schema syntax.
3. The customized validator implementation based on the jsonschema library.
4. The reworked config_reader module.
5. The updated documentation for all of the above.

Tentative Timeline

This is an indicative schedule of how much time I plan to spend on each part of this project. I plan to work every day for 8 or more hours. In the case I miss a day for any unpredictable reason, my backup plan is to exchange the missed day with a day of the weekend. Other than my term's final exams which usually start on June 15 and will set me back around 5 nonconsecutive days, I don't have any planned commitments.

May 23 - May 31
Do all the proposed modifications on the the YAML parser and test it. *(Details)*
Deliverable: The customized YAML parser implementation.

June 1 - June 15
Transcode the current custom schema to JSON Schema syntax formatted in YAML. *(Details)*
Deliverable: The new schema.

June 16 - June 30
Create a validator based on jsonschema supporting all the custom types used by TARDIS. *(Details)*
Deliverable: The validator implementation.

July 1 - July 19
Remove everything that is unused/legacy code and design a structure for the input model handling. *(Details)*

July 20 - August 4
Implement the input model handling structure and migrate the current functionality. *(Details)*
Deliverable: The reworked config_reader module.

August 5 - August 14
Update the documentation to reflect the changes made and fix any issues found in the code.
Deliverable: The updated documentation.

August 15 - August 23
Final cleanup and submission of the code.

About me

Personal Information

Name: Fotis Tsamis
E-mail: ftsamis@gmail.com
Github username: ftsamis

Brief background

I am an undergraduate student at the Department of Informatics and Telematics of Harokopio University of Athens. I started learning programming as a hobby at the age of 14, with C being my first language. At about the same time I was introduced to the universe of Linux and Free Software and, not much later, amazed by its ideals, I started contributing, first by translating GNOME, OpenOffice and Ubuntu to Greek, and then by creating two projects. My first functional GUI project was CPU-G, a GPL-licensed Python/GTK+ application for showing a system's hardware information. Then, in 2010 I co-developed two fully-featured computer lab administration and management applications (sch-scripts and Epoptes) to help Greek schools progressively transition to Linux and FOSS. They are now both used with great success in schools, with Epoptes translated in over 37 languages.