Implementation of SBML-JSON converter and SBML-JSON scheme

Proposal for Google Summer of Code, 2020

Student

Name: Hemant Yadav
E-mail: hemant@cs.iitr.ac.in
Github: Hemant27031999
Phone: +91-8708786527
Postal Address: BG-007, Rajeev Bhawan, IIT Roorkee, Roorkee, India
HomePage: https://hemant27031999.github.io/
University: Indian Institute of Technology, Roorkee

Mentoring Organization

National Resource in Network Biology (NRNB)

Mentors

Dr. Matthias König (Humboldt-University Berlin, Germany, konigmatt@googlemail.com)
JProf. Andreas Dräger (University of Tübingen, Germany, draeger@informatik.uni-tuebingen.de)

About the Project

The System Biology Markup Language (SBML)[1] is the *de facto* standard for representation and exchange of mathematical models of biological systems. SBML can represent many different classes of phenomena in biology, including metabolic networks, signaling pathways, or regulatory networks. First released in 2001, it has gone through many updates, the latest version being level 3 version 2 (L3V2)[2]. SBML is the most popular format for the exchange and representation of biological models between different software, with 296 tools supporting SBML as of 2019[3]. SBML supports models of arbitrary complexity and is organized as a list of components (compartments, species, parameters, reactions...). For a detailed description of the SBML structure, we refer to its latest specifications[2]. Two main libraries for the parsing of SBML exist (i) libSBML[4] in C and C++ with language bindings for Python, R, Java™, and other languages; and (ii) JSBML[5] for all programming languages that can interpret Java bytecode. These libraries read SBML files into their internal data structures and provide many helper functions to work with SBML models, e.g., validation of SBML files or unit checking.

An essential feature of SBML is the annotation of the model and model components with meta-information describing the biological and mathematical meaning, as well as information about the provenance. Annotations play a crucial role in expressing the relationship between the parts and for linking components to external resources such as databases or experimental data. The added semantic layer allows users to interact with models and understand them.

Though SBML is the leading data format for biological models, JSON[6] is the primary data exchange format on the web. JSON objects consist of nested attribute-value pairs and array data types. Originally derived from JavaScript, JSON became a widely-used language-independent data format for the exchange of information, especially between web applications. All modern programming languages include methods to generate and parse JSON formatted data.
To support SBML in web applications and web-based workflows, the conversion of SBML models to JSON and vice versa without information loss is an extraordinary open challenge. Having a large JSON data format for SBML models would be a powerful output relevant to the interoperability of tools and workflows based on SBML. Tools and databases such as

cobrapy[7] or the BiGG[8] database already support and exchange models in JSON on the web. However, many issues exist with the currently used JSON representation of SBML models. Most importantly, essential information is lost while parsing the SBML model to JSON as well as in the back-conversion from JSON to SBML. For instance, in model annotations, biological qualifiers are used to define the kind of relationship which that component has with the annotated resource. These qualifiers function as verbs in the RDF triples describing how model components are related to meta-data. Without these verbs, the meaning of the metadata cannot be resolved, e.g., a vast difference exists between an "is" and "hasPart" relationship. No support for qualifiers live in the current JSON-SBML scheme of cobrapy, and if an SBML model is written in JSON-SBML, these qualifiers and the respective meaning of the annotation are lost. Many other issues of similar type exist in the JSON format, restricting the exchange of SBML as JSON.

Main problems are that

I. only a small subset of SBML is currently serialized to JSON by existing tools;
II. meta-information such as annotations are only partially supported;
III. no SBML-JSON scheme exists for validation of SBML-JSON. A well-defined scheme will help people to implement SBML-JSON support in web tools or other automated processors and provide a reference.

Having a full-featured JSON exchange format for SBML will improve the web-integration and exchange of SBML models, thereby increasing tool interoperability and reproducibility of workflows.

Motivation

Many factors motivate me to work on this project. First of all is biology, more specifically the application of computer science to better understand biological systems. I have studied biology up to the tenth standard and have learned about basic life processes. Today, after entering the field of Computer Science, and after understanding the structure of computers, I often wonder how complex the internal structure of a biological system must be to fulfill its complex tasks. Using computational methods to better understand biological systems is a perfect combination of these interests.
Secondly, open-source is something which I have always admired. I have been a part of open-source software development for the last one and a half years, and have learned a lot in that course of time. GSoC is all about open source, where people can contribute to their field of interest from any corner of the world. And the fact that the code I write will be used by many people from all around the globe to solve their problems further brings a cherry on the cake.
The JSON representation of SBML models is an important task to accomplish with impact for the biological modeling community. I look forward to gaining many new experiences by contributing my knowledge and skills to a professional open-source project.

Goals

The objectives of this project are

I. **Milestone 1:** provide comprehensive coverage of SBML features in JSON relevant for constraint-based modeling;
II. **Milestone 2:** provide full support of meta-information in JSON with lossless conversion between SBML and JSON;
III. **Milestone 3:** update the current SBML-JSON scheme with corresponding validation of SBML-JSON files against the scheme definition.

Implementation will be in Python as part of the cobrapy project with a focus on SBML models used in the constraint-based modeling community. Importantly, within this GSOC project, full representation in SBML-JSON of all features relevant for constraint-based methods, i.e., SBML core and additional SBML packages FBC and groups will be implemented, and all the current issues of JSON representation of SBML models as listed in the project[9] will be resolved.

Work plan

Before discussing the work plan in detail, let me provide a high-level overview of how cobrapy reads/writes SBML models, converts the SBML model to the internal cobra model data structure, and writes the cobra model to other data formats like JSON, YAML, etc. Depicted in Fig. 1 is a general overview of the import and export functionality of cobraby from various

file formats. For reading/writing models in formats like JSON, YAML, MATLAB, etc., cobrapy uses core python modules, i.e., 'json[10]' module for JSON models, 'scipy[11]' module for Matlab format, etc.

SBML models are parsed into cobra models using the libSBML library. Being written in C and C++, it's reading and writing methods for SBML (XML) files are very fast. The library provides language interfaces to C#, Java™, Octave, Matlab, R, Ruby, and Python. In cobrapy, the SBML models are first parsed into libSBML documents (objects of `SBMLDocument` class), which are subsequently converted to cobra models by mapping corresponding components between the two data models. So in this way, a cobra model is extracted from an SBML model.

JSON serialization works similarly by using the cobra model instance as an intermediate object. The project will mainly interact with the classes for JSON and SBML serialization, and with the internal cobra model instance.

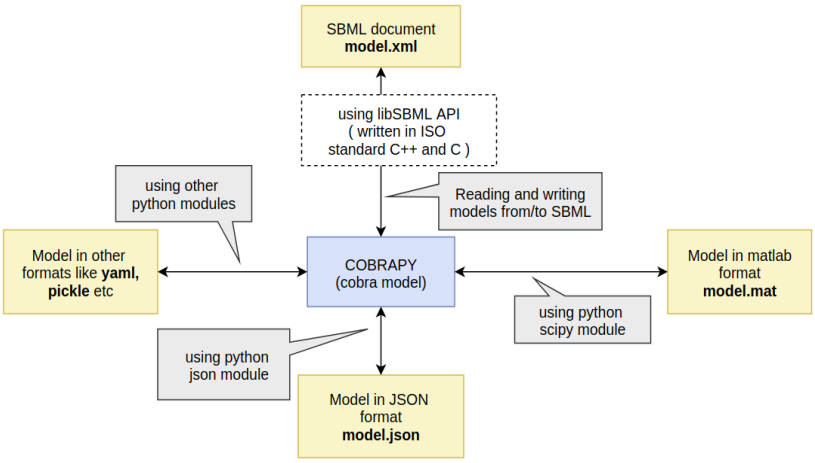


Fig.1 Cobrapy handling models in different format

The objectives of the project "**Implementation of SBML-JSON converter and SBML-JSON scheme**" will be achieved via three work packages (WP) corresponding to the respective milestones:

WP1: Comprehensive coverage of SBML features in SBML-JSON

The focus of WP1 is to correctly parse the SBML features important for constraint-based models and implement the representation of these model features in SBML-JSON. This includes the information from the SBML core, as well as the information in the additional SBML packages fbc and groups.

Milestone 1.1: Complete support of FBC-version 3 features in SBML-JSON

Currently, the SBML-JSON format only supports a subset of features introduced in the fbc version 2 package. With the latest fbc package, version 3 (fbc-v3), new vital functions have been added (see Fig. 2).

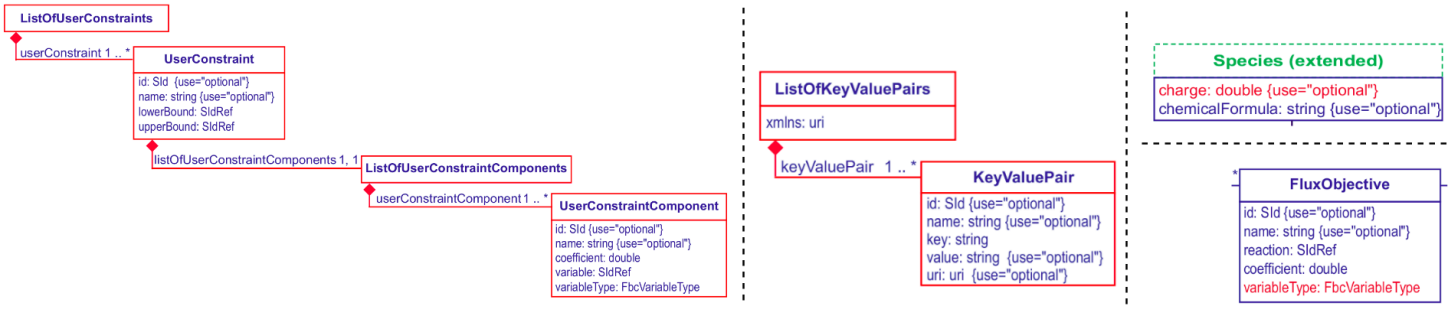


Fig. 2 New features introduced in fbc v3

The SBML package introduces non-stoichiometric constraints in addition to the classical stoichiometric constraints from reaction stoichiometries in the linear programs (LP). This vital extension allows for encoding many additional types of constraint-based models. Additional changes include better support for mass and charge balance calculation,

the introduction of `variableType` attribute in flux objective class that indicates whether a variable should be considered as 'linear' or 'quadratic' and a new extended annotation structure with the introduction of key-value pairs for storing biological information. Another open issue is the support for alternative objective functions, which can be encoded in SBML but are not yet represented in cobrapy. These features are necessary for the representation of constraint-based models in any type of format. An important task, hence, will be the implementation of this additional constraint information in the cobrapy model class and the SBML-JSON format.

Milestone 1.2: Complete support of 'Groups' features in SBML-JSON

A piece of valuable information in SBML models used in constraint-based modeling is an association of subsystems and groups to reactions which function as pathway definitions. The SBML *Groups* package allows for encoding such information. It adds the necessary features to SBML to enable the grouping of arbitrary model components to be expressed. Such groups do not affect the mathematical interpretation of a model, but they do provide a way to add information that can be useful for modelers and software tools. The SBML Groups package enables a modeler to include definitions of groups and nested groups, each of which may be annotated to convey why that group was created, and what it represents. Shown in Fig. 3 is the structure of the *Group* component. An SBML model has a top-level child component 'listOfGroups,' which has a list of different types of Groups under it. These members may or may not be of the same type. The Group collectively is used to either denote something or to attach the same kind of data to all its members to reduce writing the same content again and again.

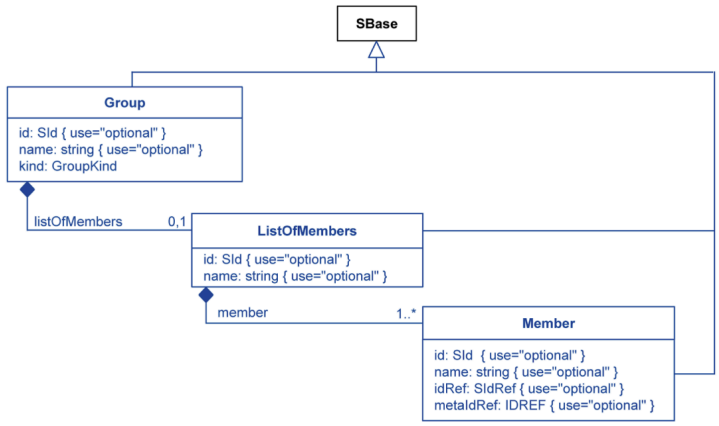


Fig. 3 The Group class structure in SBML

Currently, the *Groups* are supported in cobrapy, but the *Groups* information is not encoded in the SBML-JSON format. The Groups component is not very complicated, and can easily be written to SBML-JSON. Hence, an essential task of this project is to provide full support for encoding the *Groups* features in SBML-JSON. Dedicated methods for parsing the Group data to SBML-JSON will be made, which will reduce the data loss and will provide better model representation in SBML-JSON format.

Milestone 1.3: Solving issues related to model representation

Currently, multiple problems related to model representation in cobrapy exist, which are relevant for a complete representation of information in SBML-JSON. As part of this project, solutions for these issues will be implemented.

- **The consistent output of multiple-value elements as an array of string in JSON/YAML format (cobrapy issue #706).**
  When an SBML model is written in YAML/JSON format, the *identifiers* are represented as a single string if there is a unique *identifier*, and as a list of strings, if multiple *identifiers* corresponding to that *provider* exist. This makes the data type inconsistent and difficult to work with (string vs. a list of strings). This issue affects all formats supported by cobrapy because the cobra model itself stores the *provider-identifier* data in this manner. As a solution, consistent storage and serialization as a list of strings will be implemented, which will simplify the parsing of JSON/YAML (and other formats). For this, the '`_parse_annotations()`' method (where this parsing is done) will be updated.

- **JSON export version attribute does not match JSON-scheme (cobrapy issue #720)**
  The current JSON scheme specifies the *'version'* attribute of a JSON model to be of type integer, but the following string type global variable is used to write the version attribute to SBML-JSON file:

```
JSON_SPEC = "1"
```

This produces a `TypeError` due to a type mismatch. This JSON_SPEC variable should be an integer because: (i) the JSON scheme specifies its type to be of integer type, and (ii) the SBML also has its version of integer type. This issue simply requires `JSON_SPEC` to have an integer value instead of a string. Along with this, the version attributes of other formats also (like YAML), should also be made of integer type to have consistent types for all formats. Moreover, along with the version attribute, a level attribute should also be used (of integer type) to denote the minor updates, as used in SBML format.

- **Cannot export a model from JSON to SBML when the annotation is a list of lists** (**cobrapy issue #736**)
  Many models in JSON format have annotations represented in the form of a list of lists (e.g., BiGG universal model). Though the JSON scheme clearly defines annotation to be of type object (such as a python dictionary), some models have annotation in the form of a list of lists. The motivation behind this can be that in SBML-XML, the Annotation content type is any, allowing essentially arbitrary well-formed XML data content. But, even SBML places restrictions on the organization of the content of the annotation (discussed in *specification doc*[2]). Hence, in JSON also, the data structure format for annotation has to be fixed, which will be covered in **Milestone 2.1** for the complete implementation of annotation.

- **JSON format does not support -infinity/infinity bounds** (**cobrapy issue #856**)
  The current JSON format for the representation of models does not allow any attribute (like upper/lower flux bound of a reaction, reaction parameters, etc.) to take -infinity/infinity values. Though SBML supports infinity values, if such a model is written in JSON a ValueError is thrown:

  ```
  ValueError: Out of range float values are not JSON compliant: inf
  ```

  This is not an error due to JSON not supporting infinity values. It is only because the value for the "*allow_nan*" parameter passed to method '`json.dump(...)`', which writes the model in JSON form, is currently false. The *'allow_nan'* parameter specifies whether to allow *NaN* values (such as infinity, -infinity, number/0, etc.) to be written from a dictionary to the JSON file or not. Hence, instead of using the default value to be false, the user should decide whether to allow NaN values to be written to the JSON file or not. The corresponding method will be modified, with an extra argument to enable the user to the setting of '*allow_nan*' value.

WP2: Provide full support of meta-information in SBML-JSON

Meta-information in the SBML model is provided in the form of annotations and notes attached to each component of the model. Currently, this meta-information is not fully supported in cobrapy, and many issues exist related to it. Different metadata constructs attached to a component are used to to encode different information:

- Annotations to controlled vocabulary terms and database identifiers link to external resources and describe the meaning of model components (addressed in *Milestone 2.1*).
- Notes are used to attach information to be seen by humans and are only minimally supported in the current version of cobrapy (addressed in *Milestone 2.2*).
- Meta-information on model provenance and model history is encoded by the ModelHistory and Creator elements (addressed in *Milestone 2.3*).

Milestone 2.1: Implementation of Annotation and meta-data support

Currently, Annotations of a component in the SBML model are defined in the `Object` class of cobrapy. This `Object` class is the main parent class in cobrapy, and other classes which represent a component of the model, such as the `Reaction` class, the `Metabolite` class, inherit from it The `Object` provides the basic attributes like name, id, etc. which are used by all SBML components. Annotations are defined to be a *dictionary* type on `Object` without any restrictions placed on the data inside the Annotation (the current scheme only specifies the annotation to be of type *object*, i.e., *dictionary*). Currently, the parsing of annotation information from SBML to the cobra model is handled by

the "`_parse_annotations()`" method of `cobra.core.io.sbml` which maps the SBML annotation data using key-value pairs in the annotation dictionary of the cobra model. This method does not store the biological qualifiers present in the annotation. It merely maps the providers in the resource to the identifier. If there is more than one identifier corresponding to a provider, it stores them as a list under the provider as key and list of identifiers as value. Hence, the following SBML annotation (left) is currently stored in the following way in the cobra model (right)

SBML Annotation

```
<annotation>
  <rdf:RDF ...(other namespaces)... >
    <rdf:Description rdf:about="#G_b0351">
      <bqbiol:is>
        <rdf:Bag>
          <rdf:li rdf:resource="https://identifiers.org/uniprot/P77580" />
          <rdf:li rdf:resource="http:s//identifiers.org/uniprot/P0A6A3" />
        </rdf:Bag>
      </bqbiol:is>
      <bqbiol:isEncodedBy>
        <rdf:Bag>
          <rdf:li rdf:resource="https://identifiers.org/asap/ABE-0001207" />
          <rdf:li rdf:resource="https://identifiers.org/ecogene/EG13625" />
          <rdf:li rdf:resource="https://identifiers.org/ncbigene/945008" />
          <rdf:li rdf:resource="https://identifiers.org/ncbigene/945008" />
        </rdf:Bag>
      </bqbiol:isEncodedBy>
    </rdf:Description>
  </rdf:RDF>
</annotation>
```

Cobra model annotation

```
annotation = {
       "uniprot": ["P77580", "P0A6A3"],
       "asap": "ABE-0001207",
       "ecogene": "EG13625",
       "ncbigene": ["945008", "945008"]
 }
```

The current implementation loses the biological qualifiers when an SBML model is converted to a cobra model. A simple python dictionary is clearly insufficient to completely parse the annotation data because annotation can have an arbitrary format, which makes its structure quite complicated, except in cases where either only external database references are included via "is" relationships. There has been a discussion going on about the format for annotation, and many issues have been reported regarding the storing of meta-information on the issue list of cobrapy. Attaching other types of meta-data like the confidence score of a compartment, Gibbs free energy of a reaction, etc., are still uncertain as to where they should be attached (issue #4). A generic Annotation class, with unique methods to check the inner Annotation data format, is, hence, urgently needed to store all Annotation information in the internal cobrapy data structure and to serialize Annotation objects from/to different model exchange formats. It will be best if we, here also, follow the structure followed by libSBML to handle new different types of annotation, i.e., a tree of XMLNodes (shown in fig. 5).

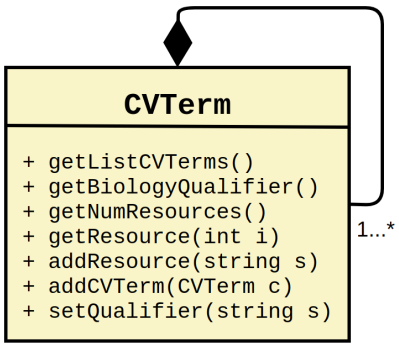


Fig. 4 CVTerm class to handle external resources

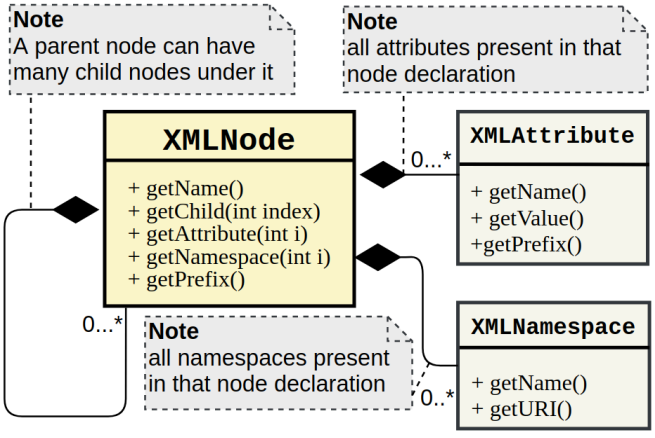


Fig. 5 An XMLNode in SBML document

The already defined annotation structures such as those of MIRIAM resources will have dedicated classes for their easy handling because they are the one which mostly appears within the annotation field. But when some new, self-defined meta-data is to be attached to any component, the XMLNode data structure will be best suited. For MIRIAM resources annotation, the class structure will be similar to the CVTerm class of libSBML (fig. 4), which is used to store the annotation's data related to database identifiers in the libSBML models. Hence, within this milestone, a general-purpose Annotation class will be implemented, which allows for handling the full spectrum of SBML RDF annotations. Moreover, methods to validate the critical complex features of the annotation will be added, such as:

- Validation against MIRIAM resources and identifiers.org resources
- Handling annotation with "URL references," which do not map to identifers.org resources, i.e., those resources which cannot be split into a resource and an identifier.
- Encoding of complex composite annotations consisting of more than a single annotation term.

This will allow validation of annotations, e.g. if the annotations follow allowed patterns or not. The following structure of annotation data in the class will look similar to a dictionary. Still, the data will be validated, and this structure will be used to map the data to the corresponding SBML-JSON file.

Milestone 2.2: Provide complete support for Notes field in cobrapy model and SBML-JSON

The current SBML-JSON format does have no complete support for notes. The subcomponent Notes is a container for XHTML 1.0 content. It is intended to serve as a place for storing optional information intended to be seen by humans. Notes are currently supported in cobrapy only in a basic manner, i.e., only data of the form "`<p>key: value</p>`" is parsed to the cobra model. But Notes can contain any well-formatted XHTML data, so any data not in the above form is lost while parsing SBML to cobra models. It is, therefore, necessary to have a well defined structured format for storing the notes field related to a component. So to handle the Notes, I propose the same format for handling Notes that are used in the libSBML, i.e., a tree of XMLNodes (shown in fig. 5). Each Node in this tree will have a list of children which themselves are going to be XMLNodes and also, a list of XMLAttributes and XMLNamespaces attached to that Node. In this way, a complete hierarchical tree of XMLNode will be formed, and there will be no issue of having a completely new different type of XHTML element anywhere. Also, to make the importing and exporting of some simple general notes easy and less cumbersome, an unordered list will be added as one top-level child of the body element of notes, and direct methods of adding and extracting notes from this unordered list will be made to make the importing and exporting of general notes from the Notes field easy and simple. This way, there will be no restriction on any type of data format until the notes data being added is following a general XHTML format.

Milestone 2.3: Support for meta-information about a model's creator and history

So along with storing external resources, annotations are also used to store information about the model's (or model component's) creator and its history. Though, while reading an SBML model from its corresponding SBML file, cobrapy stores this information in the cobra model (in a private attribute *_sbml* attached to the *cobra_model*), it is never used when the model is written in the JSON file, and, hence, this data is lost here. The parsing from the SBML file to the cobra model related to the model's history is not complete even, i.e., details about creator and the date of its creation is passed, the modification data is not yet parsed. Hence, dedicated classes for encoding model provenance and model history will be added to cobrapy within this milestone, which will ensure the passing of complete meta-information of an SBML model. The ModelHistory class of libSBML, specifically used to handle data related to the last name, given name, email address, organization, creation date, and modification date, will be taken as the reference class and the necessary methods to handle the data inside the model's history will be made (in fig. 6). Other than this, to handle the data that is not defined by this scheme but is valid XML syntax and is consistent with relevant standards for enclosing elements (shown by '`+++`' in SBML specification document), the same XMLNode tree structure will be followed which will allow reading and writing of this extra content. A simple JSON format out of this structure will then be made to write the corresponding

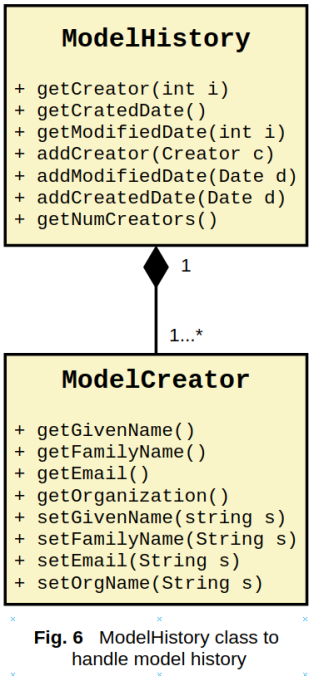


Fig. 6 ModelHistory class to handle model history

data to the SBML-JSON file. To give you an instance, see the below conversion of the model's creator and history from SBML to JSON.

SBML Annotation

```
<annotation>
  <rdf:RDF …(other namespaces)...>
    <rdf:Description rdf:about="#_180340">
      <dcterms:creator>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <vcard4:fn>
              <vcard4:text>Bruce Shapiro</vcard4:text>
            </vcard4:fn>
            <vcard4:email>bshapiro@jpl.nasa.gov</vcard4:email>
            <vcard4:organization-name>
              NASA Laboratory
            </vcard4:organization-name>
          </rdf:li>
        </rdf:Bag>
      </dcterms:creator>
      <dcterms:created rdf:parseType="Resource">
        <dcterms:W3CDTF>2005-02-06T23:39:40+00:00</dcterms:W3CDTF>
      </dcterms:created>
      <dcterms:modified rdf:parseType="Resource">
        <dcterms:W3CDTF>2005-09-13T13:24:56+00:00</dcterms:W3CDTF>
      </dcterms:modified>
    </rdf:Description>
  </rdf:RDF>
</annotation>
```

JSON Annotation

```
"annotation" : {
  "creator" : [
    {
      "fn" : {
        "text" : "Bruce Shapiro"
      },
      "email" : "bshapiro@jpl.nasa.gov",
      "organization-name" : "NASA Laboratory"
    }
  ],
  "created" : {
    "W3CDTF" : "2005-02-06T23:39:40+00:00"
  },
  "modified" : [
    {"W3CDTF" : "2005-09-13T13:24:56+00:00"}
  ]
```

WP3: Updating the current SBML-JSON scheme

Being able to validate data exchange formats against their specification is crucial for ensuring the correct and high-quality exchange of information between tools. These become particularly important on the web. The basis of validating JSON data is to provide a scheme of the respective data format, which clearly encodes the structure and allowed constructs in the data file. It adds domain-specific rules on top of the generic JSON exchange format. Importantly, with an existing scheme file, all tools can validate data sent in SBML-JSON against the scheme, because such functionality is routinely included in JSON libraries and frameworks. Work-package 3 will update the current SBML-JSON scheme.

Milestone 3.1: Updating the current SBML-JSON scheme as per the features implemented above and other additional unimplemented features.

The current SBML-JSON scheme does not support all features essential for representing constraint-based models and needs updates to accommodate additional features. For example, the current SBML-JSON scheme does not put any restrictions on the format of Annotations and Notes of a component. It merely specifies them to be of the type JSON object (like python dictionary). However, as discussed in the above implementations, we are going to have a defined format for our meta-information, therefore, corresponding to that format, the scheme will have to be updated. The outdated scheme is blocking implementations of some tests also (such as test_json.py), which are marked as *xfail* (expected to fail due to some reason) due to the outdated scheme. Also, the current scheme does not have any specification for groups; neither are groups written from SBML to JSON files. Hence, support for these additional features will be added to the scheme and for the JSON format.

Milestone 3.2: Updating the current SBML-JSON scheme draft version.

The current SBML-JSON scheme uses the draft-04 version of the scheme, whereas the latest draft version for the scheme is draft-07. Though draft-07 is just a relatively minor update of draft-06, draft-06 has many notable updates from draft-04. Hence, the SBML-JSON scheme requires a draft update also, which will make it compatible with the current scheme draft specification. This will be covered as part of this milestone.

Milestone 3.3: Validation of SBML-JSON files against the scheme definition.

The scheme for SBML-JSON is currently not used to validate the JSON files in cobrapy. If the validation of a JSON file is not performed at the time when the model is read, then subsequent errors may occur at the time when the model is analyzed further. This will confuse the user about what could have gone wrong and makes it more challenging to write algorithms based on the cobra model. An important task is, therefore, to implement validation of the JSON file against the SBML-JSON scheme when the model is read from JSON.

Milestone 3.4: Testing and documentation of scheme definition

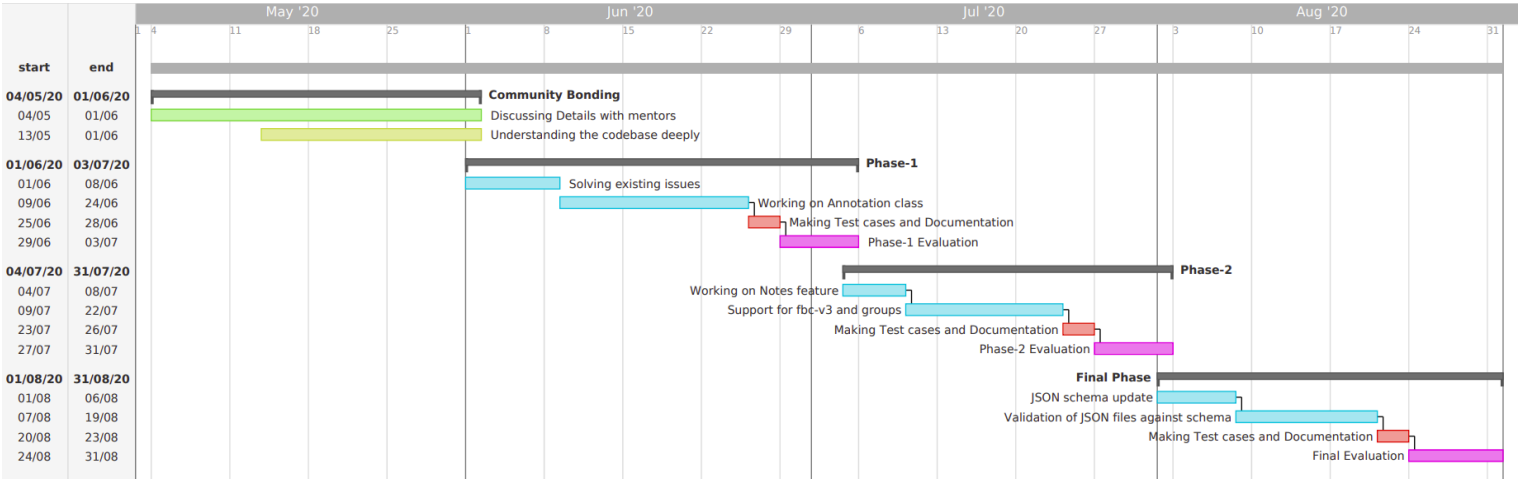
Test cases and test code for the scheme and scheme validation will be added. A test-suite of reference SBML-JSON files will be created, which allows simple testing of SBML-JSON implementation in other tools. The updated annotation format and other changes that are made in the SBML-JSON scheme will require updating existing tests of cobrapy. Tests for the added features will be implemented. Along with this, all newly implemented features will be documented to provide a complete description to users and developers.

Timeline

Date	Tasks
April 1 - May 3	1. Application submitted 2. Understand the cobrapy codebase with special focus on code related to parsing of formats and features relevant for constraint-based methods
May 4 - June 1	1. Interact with mentors, discuss implementation details and finalize them 2. I have End term examination from 24th April to 4th May. So, I will be a little less active during this period
June 1 - June 8	1. Resolve issues listed in **Milestone 1.3** 2. Start working towards **Milestone 2.1,** i.e., laying the detailed structure of the Annotation class
June 9 - June 24	1. Start implementation of Annotation classes 2. Complete **Milestone 2.1**, **Milestone 2.3** and the listOfKeyValue feature from **Milestone 1.1**
June 25 - June 28	1. Make test cases for the above classes as part of **Milestone 3.4** 2. Document the above classes as part of **Milestone 3.4**
June 29 - July 3	**PHASE-1 EVALUATION** Write my blog and interact with mentors for evaluation
July 4 - July 8	1. Implement complete Notes feature in cobrapy as well as SBML-JSON (**Milestone 2.2**)
July 9 - July 22	1. Implement support for new fbc-v3 features (**Milestone 1.1**) 2. Implement support for Groups in SBML-JSON (**Milestone 1.2**)
July 23 - July 26	1. Make test cases for the above features as part of **Milestone 3.4** 2. Document the above features as part of **Milestone 3.4**

July 27 - July 31	**PHASE-2 EVALUATION** Write my blog and interact with mentors for evaluation
August 1 - August 6	1. Update JSON scheme (**Milestone 3.1**) 2. Update the draft version of the scheme (**Milestone 3.2**)
August 7 - August 19	1. Validation of JSON files against scheme (**Milestone 3.3**)
August 20 - August 23	1. Work on test cases and fix old broken tests (**Milestone 3.4**) 2. Final documentation of all features (**Milestone 3.4**)
August 24 - August 31	**FINAL EVALUATION** Interact with mentors for evaluation and submission of complete code

Gantt Chart



Prior Contributions

Pull Requests Merged

I. **PR #930:** This PR is part of issue #736, i.e., handling annotation when they are a list of lists.
II. **PR #943:** This PR is solving issue #902, i.e., handling model notes which are not written to the SBML file.

Issues Opened

I. **Issue #937**: The issue was about the loss of relational qualifiers when round-tripping the SBML - cobra model - SBML. Though, it was later found that this issue already exists and has been reported.

About Me

Education

I. I am an undergraduate (B. Tech.) majoring in Computer Science and Engineering, currently in the second year of my studies.
II. Relevant courses include Object-Oriented Programming, Object-Oriented Analysis and Design, Data Structures and Algorithms, and Software Engineering.

Experience

I. I have been actively involved in Mobile Development since my first year. I have made several apps and libraries, which gave me substantial experience in JAVA™, XML, JSON (Networking), and working with APIs.
II. I am experienced in Web Development since I have been involved in it for the last year, both frontend and backend in a few projects. This gave me experience, among others in React (Javascript), Django (Python), and Node.js (JavaScript). My JSON experience comes from these web projects also, as JSON is the dominant data exchange format for most web applications.
III. Being familiar with technologies such as version control (git), collaborative development using Github, and experience with various IDE's, it will not take me time getting all set with development.
IV. My projects can be viewed on my Github account.
V. I have prior experience with Biology up to the tenth standard. I further studied chemistry up to twelfth standard as the main subject for the Joint Entrance Examination (JEE), where I learned about different types of reactions, rate laws, etc. under chemical kinetics.

Why Me?

I. It might sound a little odd, but I find myself as an individual who is always curious about learning new things and interacting with other people. By working with an organization like NRNB and its amazing members, I will learn many new things, not just about data handling and its manipulation from one format to another, but also about various biology models and the processes they carry out to support life.
II. I have prior experience with software development, as I have done numerous coursework projects and projects by myself. I am familiar with software development practices in teams.
III. Always being a team player, I understand the importance and way of working in open source development. I respect everyone's views and am familiar with best practices and recommendations.
IV. I have been in constant touch with the mentors of the project and the members of cobrapy, getting a better understanding of the project via self-exploration and regular clarification of unclear points with the cobrapy team and mentors.

Commitments/Precautions

I have my end semester examination scheduled from 24th April to 4th May (community bonding period). So I will be a little less active during this period. Other than this, I have no prior commitments. Because of unforeseen events, I may be unavailable for a day or two at maximum. Still, I will make sure to inform my mentors about the unavailability as soon as I can. Apart from this, I have ample time to give to the project, an average of about 40-45 working hours per week. I do not have any other internship/project for this summer, so I will easily be able to manage time for the project.

References

[1] http://sbml.org/Main_Page
[2] http://sbml.org/Documents/Specifications/SBML_Level_3/Version_2/Core
[3] http://sbml.org/SBML_Software_Guide
[4] http://sbml.org/Software/libSBML
[5] http://sbml.org/Software/JSBML
[6] https://www.json.org/json-en.html
[7] https://opencobra.github.io/cobrapy/
[8] http://bigg.ucsd.edu/
[9] https://github.com/nrnb/GoogleSummerOfCode/issues/137
[10] https://docs.python.org/3/library/json.html
[11] https://pypi.org/project/scipy/
[12] http://www.swig.org/
[13] http://bigg.ucsd.edu/data_access
[14] https://github.com/opencobra/cobra-component-models