
ObjTables documentation

Release 1.0.14

Jonathan Karr, Arthur Goldberg

Dec 10, 2020

CONTENTS

1	Contents	3
1.1	Installation	3
1.1.1	Prerequisites	3
1.1.2	Installing the latest release from PyPI	3
1.1.3	Installing the latest revision from GitHub	4
1.1.4	Installing the optional features	4
1.1.5	Configuring access to GitHub	4
1.2	Migration a dataset between versions of its schema	5
1.2.1	Overview	5
1.2.2	Concepts	5
1.2.3	Configuring migrations	8
1.2.4	Topological sort of schema changes	15
1.2.5	Migration protocol	17
1.2.6	Using migration commands	18
1.2.7	Known limitations	20
1.3	About	20
1.3.1	License	20
1.3.2	Development team	21
1.3.3	Acknowledgements	21
1.3.4	Questions and comments	21

ObjTables is a toolkit which makes it easy to use spreadsheets (e.g., XLSX workbooks) to work with complex datasets by combining spreadsheets with rigorous schemas and an object-relational mapping system (ORM; similar to Active Record (Ruby), Django (Python), Doctrine (PHP), Hibernate (Java), Propel (PHP), SQLAlchemy (Python), etc.). This combination enables users to use programs such as Microsoft Excel, LibreOffice Calc, and OpenOffice Calc to view and edit spreadsheets and use schemas and the *ObjTables* software to validate the syntax and semantics of datasets, compare and merge datasets, and parse datasets into object-oriented data structures for further querying and analysis with languages such as Python.

ObjTables makes it easy to:

- Use collections of tables (e.g., an XLSX workbook) to represent complex data consisting of multiple related objects of multiple types (e.g., rows of worksheets), each with multiple attributes (e.g., columns).
- Use complex data types (e.g., numbers, strings, numerical arrays, symbolic mathematical expressions, chemical structures, biological sequences, etc.) within tables.
- Use programs such as Excel and LibreOffice as a graphical interface for viewing and editing complex datasets.
- Use embedded tables and grammars to encode relational information into columns and groups of columns of tables.
- Define clear schemas for tabular datasets.
- Use schemas to rigorously validate tabular datasets.
- Use schemas to parse tabular datasets into data structures for further analysis in languages such as Python.
- Compare, merge, split, revision, and migrate tabular datasets.

The *ObjTables* toolkit includes five components:

- Format for schemas for tabular datasets
- Numerous data types
- Format for tabular datasets
- Software tools for parsing, validating, and manipulating tabular datasets
- Python package for more flexibility and anal

Please see <https://www.objtables.org> for an overview of *ObjTables* and https://sandbox.karrlab.org/tree/obj_tables for interactive tutorials for the *ObjTables* Python API. This website contains documentation for *ObjTables* migrations and the *ObjTables* Python API.

CONTENTS

1.1 Installation

The following is a brief guide to installing the *ObjTables* Python API and command line program. The [Dockerfile](#) in the *ObjTables* Git repository contains detailed instructions for how to install *ObjTables* in Ubuntu Linux.

1.1.1 Prerequisites

First, install the following third-party packages:

- [ChemAxon Marvin](#) (optional): to calculate major protonation and tautomerization states
 - Java ≥ 1.8
- [Git](#) (optional): to revision schemas and datasets
- [Graphviz](#) (optional): to generate UML diagrams of schemas
- [Open Babel](#) (optional): to represent and validate chemical structures
- [Pip](#) ≥ 18.0
- [Python](#) ≥ 3.6
- [SSH](#) (optional): to use Git with SSH to revision schemas and datasets

To use ChemAxon Marvin, set `JAVA_HOME` to the path to your Java virtual machine (JVM) and add Marvin to the Java class path:

```
export JAVA_HOME=/usr/lib/jvm/default-java
export CLASSPATH=$CLASSPATH:/opt/chemaxon/marvinsuite/lib/MarvinBeans.jar
```

1.1.2 Installing the latest release from PyPI

Second, we recommend that users run the following command to install the latest release of *ObjTables* from PyPI:

```
pip install obj_tables
```

1.1.3 Installing the latest revision from GitHub

We recommend that developers use the following commands to install the latest revision of *ObjTables* and its dependencies from GitHub:

```
pip install git+https://github.com/KarrLab/pkg_utils.git#egg=pkg_utils
pip install git+https://github.com/KarrLab/wc_utils.git#egg=wc_utils[chem]
pip install git+https://github.com/KarrLab/bpforms.git#egg=bpforms
pip install git+https://github.com/KarrLab/bcforms.git#egg=bcforms
pip install git+https://github.com/KarrLab/obj_tables.git#egg=obj_tables
```

1.1.4 Installing the optional features

ObjTables includes several optional features:

- *bio*: Biology attributes for sequences, sequence features, and frequency position matrices (`obj_tables.bio`)
- *chem*: Chemistry attributes for chemical formulas and structures (`obj_tables.chem`)
- *grammar*: Encoding/decoding objects and their relationships into and out of individual cells in tables (`obj_tables.grammar`)
- *math*: Mathematics attributes for arrays, tables, and symbolic expressions (`obj_tables.math`)
- *web*: Web service (`obj_tables.web_service`)
- *revisioning*: Revisioning schemas and data sets with Git (`obj_tables.migrate`)
- *sci*: Science attributes for units, quantities, uncertainty, ontology terms, and references (`obj_tables.sci`)
- *viz*: Methods to generate UML diagrams of schemas (`obj_tables.utils.viz_schema()`)

These features can be installed by installing *ObjTables* with the desired options. For example, the *bio* and *chem* features can be installed by running one of the following commands:

```
pip install obj_tables[bio,chem]
pip install git+https://github.com/KarrLab/obj_tables.git#egg=obj_tables[bio,chem]
```

1.1.5 Configuring access to GitHub

To use the revisioning and migration features, developers must configure *ObjTables* to access GitHub.

- Install the revisioning features by running `pip install obj_tables[revisioning]`.
- [Generate an API token for GitHub](#).
- Create the directory `~/ .wc/` (Ubuntu: `/home/<username>/ .wc`, Windows: `c:\Users\<username>\ .wc\`).
- Create the file `~/ .wc/wc_utils.cfg` with the following content:

```
[wc_utils]
[[github]]
    github_api_token = <GitHub API token>
```

- Follow these steps to configure SSH access GitHub:
 - Follow these [instructions](#) to generate an SSH key and add it to your GitHub account

- Create the file `~/.gitconfig` (Ubuntu: `/home/<username>/.gitconfig`, Windows: `c:\Users\<username>\.gitconfig`) with the following content:

```
[url "ssh://git@github.com/"]
  insteadOf = https://github.com/
```

1.2 Migration a dataset between versions of its schema

1.2.1 Overview

Consider some data whose structure (also known as its *data model*) is defined by a schema written in a data definition language. For example, the structure of an SQL database is defined by a schema written in SQL's *Structured query language*.

When a schema is updated then existing data must be changed so that its structure complies with the updated schema. This is called *data migration*. Many systems, including database systems and web software frameworks, provide tools that automate data migration so that users can avoid the tedious and error-prone manual effort that's usually required when a schema is changed and large amounts of data must be migrated.

Packages that use *ObjTables* (`obj_tables`) store data in CSV, TSV, or XLSX files. The structure of the data in a file is defined by a schema that uses `obj_tables`. *ObjTables migration* enables automated migration of these data files.

This page explains the concepts of *ObjTables* migration and provides detailed instructions on how to configure and use it.

1.2.2 Concepts

ObjTables migration automates the process of migrating data files that use a schema which has been updated.

Migration assumes that data files which are migrated and the schemas that define their data models are stored in Git repositories. The repository storing the data files is called the *data repo* while the repository containing the schema is the *schema repo*. While these are typically two distinct repositories, migration also supports the situation in which one repository is both the *data repo* and the *schema repo*.

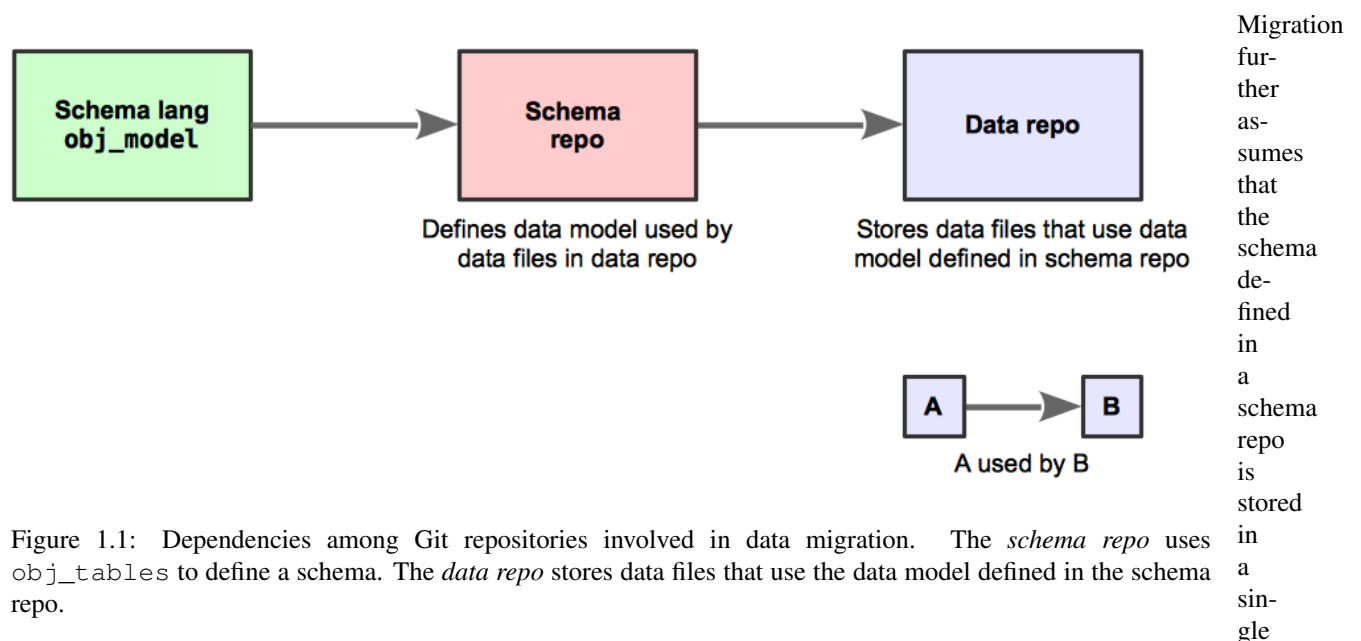


Figure 1.1: Dependencies among Git repositories involved in data migration. The *schema repo* uses `obj_tables` to define a schema. The *data repo* stores data files that use the data model defined in the schema repo.

Python file, which is called the *schema* file. Because it's stored in a Git repository, the schema file's version history is recorded in the schema repo's commits, which are used by migration. Figure 1.2 below illustrates these concepts.

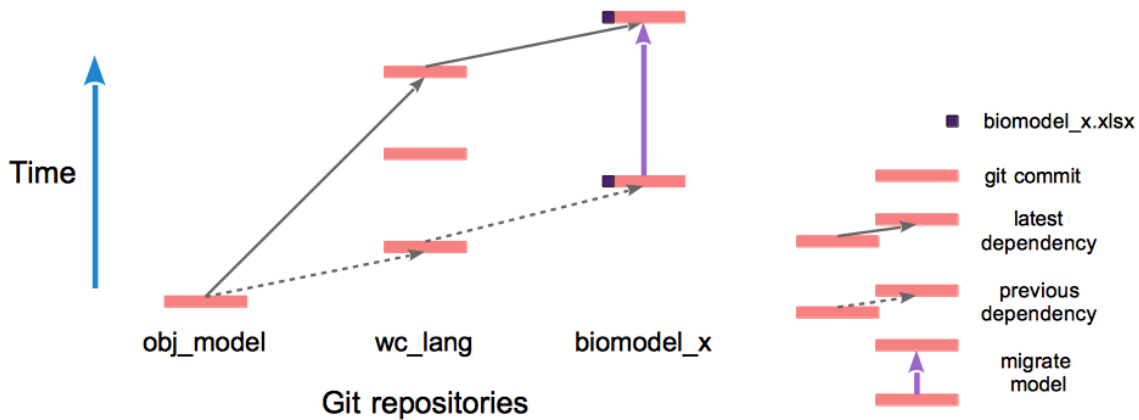


Figure 1.2: Example migration of file `biomodel_x.xlsx`. Three Git repositories are involved: `obj_tables`, `wc_lang`, and `biomodel_x`. Time increases upward, and within any repository later commits depend on earlier ones. `wc_lang` is a schema repo that defines the data model for files stored in the data repo `biomodel_x`. The earliest illustrated commit of `biomodel_x` contains a version of `biomodel_x.xlsx` that depends on the earliest commit of `wc_lang`, as indicated by the dashed arrow. Two commits update `wc_lang`. If these commits modify `wc_lang`'s schema, then `biomodel_x.xlsx` must be migrated. The migration (solid purple arrow) automatically makes the data in `biomodel_x.xlsx` consistent with the latest commit of `wc_lang`.

We decompose the ways in which a schema can be changed into these categories:

- Add a `obj_tables.core.Model` (henceforth, *Model*) definition
- Remove a *Model* definition
- Rename a *Model* definition
- Add an attribute to a *Model*
- Remove an attribute from a *Model*
- Rename an attribute of a *Model*
- Apply another type of changes to a *Model*

Migration automatically handles all of these change categories except the last one, as illustrated in Figure 1.3. Adding and removing *Model* definitions and adding and removing attributes from *Models* are migrated completely automatically. If the names of *Models* or the names of *Model* attributes are changed, then configuration information must be manually supplied because the relationship between the initial and final names cannot be determined automatically when multiple names are changed. Other types of modifications can be automated by custom Python transformation programs, which are described below.

The code below contains a schema that's defined using *ObjTables*. This documentation employs it as the schema for an example data file before migration, and refers to it as the *existing* schema:

```
from obj_tables import (Model, SlugAttribute, StringAttribute,
                        FloatAttribute, PositiveIntegerAttribute)

class Test(Model):
    id = SlugAttribute()
```

(continues on next page)

	Add	Delete	Rename	Modify
Model changes				
Attribute changes				
Attribute property change				

Handled automatically	
Annotated change	
Transformation change	

Figure 1.3: Types of schema changes. Changes that add or delete *Models* or *Model* attributes are handled automatically by migration. Changing the name of a *Models* or attributes must be annotated in a manually edited configuration file. Changes that do not fall into these categories must be handled by a custom Python transformations module that processes each *Model* as it is migrated.

(continued from previous page)

```

name = StringAttribute(default='test')
existing_attr = StringAttribute()
size = FloatAttribute()
color = StringAttribute()

class Property(Model):
    id = SlugAttribute()
    value = PositiveIntegerAttribute()

```

This example shows a changed version of the *existing* schema above, and we refer to it as the *changed* schema:

```

from obj_tables import Model, SlugAttribute, StringAttribute, IntegerAttribute

class ChangedTest(Model): # Model Test renamed to ChangedTest
    id = SlugAttribute()
    name = StringAttribute(default='test')
    # Attribute Test.existing_attr renamed to ChangedTest.migrated_attr
    migrated_attr = StringAttribute()
    # Attribute ChangedTest.revision added
    revision = StringAttribute(default='0.0')
    # Type of attribute Test.size changed to an integer
    size = IntegerAttribute()
    # Attribute Test.color removed

# Model Property removed

# Model Reference added

class Reference(Model):
    id = SlugAttribute()
    value = StringAttribute()

```

1.2.3 Configuring migrations

To make migration easier and more reliable the durable state used by migration in *schema repos* and *data repos* is recorded in configuration files.

Sentinel commits

To organize the changes in a schema repo into manageable groups, migration identifies *sentinel* commits that delimit sets of commits that change the schema. Considering the repo's commit dependency graph, sentinel commits must be located in the graph so that each commit which changes the schema depends on exactly one upstream sentinel commit, and is an ancestor of exactly one downstream sentinel commit (see [Figure 1.4](#)). In addition, a sentinel commit can have at most one ancestor sentinel commit that's reachable without traversing another sentinel commit. All the commits that are ancestors of a sentinel commit and depend upon the sentinel commit's closest ancestor sentinel commit are members of the sentinel commit's *domain*. In addition, a sentinel commit is a member of its own *domain*. We use the term *domain* to describe both commits and the changes to the schema made by the commits.

Migration migrates a data file across a sequence of sentinel commits.

Sentinel commit configurations that violate the constraints in the first paragraph of this section create Git histories that cannot be migrated. For an example, see [Figure 1.5](#).

Configuration files

Schema repos contain three types of configuration files ([Table 1.1](#)):

- A *schema changes* file identifies a sentinel commit, and annotates the changes to the schema in the sentinel commit's domain. Symmetrically, each sentinel commit must be identified by one schema changes file.
- A *transformations* file defines a Python class that performs user-customized transformations on *Models* during migration.
- A `custom_io_classes.py` file in a *schema repo* gives migration handles to the schema's `Reader` and/or `Writer` classes so they can be used to read and/or write data files that use the schema.

Since committed changes in a repository are permanent, the schema changes and transformations files provide permanent documentation of these changes for all migrations over the changes they document.

Data repos contain just one type of configuration file ([Table 1.2](#)):

- A *data-schema migration configuration* file details the migration of a set of data files in the data repo.

[Table 1.1](#) and [Table 1.2](#) describe these user-customized configuration files and code fragments in greater detail.

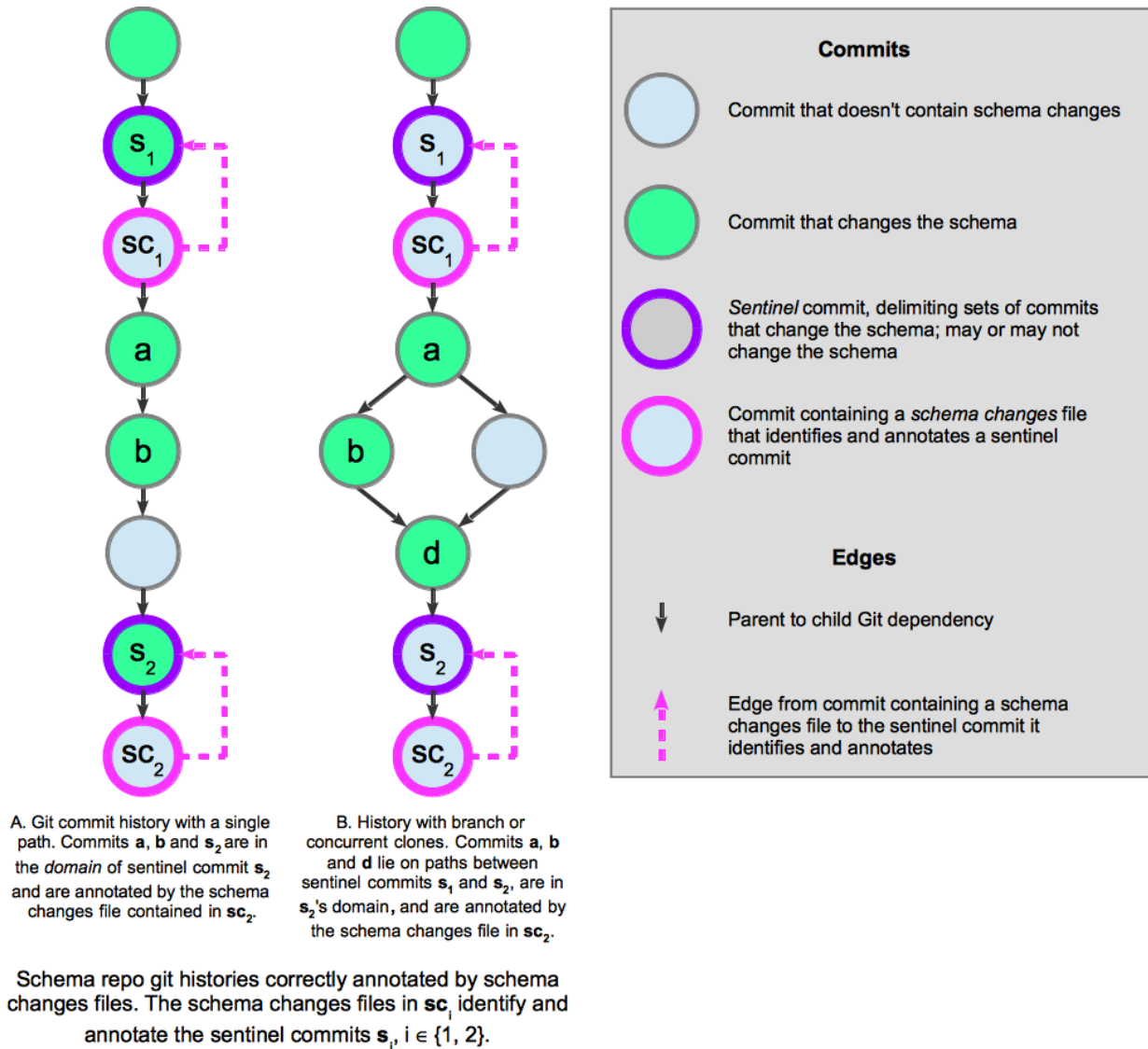


Figure 1.4: Sentinel commits in schema repo commit histories. Commits that do not change the schema may be present, but are not involved in migration. Each sentinel commit delimits the downstream boundary of a set of commits. In A, the changes in commits **a** and **b** will be applied to data being migrated from sentinel **s₁** to sentinel **s₂**. B illustrates a Git history created by branching or concurrent clones, but the commits **a**, **b**, and **d** still depend on exactly one upstream sentinel commit, **s₁**, and are ancestors of exactly one downstream sentinel commit, **s₂**.

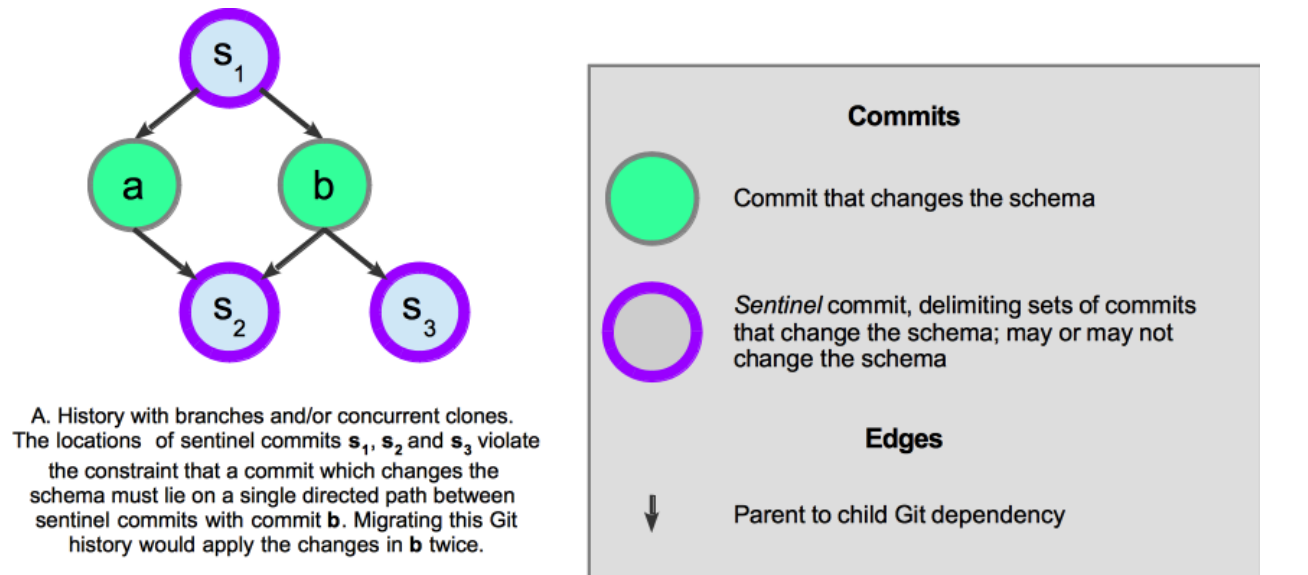


Figure 1.5: Sentinel commits in a schema repo commit history that cannot be migrated.

Table 1.1: Configuration files in schema repos

File type	File use	File location	Filename format	File format
<i>Schema changes</i>	Associated with a specific commit in the <i>schema repo</i> ; documents changes in the <i>schema repo</i> since the previous commit annotated by a <i>Schema changes</i> file	Stored in the migrations directory in the <i>schema repo</i> , which is automatically created if necessary	<code>schema_changes_{}_{}_yaml</code> , where the {} placeholders are replaced with the file's creation timestamp and the prefix of the commit's git hash, respectively	YAML
Custom IO classes	Load a Reader and/or Writer class from the <i>schema repo</i> that Migration will use to read and/or write files whose data models are defined by the schema; if <code>custom_io_classes.py</code> does not exist or doesn't define Reader or Writer the default <code>obj_tables.io.Reader/Writer</code> will be used	Stored in the <code><package name>/migrations</code> directory in the <i>schema repo</i> , where <code><package name></code> is the root directory of the python package in the <i>schema repo</i>	Must be called <code>custom_io_classes.py</code>	Python
Transformations	Define methods that can apply arbitrary transformation to each Model during migration	Stored in a filename relative to the migrations directory, specified by the <code>transformations_file</code> field in a <i>schema changes</i> file	Any Python file	Python

Table 1.2: The configuration file in data repos

File type	Data-schema migration configuration file
File use	Configure the migration of a set of files in a <i>data repo</i> whose data models are defined by the same schema in a <i>schema repo</i>
File location	Stored in the <code>migrations</code> directory in the <i>schema repo</i> , which is automatically created if necessary
Filename format	<code>data_schema_migration_conf--{}--{}--{}.yaml</code> , where the format placeholders are replaced with 1) the name of the <i>data repo</i> , 2) the name of the <i>schema repo</i> , and 3) a datetime value
File format	YAML

Example configuration files

This section presents examples of migration configuration files and code fragments that would be used to migrate data files from the *existing* schema to the *changed* schema above.

This example *Schema changes* file documents the changes between the *existing* and *changed* schema versions above:

```
# schema changes file
# stored in 'schema_changes_2019-03-26-20-16-45_820a5d1.yaml'

commit_hash: 820a5d1ac8b660b9bdf609b6b71be8b5fdbf8bd3
renamed_attributes: [[Test, existing_attr], [ChangedTest, migrated_attr]]
renamed_models: [[Test, ChangedTest]]
transformations_file: 'example_transformation.py'
```

All schema changes files contain these fields: `commit_hash`, `renamed_models`, `renamed_attributes`, and `transformations_file`.

- `commit_hash` is the hash of the sentinel Git commit that the Schema changes file annotates. That is, as illustrated in [Figure 1.6](#), the commit identified in the *Schema changes* file must depend on all commits that modified the schema since the closest upstream sentinel commit.
- `renamed_models` is a YAML list that documents all *Models* in the schema that were renamed. Each renaming is given as a pair of the form `[ExistingName, ChangedName]`.
- `renamed_attributes` is a YAML list that documents all attributes in the schema that were renamed. Each renaming is given as a pair in the form `[[ExistingModelName, ExistingAttrName], [ChangedModelName, ChangedAttrName]]`. If the *Model* name hasn't changed, then `ExistingModelName` and `ChangedModelName` will be the same. `transformations_file` optionally documents the name of a Python file that contains a class which performs custom transformations on all *Model* instances as they are migrated.

As shown in [Figure 1.6](#), by default, a schema changes file identifies the head commit as the sentinel commit that it annotates. However, a schema changes file may identify a sentinel commit further back in the dependency graph. The identification is implemented by storing the sentinel commit's hash in the schema changes file's `commit_hash`.

The changes between the *existing* and the *changed* schemas are separated into three commits, **a**, **b**, and **c**. **a** and **b** must both occur before **c**, because **a** and **b** both access *Model Test* whereas **c** renames *Model Test* to *Model ChangedTest*. Both alternative commit histories both satisfy these constraints.

Template schema changes files are generated by the CLI command `make-changes-template`, as described below.

This example *transformations* file contains a class that converts the floats in attribute `Test.size` into ints:

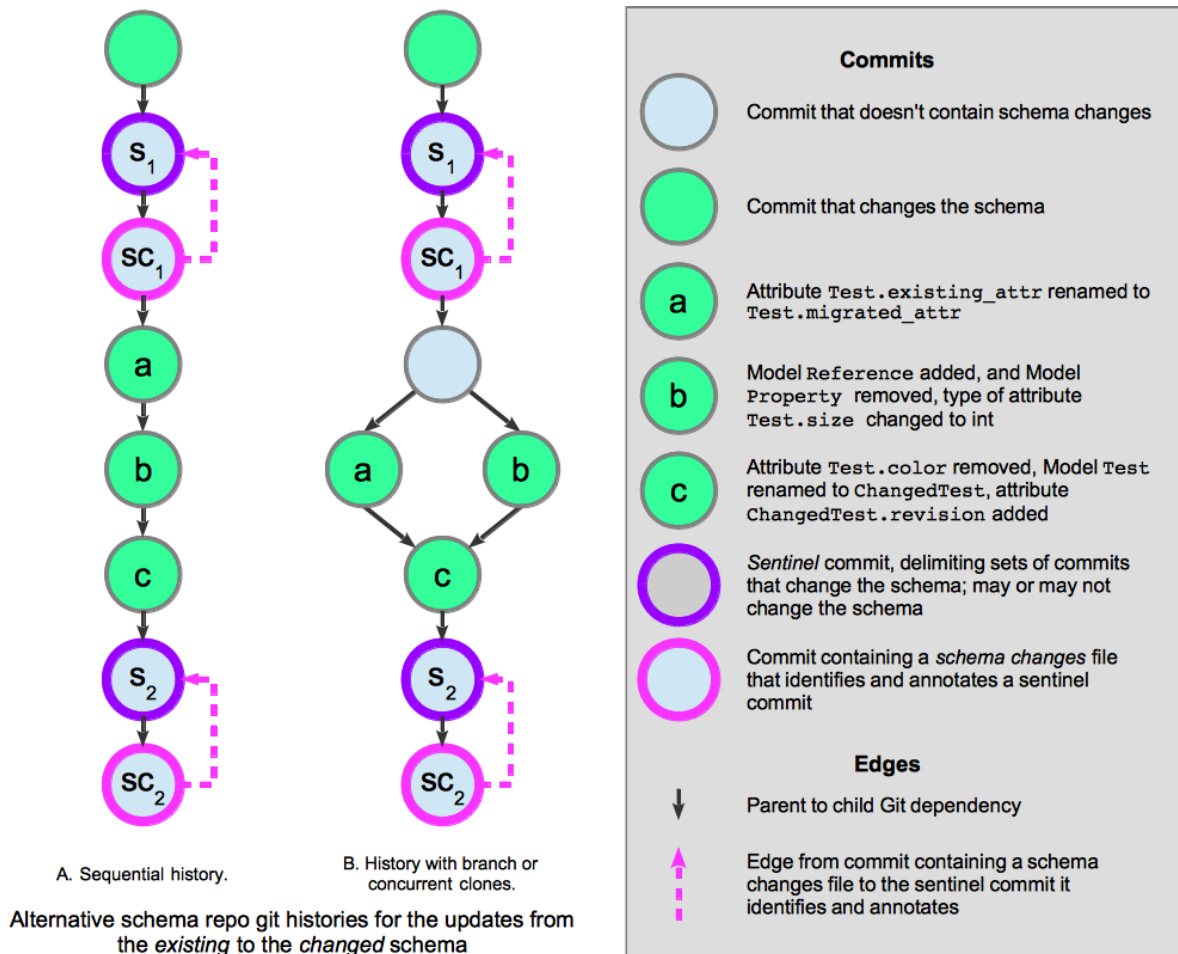


Figure 1.6: Example schema changes for the updates from the *existing* to the *changed* schema above. This figure illustrates alternative Git histories of the schema changes and migration annotations that could occur when updating the schema repo to reflect the changes between the *existing* and the *changed* schemas above.


```
# an example transformations program
from obj_tables.migrate import MigrationWrapper, MigratorError

class TransformationExample(MigrationWrapper):

    def prepare_existing_models(self, migrator, existing_models):
        """ Prepare existing models before migration

        Convert ``Test.size`` values to integers before they are migrated

        Args:
            migrator (:obj:`Migrator`:) the :obj:`Migrator` calling this method
            existing_models (:obj:`list` of :obj:`obj_tables.Model`:) the models
                that will be migrated
        """
        try:
            for existing_model in existing_models:
                if isinstance(existing_model, migrator.existing_defs['Test']):
                    existing_model.size = int(existing_model.size)
        except KeyError:
            raise MigratorError("KeyError: cannot find model 'Test' in existing_
↳ definitions")

    def modify_migrated_models(self, migrator, migrated_models):
        """ Modify migrated models after migration

        Args:
            migrator (:obj:`Migrator`:) the :obj:`Migrator` calling this method
            migrated_models (:obj:`list` of :obj:`obj_tables.Model`:) all models
                that have been migrated
        """
        pass

# a MigrationWrapper subclass instance must be assigned to :obj:`transformations`
transformations = TransformationExample()
```

Transformations are subclasses of `obj_tables.migrate.MigrationWrapper`. *Model* instances can be converted before or after migration, or both. The `prepare_existing_models` method converts models before migration, while `modify_migrated_models` converts them after migration. Both methods have the same signature. The `migrator` argument provides an instance of `obj_tables.migrate.Migrator`, the class that performs migration. Its attributes provide information about the migration. E.g., this code uses `migrator.existing_defs` which is a dictionary that maps each *Model*'s name to its class definition to obtain the definition of the *Test* class.

This example `custom_io_classes.py` file configures a migration of files that use the `wc_lang` schema to use the `wc_lang.io.Reader`:

```
""" Import wc_lang's Reader so it can be used by obj_tables migration
    of wc_lang model files """

# this file is imported by obj_tables/migrate.py
from wc_lang.io import Reader # noqa: F401
```

In general, a `custom_io_classes.py` file will be needed if the *schema repo* defines its own `Reader` or `Writer` classes for data file IO.

This example *data-schema migration configuration* file configures the migration of one file, `data_file_1.xlsx`.

```
# data-schema migration configuration file

# description of the attributes:
# 'files_to_migrate' contains paths to files in the data repo to migrate
# 'schema_repo_url' contains the URL of the schema repo
# 'branch' contains the schema's branch
# 'schema_file' contains the relative path to the schema file in the schema repo
files_to_migrate: [../data/data_file_1.xlsx]
schema_file: '../obj_tables_test_migration_repo/core.py'
schema_repo_url: 'https://github.com/KarrLab/obj_tables_test_migration_repo'
branch: 'master'
```

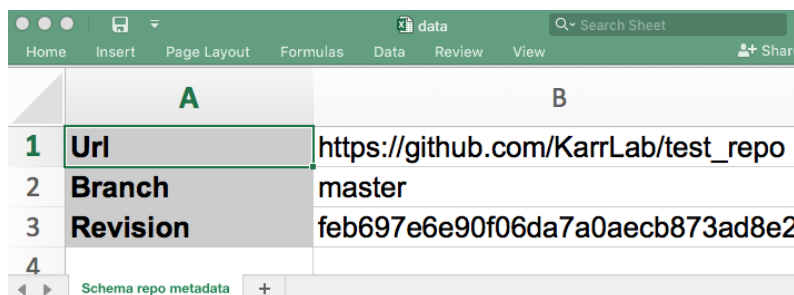
All data-schema migration config files contain four fields:

- `files_to_migrate` contains a list of paths to files in the data repo that will be migrated
- `schema_repo_url` contains the URL of the schema repo
- `branch` contains the schema repo's branch
- `schema_file` contains the path of the schema file in the schema repo relative to its URL

Migration commands create data-schema migration configuration and schema changes files, as listed in [Table 1.3](#) below.

Schema Git metadata in data files

Each data file in the *data repo* must contain a *Model* that documents the version of the *schema repo* upon which the file depends. For migration to work properly this version must be a sentinel commit in the schema repo. This Git metadata is stored in a *SchemaRepoMetadata Model* (which will be in a *Schema repo metadata* worksheet in an XLSX file). The metadata specifies the schema's version with its URL, branch, and commit hash. A migration of the data file will start at the specified commit in the *schema repo*. An example Schema repo metadata worksheet in an XLSX file is illustrated below:



	A	B
1	Url	https://github.com/KarrLab/test_repo
2	Branch	master
3	Revision	feb697e6e90f06da7a0aecb873ad8e2
4		

Figure 1.7: Example Schema repo metadata worksheet in an XLSX data file. This schema repo metadata provides the point in the schema's commit history at which migration of the data file would start.

Migration migrates a data file from the schema commit identified in the file's schema's Git metadata to the last sentinel commit in the *schema repo*.

1.2.4 Topological sort of schema changes

The migration of a data file modifies data so that its structure is consistent with the schema changes saved in Git commits in the schema repo. Because the dependencies between commits cannot be circular, the dependency graph of commits is a directed acyclic graph (DAG).

Migration executes this algorithm:

```
def migrate_file(existing_filename, migrated_filename, schema_repo):
    """ Migrate the models in `existing_filename` according to the
        schema changes in `schema_repo`, and write the results in `migrated_filename`.
    """

    # get_schema_commit() reads the Schema repo metadata in the file,
    # and obtains the corresponding commit
    starting_commit = get_schema_commit(existing_filename)
    # obtain the schema changes that depend on `starting_commit`
    schema_changes = schema_repo.get_dependent_schema_changes(starting_commit)

    # topologically sort schema_changes using dependencies in the schema repo's
    ↪ commit DAG
    ordered_schema_changes = schema_repo.topological_sort(schema_changes)
    existing_models = read_file(filename)
    existing_schema = get_schema(starting_commit)

    # iterate over the topologically sorted schema changes
    for schema_change in ordered_schema_changes:
        end_commit = schema_change.get_commit()
        migrated_schema = get_schema(end_commit)
        # migrate() migrates existing_models from the existing_schema to the migrated_
        ↪ schema
        migrated_models = migrate(existing_models, existing_schema, migrated_schema)
        existing_models = migrated_models
        existing_schema = migrated_schema

    write_file(migrated_filename, migrated_models)
```

A **topological sort** of a DAG finds a sequence of nodes in the DAG such that if node X transitively depends on node Y in the DAG then X appears after Y in the sequence. Topological sorts are non-deterministic because node pairs that have no transitive dependency relationship in the DAG can appear in any order in the sequence. For example, a DAG with the edges $A \rightarrow B \rightarrow D$, $A \rightarrow C \rightarrow D$, can be topologically sorted to either $A \rightarrow B \rightarrow C \rightarrow D$ or $A \rightarrow C \rightarrow B \rightarrow D$.

Sentinel commits must therefore be selected such that *any* topological sort of them produces a legal migration. We illustrate incorrect and correct placement of sentinel commits in [Figure 1.8](#).

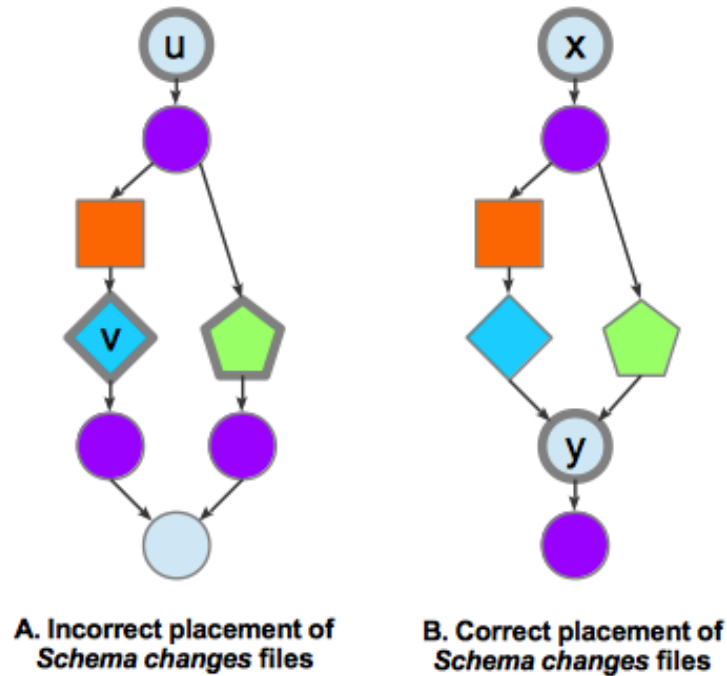


Figure 1.8: Placement of schema changes commits in a Git history (this figure reuses the legend in [Figure 1.6](#)). Migration topologically sorts the commits annotated by the schema changes files (indicated by thick outlines). In **A**, since the blue diamond commit and green pentagon commit have no dependency relationship in the Git commit DAG, they can be sorted in either order. This non-determinism is problematic for a migration that uses the commit history in **A**: if the diamond commit is sorted before the pentagon commit, then migration to the pentagon commit will fail because it accesses *Model Test* which will no longer exist because migration to the diamond commit renames *Test* to *ChangedTest*. No non-determinism exists in **B** because the commits annotated by the schema changes files – *x* and *y* – are related by $x \rightarrow y$ in the Git commit DAG. A migration of **B** will not have the problem in **A** because the existing *Models* that get accessed by the transformation above will succeed because it uses the schema defined by the top commit.

1.2.5 Migration protocol

As discussed above, using migration involves creating configuration files at various times in the schema and data repos, and then migrating data files. This section summarizes the overall protocol users should follow to migrate data.

Configuring migration in a schema repository

Schema builders are responsible for these steps.

1. Make changes to the schema, which may involve multiple commits and multiple branches or concurrent repository clones
2. Confirm that the schema changes work and form a set of related changes
3. Git commit and push the schema changes; the last commit will be a *sentinel commit*
4. Use the `make-changes-template` command to create a template schema changes file
5. Determine the ways in which *Models* and attributes were renamed in step 1 and document them in the template schema changes file
6. Identify any other model changes that require a transformation (as shown in [Figure 1.3](#)); if they exist, create and test a transformations module, and provide its filename as the `transformations_file` in the schema changes file
7. Git commit and push the schema changes file, and transformations module, if one was created
8. Test the new schema changes file by migrating a data file that depends (using its schema Git metadata as in [Figure 1.7](#)) on the version of the schema that existed before the changes in step 1

While this approach identifies sentinel commits and creates template schema changes files immediately after the schema has been changed, that process can be performed later, as

Migration of data files in a data repository

People who use *ObjTables* schemas, such as whole-cell modelers, should follow one of these sequences of steps to migrated files.

Migrate arbitrary data files

1. Decide to migrate some data files
2. Git commit and push the data repo to backup all data files on the Git server
3. Use the `migrate-data` command to migrate the files; the migrated files will overwrite the initial existing files

Use a data-schema migration configuration file to migrate data files

1. Decide to migrate some data files
2. If a *data-schema migration configuration* file for the files does not exist, use the `make-data-schema-migration-config-file` command to make one
3. Git commit and push the data repo to backup all data files on the Git server
4. Use the `do-configured-migration` command to migrate the files; the migrated files will overwrite the initial existing files

1.2.6 Using migration commands

Migration commands are run via the wholecell command line interface program `wc-cli` on the command line. As listed in Table 1.3, different commands are available for *schema repos* and *data repos*.

Table 1.3: Migration commands

Repository	Command	Purpose
schema	<code>make-changes-template</code>	Create a template <i>schema changes</i> file
data	<code>migrate-data</code>	Migrate specified data files (without using a <i>data-schema migration configuration</i> file)
data	<code>do-configured-migrate</code>	Migrate the data files specified in a data-schema migration config file
data	<code>make-data-schema-migration-config-file</code>	Create a <i>data-schema migration configuration</i> file

Schema repo migration commands

`wc_lang` (abbreviated *lang*) is a schema repo. All schema repos will support this command.

The `make-changes-template` command creates a template *Schema changes* file. By default, it creates a *Schema changes* template in the schema repo that contains the current directory. To use another schema repo, specify a directory in it with the `--schema_repo_dir` option.

By default, the *Schema changes* template created identifies the most recent commit in the schema repo as a sentinel commit. To have the *Schema changes* file identify another commit as the sentinel, provide its hash with the `--commit` option. This makes it easy to add a schema changes file that identifies an older commit as a sentinel commit, after making other commits downstream from the sentinel.

`make-changes-template` initializes `commit_hash` in the template as the sentinel commit's hash. The hash's prefix also appears in the file's name. The format of the fields `renamed_models`, `renamed_attributes`, and `transformations_file` is written, but their data must be entered manually.

```
usage: wc-cli tool lang make-changes-template [-h]
                                           [--schema_repo_dir SCHEMA_REPO_DIR]
                                           [--commit COMMIT]

Create a template schema changes file

optional arguments:
  --schema_repo_dir SCHEMA_REPO_DIR
                        path of the directory of the schema's repository;
                        defaults to the current directory
  --commit COMMIT      hash of a commit containing the changes; default is
                        most recent commit
```

Data repo migration commands

`wc_sim` (abbreviated *sim*) is a data repo. All data repos will support the same commands.

The `make-data-schema-migration-config-file` command creates a data-schema migration configuration file. It must be given the full URL of the Python schema file in its Git repository, including its branch. For example `https://github.com/KarrLab/wc_lang/blob/master/wc_lang/core.py` is the URL of the schema in `wc_lang`. It must also be given the absolute or relative path of at least one data file that will be migrated when the data-schema migration config file is used. The config file can always be edited to add, remove or changes data files.

By default, `make-data-schema-migration-config-file` assumes that the current directory is contained in a clone of the data repo that will be configured in the new migration config file. A different data repo can be specified by using the `--data_repo_dir` option.

```
usage: wc-cli tool sim make-data-schema-migration-config-file
       [-h] [--data_repo_dir DATA_REPO_DIR]
       schema_url file_to_migrate [file_to_migrate ...]

Create a data-schema migration configuration file

positional arguments:
  schema_url            URL of the schema in its Git repository,
                        including the branch
  file_to_migrate       a file to migrate

optional arguments:
  --data_repo_dir DATA_REPO_DIR
                        path of the directory of the repository storing the
                        data file(s) to migrate; defaults to the current
                        directory
```

The `do-configured-migration` command migrates the data files specified in a data-schema migration config file. Each data file that's migrated is replaced by its migrated file.

```
usage: wc-cli tool sim do-configured-migration [-h] migration_config_file

Migrate data file(s) as configured in a data-schema migration
configuration file

positional arguments:
  migration_config_file
                        name of the data-schema migration configuration file to use
```

The `migrate-data` command migrates specified data file(s). Like `make-data-schema-migration-config-file`, it must be given the full URL of the Python schema file in its Git repository, including its branch, and the absolute or relative path of at least one data file to migrate. By default, `migrate-data` assumes that the current directory is contained in a clone of the data repo that contains the data files to migrate. A different data repo can be specified by using the `--data_repo_dir` option. Each data file that's migrated is replaced by its migrated file.

```
usage: wc-cli tool sim migrate-data [-h] [--data_repo_dir DATA_REPO_DIR]
                                     schema_url file_to_migrate
                                     [file_to_migrate ...]

Migrate specified data file(s)

positional arguments:
  schema_url            URL of the schema in its Git repository,
                        including the branch
  file_to_migrate       a file to migrate

optional arguments:
  --data_repo_dir DATA_REPO_DIR
                        path of the directory of the repository storing the
                        data file(s) to migrate; defaults to the current
                        directory
```

Practical considerations

The user must have access rights that allow them to clone the data repo and schema repo.

1.2.7 Known limitations

As of August 2019, the implementation of migration has these limitation:

- Migration requires that schemas and data files be stored in Git repositories – no other version control systems are supported.
- Only one schema file per schema repo is supported.
- Migration of large data files runs slowly.
- Options that store a migrated file in a different location than its data file are not exposed at the command line.

1.3 About

1.3.1 License

The software is released under the MIT license

```
The MIT License (MIT)
```

```
Copyright (c) 2017-2020 ObjTables developers
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy  
of this software and associated documentation files (the "Software"), to deal  
in the Software without restriction, including without limitation the rights  
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
copies of the Software, and to permit persons to whom the Software is  
furnished to do so, subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in all  
copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR  
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE  
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER  
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,  
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE  
SOFTWARE.
```


1.3.2 Development team

This package was developed by the [Karr Lab](#) at the Icahn School of Medicine at Mount Sinai in New York, US and the [Applied Mathematics and Computer Science, from Genomes to the Environment research unit](#) at the National Research Institute for Agriculture, Food and Environment in Jouy en Josas, FR.

- [Jonathan Karr](#)
- [Arthur Goldberg](#)
- [Wolfram Liebermeister](#)
- [John Sekar](#)
- [Bilal Shaikh](#)

1.3.3 Acknowledgements

This work was supported by a National Institute of Health P41 award (P41EB023912), a National Institute of Health MIRA R35 award (R35GM119771), and a National Science Foundation INSPIRE award (1649014).

1.3.4 Questions and comments

Please contact the [developers](#) with any questions or comments.