Extending the FLINT framework to support Monte Carlo and Propagation of Error Uncertainty

February 15, 2020

Abstract

Uncertainty assessment is important for land based greenhouse gas estimation, as it is incorporated in all aspects of reporting, from national greenhouse gas inventories through to results-based payments for REDD+. The purpose of this design document is to describe the FLINT framework enhancements required for implementing two commonly used IPCC methods for combining uncertainties. These are the Propagation of Error and Monte Carlo methods for combining uncertainties. A key design requirement is that the methods are consistent with the 2006 IPCC Guidelines for national greenhouse gas inventories, and are also capable of supporting other common land sector MRV programs and reporting requirements.

Introduction

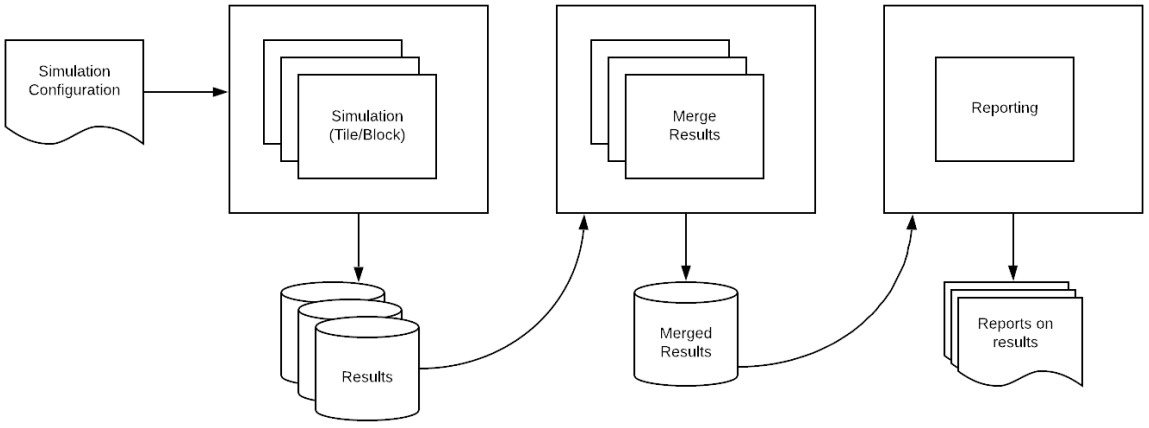
Following on from the initial design document (Design for Monte Carlo and Propagation of Error Uncertainty in the Full Lands Integration Tool), this document's purpose is to detail changes required in the Full Lands Integration Tool (FLINT) framework to support a generic (but extensible) method to run Monte Carlo analysis during a FLINT simulation.

The FLINT framework is designed to run Simulations at large scale, in some cases at National level over 25m2 land units. While a full Simulation can always be run in a single instance, the time required to complete may be significant. The solution is to run distributed processing by breaking the Simulation area into defined chunks. This has been handled by using a [Tile, Block, Cell] index to reference Land Units, and group processing of these Land Units at either a [Tile, Block] or [Tile] level.

Once distributed processing is introduced, the requirement to merge results across chunks becomes necessary. All raw data could be maintained, and no aggregation applied. Perhaps writing results directly from each chunk into a common relational database directly. However, as with Simulation time (and depending on the scale of Simulation) the storage required could be significant.

One solution applied in current FLINT implementations is shown in the workflow Distributed processing using the FLINT framework.

The extensions to FLINT in this document focus on the Distributed method, but could also be applied to Simulating in a single instance.



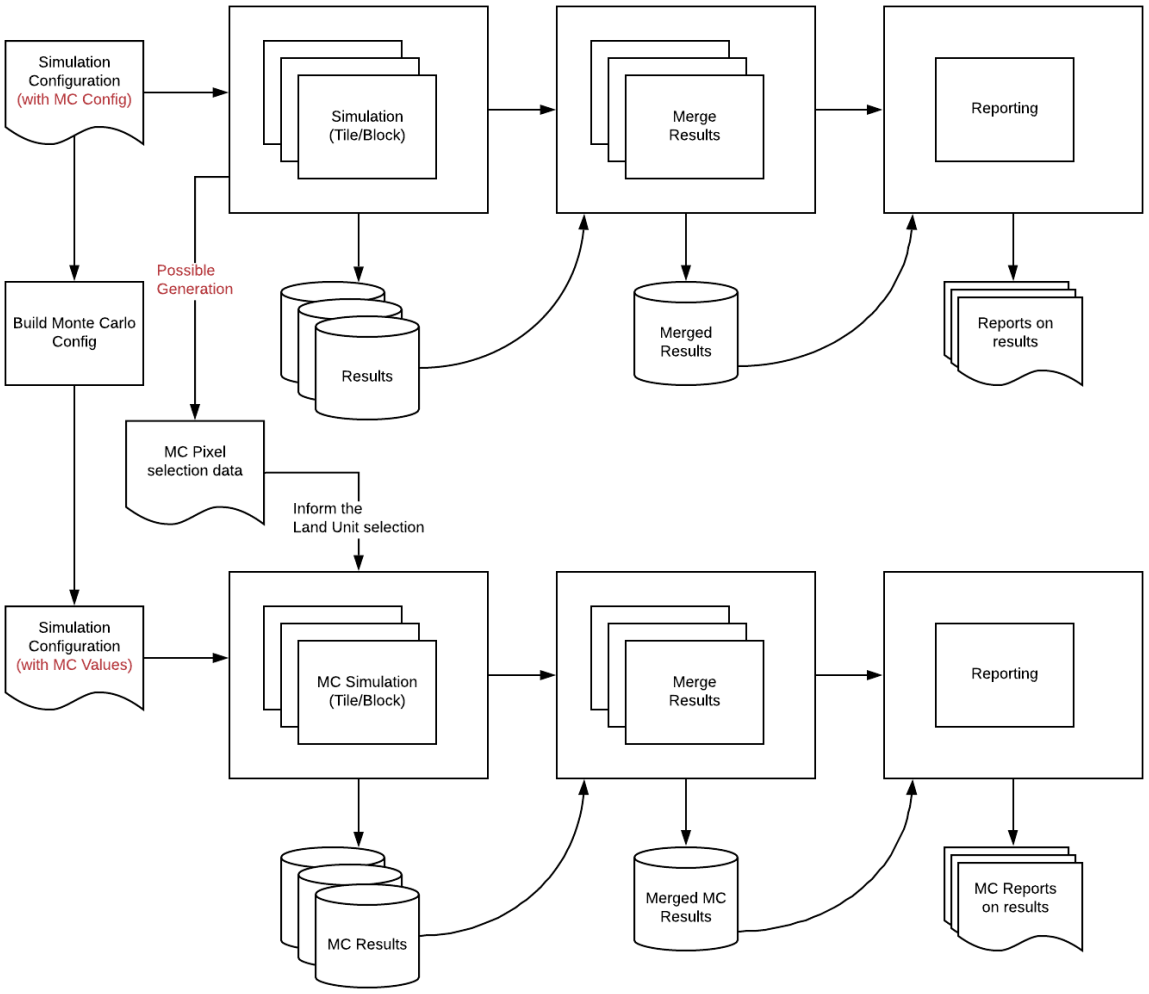
Distributed processing using the FLINT framework

Monte Carlo FLINT Processing

There are 3 workflows that this design in this document will address to handle distributed processing for generation of Monte Carlo results.

Workflow 1 - Separate processing

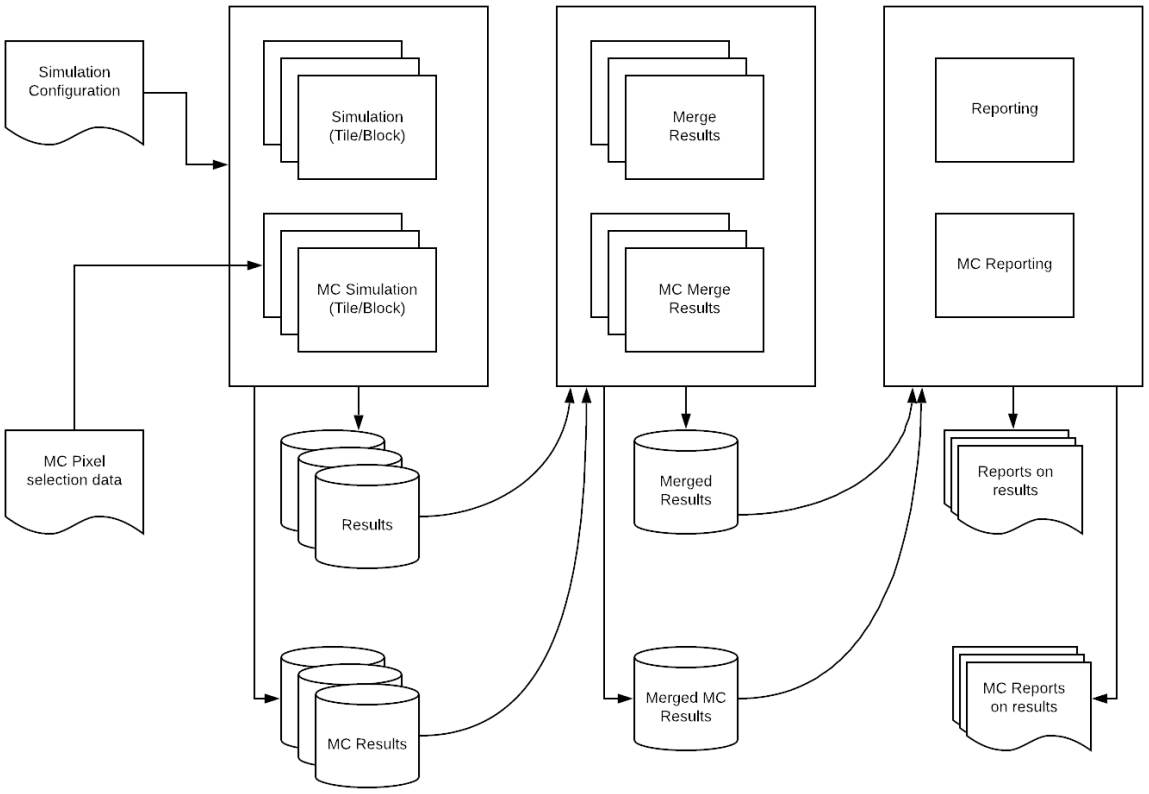
During this process the selection for Land Units to include in the Monte Carlo processing can be made. This Land Unit selection dataset can then be used to run the distributed Monte Carlo Simulation/Merge process.



Separate processing

Workflow 2 - Combined processing

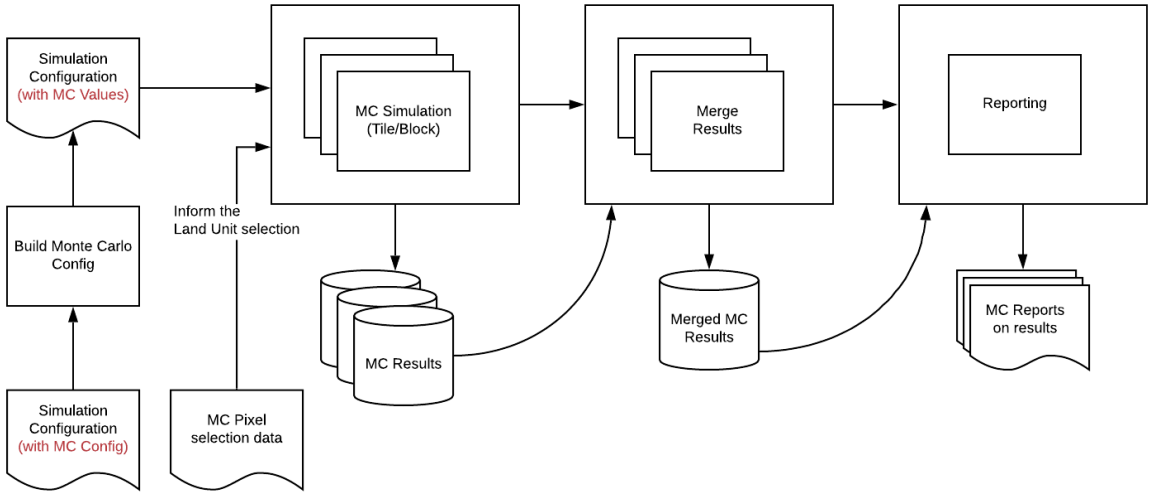
If the Land Unit selection method can be generated without running the full simulation (e.g. a spatial mask file OR database has been pre-generated to indicate selection across the simulation landscape) then both processes can be run at the same time.



Combined processing

Workflow 3 - Only Monte Carlo processing

In some situations the normal Simulation run will not be required, just the Monte Carlo Simulation. This is the simplest case, but means the Monte Carlo mask cannot be generated during the normal Simulation.



Only Monte Carlo processing

FLINT Extensions for Monte Carlo

This section will outline the additions and modifications to the existing FLINT framework that are required to run Monte Carlo Simulations.

Simulation Configuration file

There will be some top level sections added to the FLINT Simulation configuration file. The "*Modules*" section will also require some modifications to support Modules that don't run during the Monte Carlo Simulation, and some that only run during it.

Monte Carlo settings

This section will define some of the Monte Carlo simulation wide values.

```
 "MonteCarlo": {
    "landUnitSelectionVariable": "monte_carlo_selection_variable",
    "iteration_variable_name": "monte_carlo_iteration_variable",
    "iterations": 1000,
    "type": "montecarlo_spatial_tiled"
},
```

Monte Carlo configuration

Monte Carlo variable settings

This section will define the variables that will be modified during Monte Carlo simulations, and how the values will be generated.

The initial part of the configuration will have the type of variable that we are trying to replace each iteration of a Monte Carlo simulation. The values section will be generated or manually entered.

```
"MonteCarlo_Variables": {
    {
        "tree_age_1": {
            "transform": {
                "monte_carlo_type": "Bounded",
                "settings": {
                    "upper": 100,
                    "lower": 1
                },
                "data_type": "Int",
                "data_adjustments_type": "Replace",
                "library": "internal.flint",
                "type": "MonteCarloScalarTransform",
                "values": []
            }
        }
    },
    {
        "tree_age_2": {
            "transform": {
```

```
                "monte_carlo_type": "Distribution",
                "settings": {
                    "dist_type": “dist_type_1”
                    "dist_setting_1": “1122.33”
                },
                "data_type": "Double",
                "data_adjustments_type": “Replace”,
                "library": "internal.flint",
                "type": "MonteCarloScalarTransform",
                "values": []
            }
        }
    },
    {
        "tree_properties": {
            "transform": {
                "monte_carlo_type": "Distribution",
                "settings": {
                    "dist_type": “dist_type_1”
                    "dist_setting_1": “10.1”
                },
                "data_type": "Int",
                "data_pair_index": 2,
                "data_adjustments_type": “Replace”,
                "library": "internal.flint",
                "type": "MonteCarloDictionaryTransform",
                "values": []
            }
        }
    },
    {
        "tree_list": {
            "transform": {
                "monte_carlo_type": "Distribution",
                "settings": {
                    "dist_type": “dist_type_1”
                    "dist_setting_1": “10.2”
                },
                "data_type": "Int",
                "data_pair_index": 3,
                "data_adjustments_type": “Offset”,
                "library": "internal.flint",
                "type": "MonteCarloDictionaryListTransform",
                "values": []
            }
        }
    },
},
```

Example Monte Carlo variable configuration

Transforms

There are 4 new Transforms to be defined. One for each data type we expect to be returning:

- MonteCarloScalarTransform
- MonteCarloDictionaryTransform
- MonteCarloDictionaryListTransform
- MonteCarloTimeseriesTransform

These will all take the same settings as defined below.

Common Settings

*“**data_type**”*: Scalar type of the identified variable. Valid values:

Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Float, Double

"***data_adjustments_type***": Valid values:

“Replace” - variables will be replaced with the value for the current iteration.
“Offset” - variables will be offset with the value for the current iteration.

“**monte_carlo_type**”: Valid values:

“***Bounded**”:* - Would require - “upper” and “lower” values corresponding to the data type defined in the main body of the Transform.

Settings required:

```
"settings": {
    "upper": 100,
    "lower": 1
},
```

“***Distribution***” - Multiple distribution types will be implemented. The full list to be decided during implementation. However, as these will be handled with Transforms it will be easily extended.

“***Manual***” - The user will be required to enter the correct number of values to substitute during a Monte Carlo Simulation.

The Transforms can wrap the original, using it to gather the normal data as it would appear without Monte Carlo, then adjust as defined by the type set and the array of replacement values/offsets already generated.

Monte Carlo Modules section

This section will define the Monte Carlo only Modules to be run. These Modules will be run in addition to the normally defined Modules.

There will be a flag added that allows normal Modules to be excluded during the Monte Carlo Simulation. This will be a user decision, driven by a configuration flag. This would normally be (but not limited to) special output handling, not modules that are involved in generating fluxes.

```
"MonteCarlo_Modules": {
   "AggregatorAndWriteMonteCarlo": {
        "enabled": true,
        "library": "internal.flint",
        "order": 1,
        "monte_carlo_status": "MonteCarlo"
```

```
    },
```

Below is a sample of the flag to exclude a Module from a Monte Carlo Simulation (the default is to include).

```
"Modules": {
    "AggregatorStock": {
        "enabled": true,
        "library": "internal.flint",
        "order": 70,
        "monte_carlo_exclude": True
    },
    "ErrorScreenWriter": {
        "enabled": true,
        "library": "internal.flint",
        "order": 72
    },
```

Simulation module exclude flag

Build Monte Carlo Configuration

If any of the Monte Carlo values require generation, the configuration file will need to be run through a generation process. This executable will read the defined Monte Carlo Variables and generate the value required for the iterations of simulations.

The assumption here is that each distributed part of the processing requires the same generated values for the Monte Carlo variables, hence, the numbers need to be pre-prepared.

The program arguments will be:

```
Allowed options:

Info options:
  -h [ --help ]          produce a help message
  -v [ --version ]       output the version number

Run options:
  -i [ --input_config_file ] arg    path to Moja run config file
  -o [ --output_config_file ] arg   path to Moja run config file
```

The output file will be the same as the original, with the values section filled in where required. The example below shows a Distribution with the value populated after running the program (Note: Iterations/number of values is driven by the MonteCarlo config section - assuming 5 for this sample).

```
{
    "tree_age_2": {
        "transform": {
            "monte_carlo_type": "Distribution",
            "settings": {
                "dist_type": "dist_type_1"
                "dist_setting_1": "1122.33"
```

```
            },
            "data_type": "Double",
            "data_adjustment": "Direct",
            "library": "internal.flint",
            "type": "MonteCarloScalarTransform",
            "values": [2.22, 2.31, 4.45, 5.67, 7.89]
        }
    }
},
```

Land Unit Selection for Monte Carlo Simulations

The decision to start the Monte Carlo Simulation for any given Land Unit will be driven by the setting in the Configuration file (Monte Carlo FLINT Processing). By nominating a variable that will return a Boolean value. How this variable is derived can be easily modified. For example:

1. Spatial layer containing True/False values for each Land Unit simulated (commonly called a mask file). This variable could use the existing transform to read values from a spatial layer:

```
{
    "monte_carlo_selection_variable": {
        "transform": {
            "data_id": "monte_carlo_selection",
            "library": "internal.flint",
            "provider": "FlintTiled",
            "type": "LocationIdxFromFlintDataTransform"
        }
    }
},
```

2. Database lookup that would perform a query using the current location (and any other useful information) to decide.

```
{
    "monte_carlo_selection_variable": {
        "transform": {
            "library": "moja.modules.mulliongroup.base",
            "provider": "SQLite",
            "queryString": "select should_i_monte_carlo from monte_carlo_data_table where
tile_id = {var:tile_id} and block_id = {var:block_id} and cell_id = {var:cell_id};",
            "type": "SQLQueryTransform"
        }
    }
},
```

3. Custom transform (Written in C++ FLINT module) that makes the selection based on calculations. An example of configuration for this would be:

```
{
   "monte_carlo_selection_variable": {
       "transform": {
           "library": "moja.modules.mulliongroup.montecarlo",
           "type": "MonteCarloLandUnitSelectorVersion1",
           "some_settings": "whatever is required"
       }
```

```
    }
},
```

By having the Land Unit selection driven by a FLINT defined variable the system gains both flexibility and extensibility (as additional methods of Land Unit selection can be developed independently).

Monte Carlo Local Domain Controller

The Local Domain Controller concept (LDC) in the FLINT framework is a construct used to control the processing on a defined set of Land Units. The implemented version used for Spatial Tiled Simulations is *moja::flint::SpatialTiledLocalDomainController* (ST-LDC).

The *ST-LDC* iterates over a defined set of [Tile, Block, Cell] indexes and performs Simulations driven by the FLINT Simulation configuration file.

Ultimately, how the results (Pool values and Carbon movements) are recorded is dependent on the Modules configured in the FLINT Simulation configuration file. Generally there would also be some level of aggregation of Pool and Flux information over the simulation date range, often including some other spatially explicit information as dimensions of the result set (e.g. Geographical region or Ecological Zone).

For the Monte Carlo extension a new Local Domain Controller will be implemented. This controller will work in a similar fashion to the *moja::flint::SpatialTiledLocalDomainController*. It will be called the ***moja::flint::MonteCarloSpatialTiledLocalDomainController*** (MC-LDC).

The key features of the MC-LDC are that it will use the Monte Carlo section of the FLINT simulation configuration file (Configuration Extensions) to adjust the Simulations for each of the number of iterations defined. The set of [Tile, Block, Cell] combinations selected to run will be the same, non-Monte Carlo Land Units will be masked out.

The MC-LDC will read and use the Monte Carlo configurations sections as defined in Simulation Configuration file to drive the Iterations, Simulations and Variable replacement.

On each iteration, the MC-LDC will set the iteration variable (as defined in the Monte Carlo settings) and adjust the values for the defined variables (Monte Carlo variable configuration).

Random seeds

The FLINT generates random seeds at a Global, Tile, Block and Cell level. These provide a reproducible way to generate random numbers when required within Modules.

This enables runs that use randomness, to reproduce runs by seeding with the same random seed used previously.

Each iteration of a Monte Carlo Simulation for a given Land Unit, will use the same Random Seeds. This will be handled by the MC-LDC.

Moja command line executable

The command line tool (CLI) will require some extensions to handle the new Monte Carlo additions. The CLI is used to run Simulations, it will need to be extended to run simulations using the new Monte Carlo Local Domain Controller (MC-LDC)

The current command line options are:

```
Allowed options:

General options:
  -h [ --help ]          produce a help message
  --help-section arg     produce a help message for a named section
  -v [ --version ]       output the version number

Commandline only options:
  --logging_config arg   path to Moja logging config file
  --config_file arg      path to Moja run config file
  --provider_file arg    path to Moja data provider config file

Configuration file options:
  --config arg           path to Moja project config files
  --config_provider arg  path to Moja project config files for data providers
```

The proposed changes will add 2 options to the *'commandline'* only section.

```
Allowed options:

General options:
  -h [ --help ]          produce a help message
  --help-section arg     produce a help message for a named section
  -v [ --version ]       output the version number

Commandline only options:
  --logging_config arg   path to Moja logging config file
  --config_file arg      path to Moja run config file
  --provider_file arg    path to Moja data provider config file
  --doSim arg            True/False flag to run the normal Simulation [Default True]
  --doMonteCarlo arg     True/False flag to run the Monte Carlo Simulation [Default False]

Configuration file options:
  --config arg           path to Moja project config files
  --config_provider arg  path to Moja project config files for data providers
```

The program will also need to handle the new Module property `***monte_carlo_exclude***` *(see Monte Carlo Modules section*).

Flux recording and aggregation

During a Simulation, while it is possible to record each and every flux, generally this is not practical. For the purpose of the Monte Carlo Simulations the records will be aggregated to reduce data size. The dimensions we are interested in capturing are listed in Record definition in JSON.

```
{
   "monte_carlo_flux_record": {
       "id": 10101,
       "year": 1999,
       "source_pool": "atmosphereCM",
       "destination_pool": "soilOrganicCM",
       "flux_value": 1.194643,
       "lu_count": 9846,
       "iteration": 756
   }
}
```

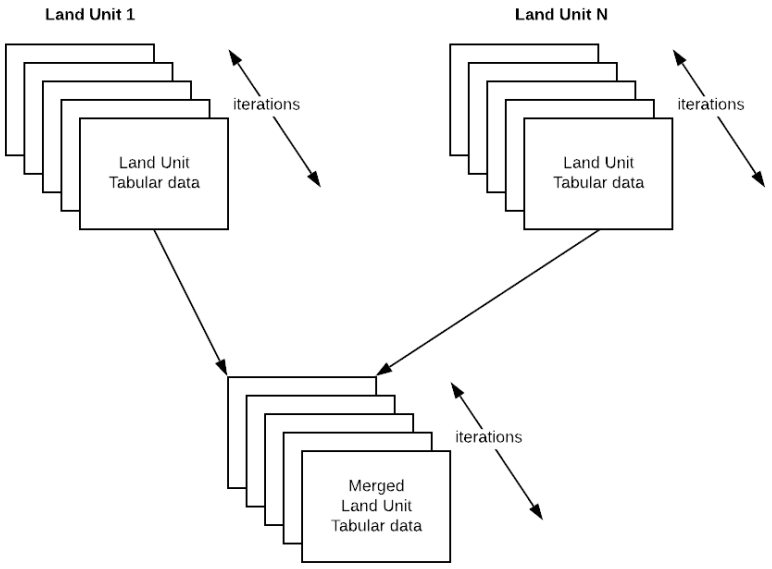
Sample flux record definition in JSON

The record structure will be used to record each flux in the Simulation, new records that match and existing record (on "*year*", "*source_pool*", "*destination_pool*" and "*iteration*") will be aggregated. Meaning the "*lu_count*" and "*flux_value*" will be added to the existing record.

id	year	source_pool	destination_pool	flux_value	lu_count	iteration
101	1999	atmosphereCM	soilOrganicCM	1.194643	9846	1
102	1999	atmosphereCM	soilOrganicCM	1.6564	1111	2
103	1999	atmosphereCM	soilOrganicCM	1.5456	233	3
104	1999	atmosphereCM	soilOrganicCM	1.54546	656	4
105	1999	atmosphereCM	soilOrganicCM	1.77554	543	5
106	2000	atmosphereCM	soilOrganicCM	1.5466	788	1
107	2000	atmosphereCM	soilOrganicCM	1.3456	534	2
108	2000	atmosphereCM	soilOrganicCM	1.7567	8756	3
109	2000	atmosphereCM	soilOrganicCM	1.75674	866	4
110	2000	atmosphereCM	soilOrganicCM	1.9546	455	5

Tabular example of Monte Carlo records

Tabular example of Monte Carlo records is a sample running with only 5 iterations and showing only *atmosphereCM* to *soilOrganicCM* fluxes. The data has also been aggregated to Year.



Aggregation across land units

Aggregation and Output Module

A Module will be created to record, aggregate and write results to a Database. This module will be called "***flint::moja::AggregatorAndWriterMonteCarlo***".

The Module will be simple, capturing Fluxes where required, aggregating into an internal collection. At the end of a Local Domain Processing Unit it will write the results into a SQLite Database.

The Database would be initialized and written to using these FLINT notifications:

```
signals::SystemInit             # init all collections
signals::LocalDomainShutdown    # write all records to the database
signals::SystemShutdown         # cleanup all collections and close database
```

Fluxes would be recorded on the following FLINT notifications:

```
signals::TimingPostInit         # handle init fluxes
signals::TimingEndStep          # capture all normal timing step fluxes
signals::PostDisturbanceEven    # capture event specific fluxes (mid timing step)
```

The configuration settings for Module would be:

```
"AggregatorAndWriterMonteCarlo": {
   "enabled": true,
   "library": "internal.flint",
   "order": 1,
   "settings": {
```

```
      "databasename":
"./data/results/monte_carlo/fluxes_{var:spatialLocationInfo.tileIdx}_{var:spatialLocationInfo.blockIdx}_{var:spatialLocationInfo.cellIdx}.db"
   }
},
```

Merge and Reporting

As there are currently no record identifiers to reconcile across multiple Monte Carlo results databases (i.e. other dimension tables that have unique ID's per processing chunk), the simplest method to merge results is to read each database and aggregate in the same way the original simulations did.

A new command line program will be written to read multiple database files with Monte Carlo results and merge them into a single Database.

The program arguments will be:

```
Allowed options:

Info options:
  -h [ --help ]          produce a help message
  -v [ --version ]       output the version number

Run options:
  -i [ --input_db_dir ] arg     path to the database files (*.db)
  -o [ --output_db_file ] arg   filename to write merged database
  -m [ --do_monte_carlo ] arg   Flag (True/False) to generate Monte Carlo table
```

Future Extensions

Custom Monte Carlo Transforms

To work in the Monte Carlo Simulation a Transform needs to provide a distribution of values, which can be used in multiple iterations of a Simulation.

With a defined interface for Monte Carlo Transforms, it will be possible to write additional methods to supply Distributions of values to be used in a Monte Carlo run.

The base class inherited from would need to be the interfaces used for other Monte Carlo Transforms, hence exposing required methods.

```
#ifndef MOJA_FLINT_ITRANSFORM_MONTECARLO_H_
#define MOJA_FLINT_ITRANSFORM_MONTECARLO_H_

#include "moja/flint/_flint_exports.h"

#include <moja/dynamic.h>
#include <moja/ivariable.h>

namespace moja {
    namespace datarepository {
        class DataRepository;
    }
    namespace flint {

        class ILandUnitController;

        class FLINT_API ITransformMonteCarlo : public ITransform {
            public:
            virtual ~ITransformMonteCarlo() = default;

            virtual void configure(DynamicObject config,
                                   const ILandUnitController& landUnitController,
                                   datarepository::DataRepository& dataRepository) = 0;

            virtual void generate_value_distribution() const = 0;
            virtual void set_iteration(DynamicVar&) const = 0;
            virtual void set_original_variable(IVariable&) const = 0;
            virtual const DynamicVar& value() const = 0;
        };

    }  // namespace flint
}  // namespace moja

#endif  // MOJA_FLINT_ITRANSFORM_MONTECARLO_H_
```

Example of Monte Carlo Transform Interface definition

Aggregation of Results and addition dimensions (classifiers)

In more complex systems (i.e. Canadian GCBM) it is possible that more dimensional information is required in the Monte Carlo results. The suggested pattern would be to extend the existing record structure (defined above) with the additional dimensions.

For example, if you wanted to add District and Ecological Zone information, add these as codes to the record.

```
{
   "monte_carlo_flux_record_gcbm": {
       "id": 10101,
       "year": 1999,
       "district": 1,
       "eco_zone": 99,
       "source_pool": "atmosphereCM",
       "destination_pool": "soilOrganicCM",
       "flux_value": 1.194643,
       "lu_count": 9846,
       "iteration": 756
   }
}
```

The aggregation method would need to be updated to handle the new dimensions and any merge process would also need to incorporate the updates.

Then create a new Module to handle the changes (based on ***AggregatorAndWriteMonteCarlo)***. Using this Module in your Monte Carlo Variable section in configuration.

Light pre-runs to gather variable information

More complex variables (Dictionary and Dictionary Lists) required knowledge of the structure return through the FLINT system. Using a light pre-run could enable the system to gather descriptions of these complex variables, to better inform users who are trying to set Distribution types to modify during a Monte Carlo run.