Project Owner	James Leitch
Code maintainers	Max Fellows James Leitch Malcolm Francis

PLEASE READ THIS BEFORE YOU CONTINUE:

This is an open source document. Everybody is welcome to contribute. To allow us to be efficient we have only a few rules:

- All contributors use comments to ask questions or make substantive and structural suggestions. Always suggest solutions in your comments not problems.
- Only Maintainers can resolve comments.
- All contributors make edits in "suggesting mode" (even if you are a maintainer or editor, so the other maintainers or editors can cross-check and accept your contribution.)
- Write suggested edits directly into the text in 'suggesting mode' do NOT use comments to suggest edits
- Only Editors assigned to the document can approve edits
- Spelling will be UK English
- Referencing Style is Harvard (author-date) type. Tools such as Zotero may be useful.

More info about how the moja global community works on projects can be found here.

Extending the FLINT framework to support Pool Cohorts

November 15, 2020

Abstract

The Full Lands Integration Tool (FLINT) has been developed to support the GHG inventories development and the implementation of mitigation actions in the AFOLU sector. But, some functions and features of the FLINT need to be refined or added in order to fully support the effective implementation of the enhanced transparency framework of the Paris Agreement by developing countries, in particular, with regard to GHG inventories, mitigation actions and establishment and maintenance of sustainable GHG inventory management systems.

The UNFCCC secretariat has initiated a collaboration with Moja Global to refine the FLINT and make it available to developing countries across all regions.

This module will update FLINT code to allow it to track multiple streams within carbon pools based on different sources (cohorts) over time. Currently the FLINT tracks carbon pools as a single pool based on the modules attached and initial system configuration.

The proposed extension would allow a dynamic list of cohorts to be used within any carbon pool tracked, allowing different carbon movements per cohort.

An example being tracking tree species information in decomposition. Rather than the single pool there would be a dynamic list of decomposition applied per tree species the system processes. This would allow for different decomposition rates to be applied to carbon movements per tree species defined and the apportioned carbon stock recorded by the FLINT.

Introduction

This document's purpose is to detail changes required in the Full Lands Integration Tool (FLINT) framework to support Pool Cohorts (or Nested Pools). Cohorts are the concept of a Parent pool with Children pools (including rules on how these child pool values would be propagated). In functional terms, cohorts to a biomass pool that is present for a simulation unit (pixel) that is divided into sub-pools each with distinct properties. For example, a standard pool is dead organic matter (DOM). Through cohorts, it will be possible to model the DOM produced from one forest type separate from the DOM produced by a previous forest type, for example. The results can then be aggregated and reported out just for DOM.

The FLINT framework is designed to run Carbon Simulations both at single point (non-spatial) and at large scale (i.e. National level Spatial simulations). The results from these simulations are generally carbon values - both Stock and Movement (fluxes). The construct of the pools was initially simplified to ensure the core framework was stable with the reduced complexity of cohorts.

Adding the ability to track a Pool by a dynamic list of Cohorts (or Nested Pools) will greatly improve the use and value to the results generated. For example, tracking tree species information in decomposition. Rather than the single pool there would be a dynamic list of decomposition applied per tree species the system processes. This would allow for different decomposition rates to be applied to carbon transfers per tree species defined and the apportioned carbon stock recorded by the FLINT. Giving more control on the Simulation and the ability to produce more accurate results. Importantly, the expansion to chorts requires managing the computational complexity of an nested pool structure while maintaining the core modelling requirements, such as mass balance. However, overcoming these technical challenges will be critically important for more accurate modelling land use change, in particular in areas where there are substantial changes in the pool characteristics between land uses.

Framework extensions will need to be backward compatible, i.e. allowing existing systems to continue to work as intended.

Fig 1: Existing System Overview

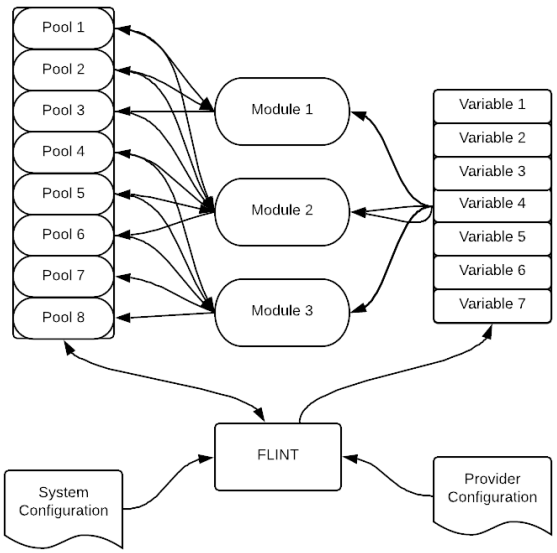
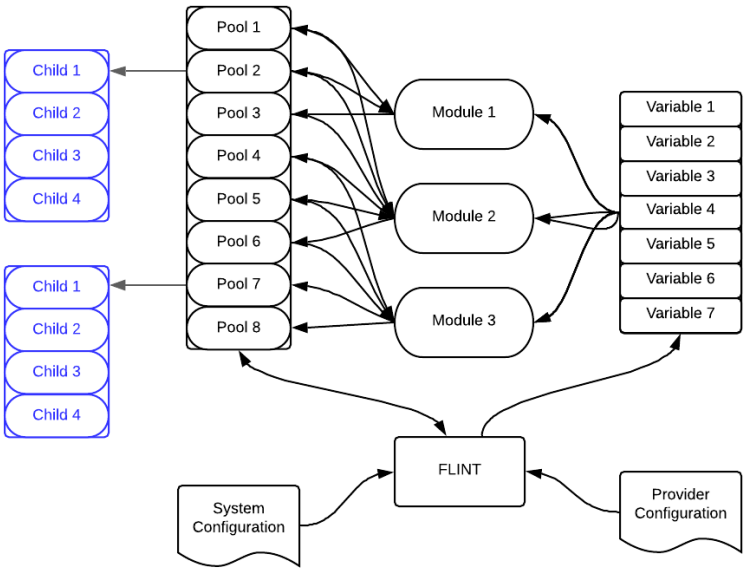


Fig 2: Proposed System



Functional Requirements

- Allow system configuration to define Children for any Pool
- Allow Modules to Dynamically add Children to any Pool during a Simulation
- Allow Modules to make Stock and Proportional transfers to and from Child Pools
- Reporting on any Pool value will aggregate the all relevant Nested Pools
- Existing Simulation Configurations should continue to run, without affecting output results
- Defining a Children Pools will not change the system mass balance
- The aggregated value of a Pool with Children will be equal to the sum of its Children

Non-Functional Requirements

- All Pool names will be unique
- Pool names will not contain Non-alphanumeric characters

FLINT Extension Summary

To extend the current framework to handle Nested Pools various changes will be required, these include:

- Configuration of Nested Pools
    - Allow Nested Pool system settings to be configured
    - Extend *moja.flint.configuration* library to handle Child Pool configuration
    - Extend processing of configuration json to handle Nested Pools
- Framework handling of Nested Pools
    - Set up Handling of Nested Pools using system settings
    - Extend *moja.flint* library to define Child Pools
    - Extend *moja.flint* library to handle transfers to and from Child Pools
    - Extend Pool value reporting to aggregate relevant Child Pools

Configuration Files

The definition of the Nested Pools will be added to the Simulation Configuration JSON file. The System Configuration loading system will also be modified to handle the additions appropriately.

Currently the JSON file defines top level components of a simulation:

- LocalDomain
- Libraries
- Pools
- Modules
- Variables

Pool Configuration

The "***Pool***" section will be extended to handle definitions of Children.

Currently there are 3 methods to define the Pools for a Simulation. The first is minimal details required. Initial pool values are set to **0.0**:

```
"Pools": [
   "initialValues",
   "atmosphereCM",
   "soilOrganicCM",
   "forestAboveGroundCM",
   "forestBelowGroundCM",
   "forestDeadOrganicMatterCM"
],
```

The second allows initial Pool values to be defined:

```
"Pools": [
   { "Pool 1": 100.0 },
   { "Pool 2": 100.0 },
   { "Pool 3": 100.0 }
],
```

The third allows more detailed information to be defined:

```
"Pools": [
   {
     “name”: "Pool 1",
     “description”: "Pool 1",
     “units”: "ha",
     “scale”: 1.0,
     “order”: 1,
     “value”: 100.0
   },
   {
     “name”: "Pool 2",
     “description”: "Pool 2",
     “units”: "ha",
     “scale”: 1.0,
     “order”: 2,
     “value”: 200.0
   }
],
```

To allow a pool to be defined as a Child it will allow the following addition:

```
"Pools": [
   {
     "name": "Pool 1",
     "description": "Pool 1",
     "units": "ha",
     "scale": 1.0,
     "order": 1,
     "value": 100.0
   },
   {
     "name": "Pool 2",
     "description": "Pool 2",
     "units": "ha",
     "scale": 1.0,
     "order": 2,
```

```
      "value": 200.0
    },
    {
      "name": "Pool 2 Child 1",
      "description": "Child Example 1",
      "units": "ha",
      "scale": 1.0,
      "order": 2,
      "value": 10.0,
      "parent_pool_name": "Pool 2"                <<---- Addition here ---->>
    }
],
```

The default value for the new property is an empty string (no parent). This will allow existing configurations to run without modifications.

The structure of the configuration Pool is defined in the ***class moja::flint::configuration::Pool***. It will be extended to add a parent name (empty string indicates no parent).

```
class CONFIGURATION_API Pool {
  public:
   Pool(const std::string& name, double initValue = 0.0, const std::string& _parent_pool_name = "");
   Pool(const std::string& name, const std::string& description, const std::string& units,
        double scale, int order, double initValue = 0.0,
        const std::string& _parent_pool_name = "");
   virtual ~Pool() {}

   virtual const std::string& name() const { return _name; }
   virtual const std::string& description() const { return _description; }
   virtual const std::string& units() const { return _units; }
   virtual double scale() const { return _scale; }
   virtual int order() const { return _order; }
   virtual double initValue() const { return _initValue; }
   virtual const std::string& parent_pool_name() const { return _parent_pool_name; };  // New

  private:
   std::string _name;
   std::string _description;
   std::string _units;
   double _scale;
   int _order;
   double _initValue;
   std::string _parent_pool_name;    // New
};
```

Configuration Loading

The class used to read the config file is ***moja::flint::configuration::JSON2ConfigurationProvider***. This class will be extended to read the JSON defined above. The configuration Pools are converted into a Simulation Pool object. This class will also be extended to handle the "*Parent Pool Name*" data.

The C++ Interface for ***moja::flint::IPool*** will be modified:

```
class FLINT_API IPool {
  public:
   IPool() = default;
   virtual ~IPool() = default;

   virtual const std::string& name() const = 0;
   virtual const std::string& description() const = 0;
   virtual const std::string& units() const = 0;
   virtual double scale() const = 0;
   virtual int order() const = 0;
   virtual double initValue() const = 0;

   virtual int idx() const = 0;

   // Additions for parent pool                                  <<---- Additions here ---->>
   virtual const IPool* parent() const = 0;
   virtual const std::vector<const IPool*>& children() const = 0;

   virtual double value() const = 0;
   virtual void set_value(double value) = 0;
   virtual void init() = 0;

   virtual const PoolMetaData& metadata() { return _metadata; };
  protected:
   PoolMetaData _metadata;
};
```

The metadata as follows:

```
class FLINT_API PoolMetaData {
  public:
   PoolMetaData() : _name(""), _description(""), _units(""), _scale(1.0), _order(-1), _parent_name(“”) {}

   PoolMetaData(const std::string& name, const std::string& description, const std::string& units,
                double scale, int order, const std::string& parent_name)
       : _name(name), _description(description), _units(units), _scale(scale),
         _order(order), _parent_name(parent_name) {}

   virtual ~PoolMetaData() = default;
   virtual const std::string& name() const { return _name; }
   virtual const std::string& description() const { return _description; }
   virtual const std::string& units() const { return _units; }
   virtual double scale() const { return _scale; }
   virtual int order() const { return _order; }
   virtual const std::string& parent_name() const { return _parent_name; }

  private:
   std::string _name;
   std::string _description;
   std::string _units;
   double _scale;
   int _order;
   std::string _parent_name;
};
```

Framework handling of Nested Pools

During a simulation FLINT modules have access to the Land Unit Data object (LUD). The Interface is defined in ***moja::flint::ILandUnitDataWrapper*.**

This interface will need to be extended to allow the dynamic addition of Children Pools. The Parent Pool must exist. On error an exception will be thrown.

```
class FLINT_API ILandUnitDataWrapper {
  public:
   virtual ~ILandUnitDataWrapper() {}

   virtual std::shared_ptr<IOperation> createStockOperation() = 0;
   virtual std::shared_ptr<IOperation> createStockOperation(DynamicVar& dataPackage) = 0;
   virtual std::shared_ptr<IOperation> createProportionalOperation() = 0;
   virtual std::shared_ptr<IOperation> createProportionalOperation(DynamicVar& dataPackage) = 0;
   virtual void submitOperation(std::shared_ptr<IOperation> iOperation) = 0;
   virtual void applyOperations() = 0;

   virtual const OperationResultCollection& getOperationPendingIterator() = 0;
   virtual const OperationResultCollection& getOperationLastAppliedIterator() = 0;
   virtual const OperationResultCollection& getOperationCommittedIterator() = 0;

   virtual bool hasLastAppliedOperationResults() const = 0;

   virtual void clearLastAppliedOperationResults() = 0;
   virtual void clearAllOperationResults() = 0;

   virtual PoolCollection poolCollection() const = 0;

   virtual int getPoolCount() const = 0;
   virtual const IPool* getPool(const std::string& name) const = 0;
   virtual const IPool* getPool(int index) const = 0;

   virtual IVariable* getVariable(const std::string& name) = 0;
   virtual const IVariable* getVariable(const std::string& name) const = 0;
   virtual std::vector<std::shared_ptr<IVariable>> variables() const = 0;
   virtual bool hasVariable(const std::string& name) const = 0;

   virtual IOperationManager* operationManager() = 0;
   virtual const IOperationManager* operationManager() const = 0;

   virtual ITiming* timing() = 0;
   virtual const ITiming* timing() const = 0;

   virtual const configuration::Configuration* config() = 0;
   virtual void setLandUnitController(ILandUnitController* landUnitController) = 0;
};
```

This object allows modules to create and apply Pool transfers. The workflow for creating a transfer is:

- Using the LUD, create an operation (either a Stock or Proportional operation).
- This returns an ***IOperation*** shared_ptr
- Using the IOperation class, call the method ***addTransfer(src pool, sink pool)*** as required.
- Once the required set of Pool transfers have been defined, call the LUD->***submitOperation*** method

The ***ILandUnitDataWrapper*** will be extended with methods to support Children Pools and all existing methods using Pool names will use Dot notation to identify Children.

```
   virtual const IPool* getPool(const std::string& name, const std::string& parent_pool_name) const = 0;
   virtual const IPool* getPool(IPool* parent_pool, const std::string& child_name) const = 0;
```

```
   virtual const IPool* addChildPool(const std::string& parent_pool_name,
                       const std::string& name, double initValue = 0.0) override;
   virtual const IPool* addChildPool(const std::string& parent_pool_name,
                       const std::string& name, const std::string& description,
                       const std::string& units, double scale,
                       int order, double initValue = 0.0) override;
   virtual const IPool* addChildPool(const std::string& parent_pool_name,
                       PoolMetaData& metadata, double initValue) override;
```

The ***IOperation*** interface will be extended to allow both Stock and Proportional transfers to and from Child Pools. Existing methods will work, assuming the Module has used the methods provided to get the correct Pool handles. However, a utility method will be provided which allows the Child lookup at in the same request:

```
class IOperation {
  public:
   explicit IOperation(const ModuleMetaData* metaData) : _metaData(metaData), _hasDataPackage(false) {}
   IOperation(const ModuleMetaData* metaData, DynamicVar& dataPackage)
       : _metaData(metaData), _dataPackage(dataPackage), _hasDataPackage(true) {}
   virtual ~IOperation() = default;

   virtual IOperation* addTransfer(const IPool* source, const IPool* sink, double value) = 0;

   // ********************************************************************************************
   // New Parent/Child pool methods
   // ********************************************************************************************
   virtual IOperation* addTransfer(const IPool* source, const std::string& source_child_name,
                                   const IPool* sink  , const std::string& sink_child_name,
                                   double value) = 0;
   virtual IOperation* addTransfer(const IPool* source, const std::string& source_child_name,
                                   const IPool* sink,
                                   double value) = 0;
   virtual IOperation* addTransfer(const IPool* source,
                                   const IPool* sink  , const std::string& sink_child_name,
                                   double value) = 0;
   // ********************************************************************************************

   virtual std::shared_ptr<IOperationResult> computeOperation(ITiming& _timing) = 0;
   virtual void submitOperation() = 0;

   virtual OperationTransferType transferType() const = 0;
   virtual const ModuleMetaData* metaData() const { return _metaData; }

   virtual const DynamicVar& dataPackage() const { return _dataPackage; }
   virtual bool hasDataPackage() const { return _hasDataPackage; }

   virtual OperationTransferHandlingType transferHandlingType() const = 0;

  protected:
   const ModuleMetaData* _metaData;
   const DynamicVar _dataPackage;
   bool _hasDataPackage;
};
```

Dot Notation:

Nested Pool names will use dot notation to identify levels of Children. This takes “.” in Pool name strings to identify levels. Some examples:

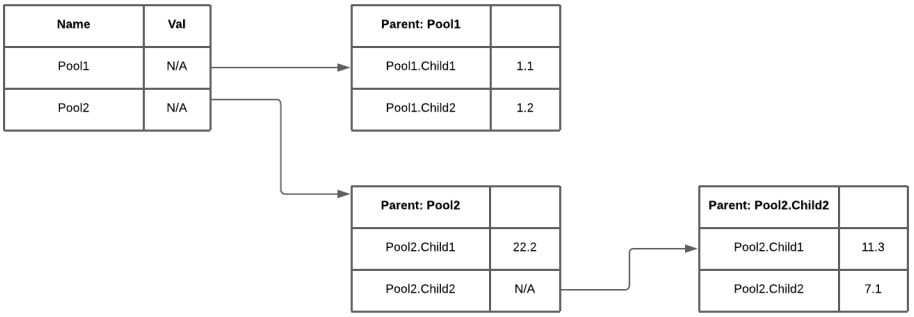
```
debrisCM                        # single Pool
debrisCM.eucalyptus             # referencing Child Pool
debrisCM.eucalyptus.BlueGum     # referencing Child of Child Pool
debrisCM.eucalyptus.SpottedGum  # referencing Child of Child Pool
```

```
aboveGroundCM
aboveGroundCM.Stems
aboveGroundCM.Branches
```

Examples of Nested Pool Value calculations

Below in the figures are some examples of how Pool values would be calculated in various configuration scenarios.

Fig 4: nested calculations



Pool name	Result	
Pool1	2.3	Sum of (1.1 + 1.2)
Pool2	40.6	Sum of (22.2 + (11.3 + 7.1))
Pool2.Child1	22.2	Sum of (22.2)
Pool2.Child2	18.4	Sum of (11.3 + 7.1)

Matrix of valid Pool transfers

Warning: Source and Sink Pools cannot be the same

	Sink Pool type			
Source Pool Type	**Pool**	**Parent Pool**	**Child Pool (Parent A)**	**Child Pool (Parent B)**
Pool	Yes	No	Yes	Yes
Parent Pool	No	No	No	No
Child Pool (Parent A)	Yes	No	Yes	Yes

NOTE: The system will throw an Exception when an invalid transfer is made.

Future Extensions

Add the concept of a Default cohort - so that Pools can have cohorts added post value change, and so that the range of valid moves to and from pools is complete (even if a little obscure to a user). This would essentially mean any pool that has no children, and has one or added, will have any value it currently has assumed to be in a child called “default”.

While this addition is not required to handle cohorts, it may add some flexibility to the system with little effort to implement.

System level settings

A “***NestedPools***” settings section will be added to the “***LocalDomain***” section, allowing system level options to be defined.

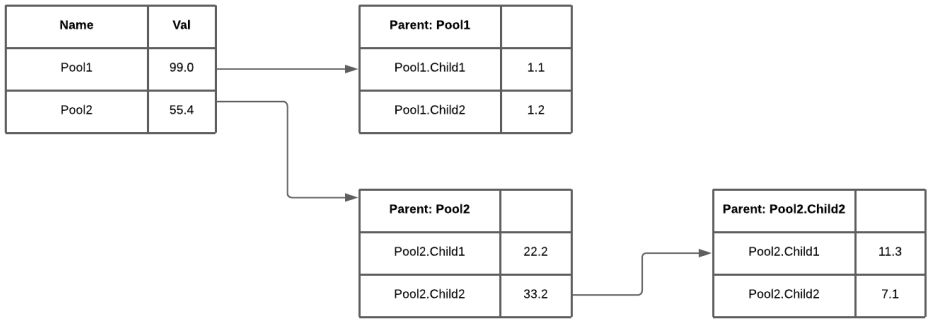
These options will be:

Setting	Description
default_child_enabled	Any Pool that has a Child added will get an Automatic “Default” Child Pool. This allows handling for: 1. Pools that already have values when a child is added. The existing value is treated as the “Default” Child Pool. 2. Transfers to/from Parent Pools (without a Child specified). The transfer will be assumed to be from the “Default” Child In the case this flag is not enabled, invalid actions will cause the system to throw an Exception. Some of these include: 1. Addition of a Child Pool to a Pool that already has a value 2. Incompatible Source/Sink Pool combinations in transfers (see transfer matrix)

Examples of Nested Pool Value calculations

Below in the figures are some examples of how Pool values would be calculated in various configuration scenarios.

Fig 3: With “Default Child” enabled



Pool name	Result	
Pool1	101.3	Sum of (99.0 + 1.1 + 1.2)
Pool2	129.2	Sum of (55.4 + (22.2 + 33.2) + (11.3 + 7.1))
Pool2.Child1	22.2	Sum of (22.2)
Pool2.Child2	51.6	Sum of (33.2 + (11.3 + 7.1))

Matrix of valid Pool transfers

Warning: Source and Sink Pools cannot be the same

If Nested Pools “*Default Child*” is enabled

	Sink Pool type			
Source Pool Type	**Pool**	**Parent Pool**	**Child Pool (Parent A)**	**Child Pool (Parent B)**
Pool	Yes	Yes	Yes	Yes
Parent Pool	Yes	Yes	Yes	Yes
Child Pool (Parent A)	Yes	Yes	Yes	Yes