Module Design

Luke Hodkinson
Center for Astrophysics and Supercomputing
Swinburne University of Technology
Melbourne, Hawthorn 32000, Australia

June 9, 2013

1 Introduction

The science modules form a core part of the TAO system, not only from an operational sense but also as a part of continued development. It is our responsibility to ensure future developers are able to modify existing modules and create new ones with a minimum of difficulty.

This document describes the design of the science modules' framework to allow for intuitive development. There are two major sections, the first describes the interface for modifying and developing new modules, and the second section describes how new modules may be added into the primary TAO binary.

2 Science Module API

The science module API can be broken down into N subsections:

- module processing,
- module parameters,
- data pipeline,
- execution ordering, and
- XML description.

Each of these subsections will be discussed, in order, in what follows. Prior to that is provided a UML diagram of the overall architecture of the API as a reference in **??**.

2.1 Module Processing

The question to be answered here is, "where do I add my code to perform the processing my custom module foo needs to perform?" The base level module provides a virtual method named !execute(tao::galaxy galaxy)!. This is the primary interface for performing the execution of each module. !execute! will be called to perform the module's processing on the current chunk of galaxy data at the appropriate times by the TAO system.

2.2 Module Parameters

Talk about the dictionary.

2.3 Data Pipeline

2.3.1 Passing Data

Passing data from one module to the next is made a little tricky as a result of needed to pass data from the database alongside data generated from other modules. Once data has been acquired from the database we cannot then insert new columns into the returned rows. To handle this we provide a !galaxy! data structure designed to provide a consistent interface for setting and retrieving named data to be made available to other modules. The !galaxy! object is the accepted, and only provided, mechanism for passing data between modules.

2.3.2 Data Chunking

To minimise inefficiency data may be set on the !galaxy! object as vector data. It is expected that all data will be set as vector data, even if it is of length one, to generalise the interface for science modules. This way, a science module should be designed to work with arbitrarily sized vectors of input data.

To facilitate setting and retrieving vectored data, several methods are provided on the galaxy object.

3 Execution Ordering

The science module execution is heavily dependent on ordering. There is no sense in executing the SED module before the lightcone module has executed. To facilitate ordering a directed acyclic graph approach is used. Each module tracks a set of its parents (**??**) and ensures that each parent has been executed before it allows itself to execute. See figure **??** for a code snippet on how this operates.

In the case of data chunking where the chunk size is less than the total number of galaxies generated for the local process, the overall execution loop will need to be executed multiple times, once for each chunk. This raises the question, how does the execution system allow each module to know it needs

```
// Process all parents.
bool all_complete = !_parents.empty();
for( auto& parent : _parents )
{
  parent->process( iteration );
  if( !parent->complete() )
    all_complete = false;
}

// Call the user-defined execute routine.
if( !all_complete )
  execute();
else
  _complete = true;
```

to run again, even if it has already run? To provide this functionality we use a simple counter to track which chunk iteration we are currently working on. A module stores the index of the counter for the most recently processed chunk. It will not execute again until the counter it sees is greater than its stored value.

Terminating the overall execution is handled by each module keeping track of an internal !complete! boolean flag. Any module can set this flag itself as a part of its operation, or it can wait for this flag to be set by the TAO system automatically when all of the module's parents report that they are complete.

```
// Keep looping over modules until all report being complete.
bool complete;
unsigned long long it = 1;
do
{
  // Reset the complete flag.
  complete = true;

  // Loop over the modules.
  for( auto module : tao::factory )
  {
    module->process( it );
    if( !module->complete() )
      complete = false;
    }

    // Advance the counter.
    ++it;
  }
while( !complete );
```