Apollo

Project specification

Confidential

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete. What were you thinking, better learn to live with this fact.

# Introduction

The goal of the project system is to provide the user with access to the different data sets that exist in a project. One project may consist of multiple data sets. The project system will provide ways to logically organise and relate the different data sets in a project. The project system holds zero or one project at any one time.

A single project consists of several different parts.

* One or more datasets. Each dataset contains data that describes physical behaviour of some aspect of the project.
* The relations between the different datasets. Datasets are hierarchical and form a directed acyclic graph (i.e. it's not possible to create cyclic relations etc.). One dataset can 'spawn' multiple sub-datasets. Each sub-dataset can describe a specific set of physics, a sub-area, a specific set of variable values etc. The parent dataset may store its own data or it may simply link to the sub-data sets. Most likely the parent will only contain aggregated data (data calculated out of a set of sub-datasets). Child datasets always only have exactly one parent.
* Meta data describing the complete project. Note that this metadata does not describe the individual datasets. Examples of project wide meta-data are:
  + Any notes made by the user
  + The original goal of the project. This goal can be one of: robust design, what-if research, optimisation, research.
* The links to the history of the project and the individual datasets

The project system controls the creation, editing and deletion of the datasets owned by a project. A dataset can be created based on a request from the user or a request from a (parent) dataset. Users are only able to create child datasets for datasets that have been created by the user. It is not possible for users to create child datasets to a system created dataset. Furthermore a user is not able to edit pr delete a system created dataset although they are able to copy the dataset. A copied dataset can only be created as a sibling of the original dataset or as a root dataset.

A dataset will request the creation of a child dataset when:

* The parent dataset requires the processing of a large amount of data in order to obtain a single set of values. By delegating this data processing to a child dataset it is possible to retain all the intermediate data, thereby making subsequent data processing faster. Furthermore using a child dataset provides the standard benefits of using a dataset (parallelisation, distributed computing, easy to terminate failing processes etc.)
* The processed child data needs to remain available for later processing, but this data isn't directly necessary in the near future. By loading this data into a child dataset it maybe persisted and unloaded from memory without losing the information.
* The parent dataset would need to make changes to unique (singleton) data. An example of this is geometry (which is controlled by the CAD system). By making a copy of the data in a separate dataset it is possible to manipulate the data without influencing other processes.
* The parent dataset needs to determine the physical state of a model for many different parameter values. By creating a child dataset for each parameter variation it will be possible to process the parameter variations separately without influencing the original dataset.

Note that a datasets can only request the creation of direct child datasets.

A single dataset consists of several layers.

* A collection of data. This data may be obtained from simulations, experiments etc. Data can only be placed into a dataset through a generator. The data also contains instructions for finding values for each of the variables which are defined in the dataset. Finding the value for a variable may be done through:
  + Grabbing the value from a field, i.e. the variable describes a constant.
  + Performing a (relatively) simple calculation, e.g. calculate the pressure for a given velocity in an incompressible flow field.
  + Performing a sequence of actions through a schedule created by a generator. In this case a schedule is a directed graph that describes the relationships between the different steps necessary to obtain the value for one or more variables.
  + Performing a sequence of actions through a sub-dataset.
* One or more generators. A generator creates data, either from other data or from user input. Examples of generators are simulation tool (e.g. CFD simulation components) or a generator that imports experimental data from an Excel spreadsheet). Generators may cooperate with other generators to generate derived data.
* Zero or more visualisers. A visualiser prepares data for visualisation. Visualisers do not directly write to the screen or to the UI. They just transform the data so that it can be processed by the visualisation system in the UI of Apollo.
* A collection of metadata.

# Architecture

The project system will consist of two major parts. The first is the project service that is included in the core assembly. This will contain all the necessary components to communicate with the other parts of the core and to store the project data. Note that the data only describes the datasets and their relations but does not actually contain the actual data from the datasets. The core assembly will also contain all the proxies & communication channels necessary to handle communication with the actual datasets.

The second part is a separate (console) application that will load and hold the individual datasets. Each dataset that is loaded will be loaded into its own instance of the application. The use of a separate application to hold datasets provides the following benefits:

* It is possible to load the dataset into a 32- or 64-bit application independent from the core application. This allows loading datasets into a 64-bit application even if the core is loaded as a plug-in to a 32-bit application.
* The core should not have to load any of the plug-ins which should reduce the attack surface of the application. The dataset application can be a sandboxed application because the core controls when and how it will be started.
* Datasets will be easy to unload in case this is desired, e.g. unloading an un-used dataset or a crash.
* The datasets will reside in 'physically' separate applications which makes it less likely that problems in one dataset influence the functioning of other datasets.
* By placing datasets in separate application distributed computing becomes simpler running datasets in a different application is the standard and not a speciality.

Obviously the use of separate applications to hold datasets does have drawbacks:

* There are two or more applications (the core application and the dataset application) which need to communicate which will increase the complexity of the solution.
* The licensing situation increases in complexity because now there are multiple applications that need to check the license.
* The communication between the main application (which has the UI) and the dataset application(s) may need to handle large amounts of data (for display) which is slower than in-process transfers of data.

## Project system

The project system forms the central point for all project related activities. The main components in the project system are:

* Project service. Used to communicate with the kernel and other services.
* Project. Controls the relations between the datasets and handles communication with and between datasets.
* Computation node allocator. Allocates and controls 'nodes' for computation. In order for a schedule to be execute a 'computation node' is required. The usage of nodes ensures that it is not possible to execute too many schedules at the same time.

- Load projects

- Create new projects

- project service is a facade for the rest of the system. We don't really expose anything.

- Which API's do we expose?

-

- Note that a dataset may need to create/load sub-datasets, however we want to control what gets loaded because we don't want to overload the machine(s). So we need a global loader. There should be a default one in the project system, however we want to be able to use that as a proxy for another one in case we're running distributed calculations etc.

## Dataset application

- Holds a single data set. Can load a single dataset only, and only on startup

- Can create an empty dataset and fill it with data from other datasets (allows ad hoc creation of datasets which is needed for distributed computing)

- Can communicate with the project, parent dataset and child datasets. this allows a dataset to rely on child datasets for data / processing etc.

- Also we probably want to throw a 'script' at the app, i.e. a list of things to do and then get the full result set back. That way we don't have to tell the app every little step one at the time.

- How do we communicate? TCP? WCF? Named pipes? Probably depends on the locality of the core vs the dataset app. We can use named pipes for our own datasets on a single machine. But we'll probably need TCP for distributed stuff.

## Computation node service

- Provides distribution control for 'computation nodes'. In order to run schedules on a dataset a 'computation node' is required (sort of like a thread pool, but this allows distribution too)

- Allow installing the service (so that we can handle multiple users having multiple Apollo instances), not installing the service (allows a single user to have multiple Apollo instances, but multiple users will have multiple colliding Apollo instances).

- Might also need to tie this into the distribution / p2p system. Then we can distribute over larger number of machines etc.. This would of course require a dataset to be splittable (i.e. domain decomposition needs to work ....).

- Note that the allocator can connect to an external source for computation nodes (e.g. a service (for computation on a single machine) or a cluster master (for distributed computations)).

## Project

- Describe where the project tree is stored

- Undo / redo?

- Do we hide system generated datasets from the user. No we just mark them specially, the user can't edit them so it may work out automatically ...

- separate simulation components and data? So that we don't get the same as with MCS where we mixed it. This could be problematic with persistence etc..

- Which connections do we allow in the unit graph? Do we allow merging paths?

## Datasets

- Locking of schedule / components / specific?

- What data does a dataset store? dimension, goal, physics, materials, symbols .....

- dataset status? running, stopped, failed ...

- Tracking performance?

- What data goes where?

- The root has: domain definitions (geometry), physics, materials, etc.

- child datasets: only the data that has changed.

- Describe running a dataset. We allow running on the same machine, running on a remote machine or running distributed (controlled by a master host?)

- running datasets directly, per batch or postponed. Note that the controller is responsible for the running order etc. Main thing is when we try to run too many datasets on a single machine. We should normally leave it up to the app, but there might be a service that can handle the processing.

- Data set can define one schedule as the default one? But we don't need to have one? How will we handle simulations. Normally the user should set them off.

- We could form the default schedule ourselves. It just depends on which variables we want to know stuff about. We could for instance let the user indicate which variables they would like to see (for the minimum). Then we can create a schedule for that.

- It should be possible to unload a dataset until it is required again. This allows putting datasets to 'sleep' while they are not required. There should be a way parent datasets can prevent this from happening if they need to get data from a child on a regular basis. Also there should be some rules for when a dataset can be unloaded (or loaded in the first place for that matter).

- Can we load datasets in a lazy fashion. e.g. only load the meta-data first. Then load the individual data elements when required.

- How to handle save-points / resetting of a dataset. Note that resets are back to the original state. It only resets the calculated values. And how do we handle resetting a dataset that has child datasets for the generation of values

- Do copy on write for child datasets

- Domain decomposition can be handled by splitting the dataset into a set of children. The real (parent) dataset then only needs to control the children. This does require the creation of ad hoc children though. And the adsorption/splitting of children if we need to handle re-distribution. That means that it is possible for a dataset to be temporary and possibly not part of the project graph?

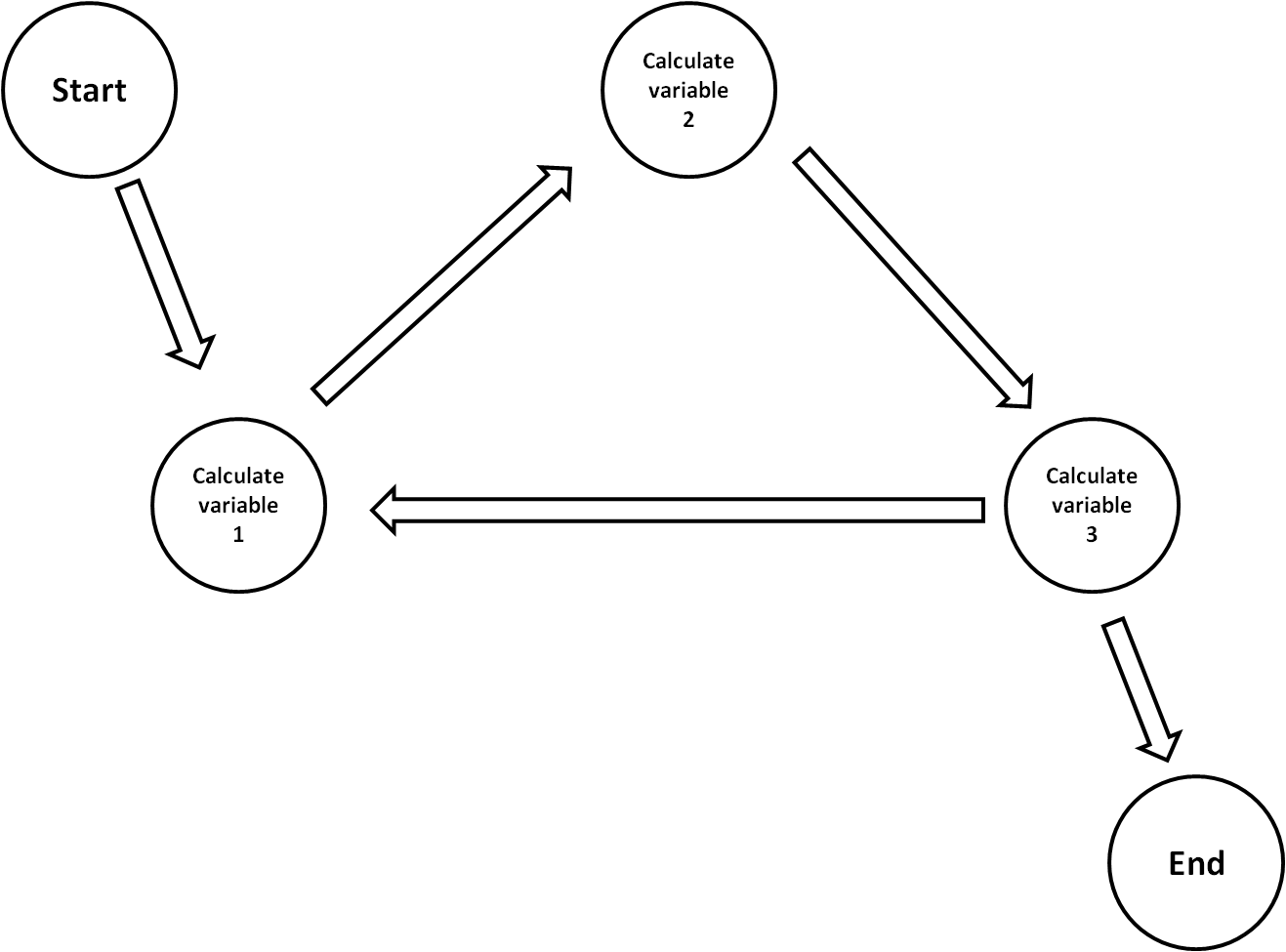
- Allowing domain decomposition requires the ability to pump across (relatively large amounts of) data. For domain decomposition we can send across the computational data only. If a domain decomposition dataset dies then you simply lose the data and have to recalculate.

## Generators

## Visualisers

## Schedule

In a schedule the graph nodes describe the action(s) that should be taken when the schedule processor arrives at the given node. The graph edges provide the connection between the different graph nodes. Graph edges may provide the schedule processor with directives indicating if a certain edge can be followed or not.



A schedule

When the value for one or more variables are desired the project creates the directed graph that will generate the variable values with the least amount of effort.

- A schedule is a directed graph that describes the relationships between different steps that need to be taken to obtain the desired results (see example?)

- Using variables can lead to a schedule (but the system will assemble it).

- Can use both in combination

- Schedules can be defined for small and big things

* Proxies (messages, timeline)
* Commands
* History
* Project
* Need to specify how the data is stored and retrieved. We should be able to define a standard data interface at some high level so that we can compare data between different generators (and between identical generators). Also visualizers will need to use the data interface somehow
* Generators: we need to define a default scheduling system. Experiment generators will need them also
* What kind of technical difficulties are we expecting