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.

# 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 or 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. Note that the root dataset cannot be copied.

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 or a stored value.
  + 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.
* 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 because 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(s)) 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.
* **Dataset allocator:** Creates and controls the dataset application. Both the project and existing datasets can request the creation/loading of one or more datasets. The allocator ensures that datasets are loaded on the correct machine (for distributed computing etc.) without overloading the machine.

The project system provides the other services with access to the currently loaded project. Furthermore it provides access to the configurations of the computation node allocator and the dataset allocator. Note however that external components do not have any influence on the actual execution of either allocator. Furthermore only the internal components can request the allocation of nodes or datasets.

## Dataset application

The dataset application is an application without an user interface. This application is used by the Apollo project to load datasets. Loading datasets into a separate application provides benefits of distribution and protection of other parts of Apollo. The application will only be able to contain a single dataset, however it will be possible to start as many instances of the application as desired, limited only by the hardware specifications of the machine on which the application is started.

The dataset, which can either be an existing one or a new one, can only be loaded on start up from a stream provided by the main application or the parent dataset. If multiple datasets need to be loaded in sequence then different instances of the application should be used. This ensures complete resource cleanup when a dataset is unloaded.

The dataset application will be able to communicate with other dataset applications and with the core application. Communication between datasets and the core application is used to:

* Send commands from the core application to the dataset.
* Send data requests from the core application to the dataset.
* Send data from the dataset back to the core application.

Communication between different datasets is used to:

* Send commands from the parent dataset to the child dataset.
* Send data requests from the parent dataset to the child dataset.
* Push data from the parent dataset to the child dataset.
* Send data from the child dataset to the parent dataset.

The communication is handled through three different channels (see figure below):

* Communication between datasets and the core application on the same machine is handled by the WCF communication layer through the use of named pipes. This layer provides abstractions for the sending of commands and requests across to another dataset or the core application on the same machine. These connections are only used for data that is related to the project system (e.g. data in a dataset, status of a schedule etc.). Communication regarding distributed computing is not done through this communication layer.
* Communication between datasets and the core application on different machines is handled by the WCF communication layer through the use of Peer-to-Peer (P2P) or TCP connections. These connections are only used for data that is related to the project system, not for data regarding distributed computing.
* Communication between datasets regarding distributed computing is handled via a MPI (Message Passing Interface) layer. Individual datasets will be able to communicate with any other dataset through this layer, even if that other dataset physically resides on a different machine.

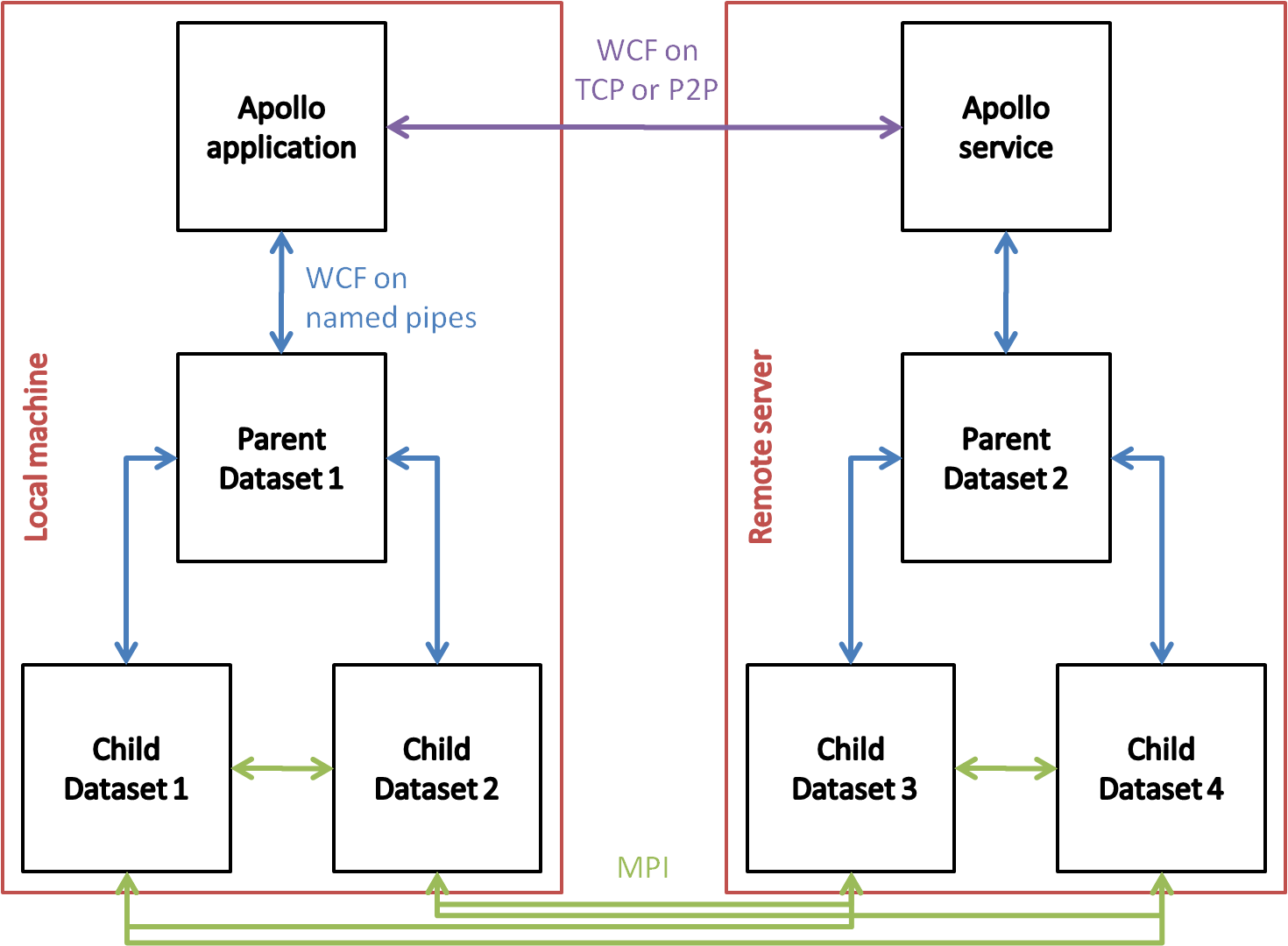


Figure : Proposed data communication channels for the dataset application

Note: Besides WCF there are several other possibilities for communication methods. The most suitable ones are:

* The MassTransit library. Used for distributed communications in business applications. The standard protocol is a XML SOAP. Different methods of serialization are available.
* The nServiceBus library. Used for distributed communications in business applications. The main method of communication is through the MSMQ (Microsoft Message Queueing).
* The .NET framework remoting capabilities.

Each of these options were reviewed and WCF was selected because:

* WCF is meant as the new communication layer for .NET. It provides several different transport modes like TCP, HTTP and named pipes.
* .NET remoting has been deprecated in favour of WCF.

In a dataset application all communication is handled by the communication layer. This layer is responsible for translating the incoming messages and passing them on to the correct parts of the application and the dataset. By letting the application handle the incoming messages it is possible to provide control over both the application and the dataset.

The communication layer handles all the communication between the application and the data sets. For communication with datasets on the same machine named pipes will be used. The communication layer is responsible for the creation of a unique named pipe for each communication pair. Note that one data set application may need to communicate with multiple partners (e.g. the Apollo application and one or more other dataset applications).

For communication with datasets on remote machines a TCP or P2P channel will be used. In this case the communication layer is responsible for the creation of a channel. In general only one channel should be created even if the application needs to communicate with multiple remote machines. The use of message identifiers will be used to route the messages to the correct recipient.

When a TCP or P2P channel is used the data stream will be compressed to improve data transfer rates. The communication layer will not secure the actual messages. That will left to the security that comes with the actual channels.

## Dataset allocation service

The dataset allocation service handles the allocation of new dataset application instances across the available machines. Clients, being the project or another dataset, can request one or more new or existing datasets to loaded. In a request the client will specify:

* The maximum number of datasets that need to be loaded.
* If the datasets need to be loaded at the same time, for instance for domain decomposition purposes, or if the datasets can be loaded sequentially
* If multiple datasets can be loaded onto the same machine or if each dataset will require its own machine.
* The machine property demands, like available memory, for the datasets. Note that it is assumed that all the datasets, for which a request is send out, have similar hardware demands.
* Machine loading preferences, like local machine or distributed. The local machine can be preferred if the child dataset is small or is loaded for visualisation purposes.

In response to a loading request the service will determine how many machines are available for use at the current time, what the specifications are for each of the available machines . This information, together with the information in the allocation configuration, is used to determine how many dataset applications can be started. Once this is done the service will instantiate the dataset applications on the most suitable machines. The following listing provides the exact process used for instantiation of new dataset applications.

**Dataset loading protocol:**

1. Client sends request for one or more datasets to be loaded. Note that these datasets may be existing datasets (i.e. persisted to a disk somewhere) or new datasets.
2. The dataset allocation service determines the minimum hardware specifications for the required machines.
3. The service determines how many machines are available that at least match the minimum hardware specification. Any machines that are available but do not meet the specification are stored in case they are required, e.g. if there are too few machines or even no machines that meet the spec.
4. For each available machine the service determines the busy-ness based on the CPU and memory loads. The collection of machines is sorted by hardware specification and machine business. This ensures that the first machines to be used will be the ones who are least busy and have the lowest hardware specifications (thus leaving the more powerful machines available in case they are required).
5. The service builds the distribution plan which describes which machines will be used and how many datasets will be loaded onto each machine. While building the distribution plan the service always favours parallel execution of datasets (i.e. distributing the different datasets as much as possible).
   1. Getting the number of datasets that are required.
   2. Getting the collection of available machines.
   3. Getting the distribution preferences.
   4. Distribute the datasets over the available machines. If multiple datasets can be placed on a single machine then we assume that each dataset will only use a single, physical, core, otherwise each dataset will be placed on a different machine. Note: The service does not make a special case for the 'local' machine of the client, however we may decide to allocate or specifically not allocate on the 'local' machine based on the loading preferences.
6. Once the distribution is known the service notifies the client and provides the suggested distribution plan.
7. In response to the suggested distribution plan the client either:
   1. Accepts the plan and request that the service executes the plan
   2. Does not accept the plan and requests a new plan from the service

In order to find available remote machines and distribute datasets to these remote machines the allocation service requires that there is a receiving application available on the remote machine. This application provides the allocation service with information about the remote machine and it instantiates the dataset applications on the remote machine. If the remote machine is the head node of a cluster of machines then the application forms the access point to the cluster. It will provide the dataset allocation service with all the information about the cluster. In order to distribute datasets across the cluster the service will request a distribution plan from the application running on the cluster head node. This ensures that the cluster application is always in charge of dataset distribution.

The user can provide a distribution configuration determines which machines the datasets can and cannot be distributed to. The configuration can also indicate a preference for a certain machine (or group of machines) for either the general case or for a specific project / dataset.

## Project

The project is the facade for the collection of related data which is stored in one or more datasets. All requests for data or data manipulations from the user interface are handled by the project and passed on to the respective dataset(s). The project stores the following information:

* Information about the datasets that belong to the project. This information includes amongst others the directed graph that describes the relations between the different datasets.
* The project metadata which contains information as:
  + ??
* The additional data about the project given by the user. Examples of this data are:
  + Notes the user made about the project.

Information describing the datasets can be divided into three different parts. The first part is the relations between the different datasets. This information is stored in a Directed Acyclic Graph (DAG). The DAG will not store all the information about the dataset, instead it will only relate the ID numbers of the different datasets. The project always has one dataset which forms the root of the dataset graph. There can only be one root dataset and it contains the general project information such as:

* Geometry
* Materials for every section of the geometry
* Physical models for every section of the geometry

The second part is the permanent dataset information. This is information describing a dataset that is always needed, even if the dataset itself is not actively loaded. This permanent information consists of:

* Who created the dataset, either the user or the system.
* The reason for the creation of the dataset. This information is only required when the system created the dataset in the first place (although it may be set when the user creates the dataset). Possible values are:
  + Domain decomposition
  + Parameter study
  + Sub-experiment
* Persistence location / ID. This describes the location of the dataset information and how to load this data into a dataset application.

The third part is the non-permanent dataset information. This information is only stored for the datasets that are currently loaded into a dataset application. This non-permanent information consists of:

* The proxy that is used to communicate with the dataset.
* The collection of additional commands that the dataset defines on top of the default commands.

The project handles all the requests for dataset creation, loading and unloading. The disconnected nature of the dataset application requires that the project is flexible in handling the different datasets. For instance it is possible that a parent dataset is waiting for one or more child datasets to perform an expensive calculation. During this time the project could decide to unload (or hibernate) the parent dataset in order to free up hardware resources. Obviously datasets can only be hibernated if the user isn't interacting with them and no calculations or visualisations are running in the dataset.

Besides the information describing the datasets the project also provides a set of commands that can be invoked by the user interface or any other part of the Apollo application in order to interact with the project or one of the datasets.

* Commands that work directly on the project are:
  + Create a new dataset or copy an existing one.
  + Activate / load an existing dataset (if it is not currently active).
  + Unload an active dataset (and possibly store the information in the dataset on disk)
  + Undo / redo last action
* Commands that work on the selected dataset are:
  + Undo / redo last action

## Datasets

A dataset encapsulates all the elements needed to store, process and visualise a collection of data. In general a dataset will consist of:

* Data storage
* Generator
* Visualizers

A dataset can store any kind of information necessary for a complete physical model and its calculation results. Normally this will mean that a dataset stores:

* Geometry information specific to the dataset
* Physical model information, including boundary information, relevant equations etc.
* Material information
* Any necessary (temporary) numerical information
* Any data obtained from one or more calculations

The dataset will only store as much information as it needs. Any information that is identical to that of its parent dataset will normally not be stored. The only time this information is stored is:

* when the dataset requires the information for calculations
* when the dataset is on a physically different machine than its parent dataset

The dataset also contains the components necessary for the data processing, either for data generation purposes (generators) or for visualization purposes (visualizers). To enable easy persistence and exchange of generators and visualizers neither component is allowed to directly store any kind of persistent state. The lack of state allows the dataset to only store component identification data for the generators and visualizers. This is useful because the Apollo system is based on a plug-in system which makes it necessary for the dataset to persist information describing the different components that handle data storage, data manipulation and data visualization. The information about the components is the bare minimum information required to load the components, the dataset does not in fact contain the actual binary code for the components.

As mentioned previously a dataset can request the creation of one or more child datasets. These datasets initially inherit all their data from the dataset that requested their creation. Once created changes can be made to the child datasets to make them unique.

The reasons for creating child datasets can be summed up as:

* Domain decomposition. This allows splitting a problem into sub-sections by giving each child one section of the original problem. Each child then gets to compute the results for that sub-section. This does require parallel calculation of the sub-problems because often the different sub-sections are related by their boundaries.
* Parameter study. This allows splitting a problem into many sub-problems where each sub-problem has one fixed set of parameter values. This allows serial calculation of the sub-problems because each problem is in fact independent of the other problems.
* Calculate one or more integral values over a domain. In this method a child calculates all the domain (or field) values and only returns the integral values to the parent.

## Generators

A generator consists of a group of components which cooperate to provide one or more ways of importing or manipulating data. At any point in time a dataset can have exactly 1 generator. The Apollo application defines two different types of generators. The first generator type handles importing and manipulating experimental data. The second generator type handles importing, generating and manipulating simulation data.

The generator only provides the facade for the data manipulation capabilities. Components from the plug-ins will do the actual data manipulation work. The generators provide the required hooks and commands that can be used by the external users and the components. The available commands, which are used by the external users, are:

* General
  + Import data from stream
  + Export data to stream
  + Add component
  + Remove component
  + Reset to last restore point
* For the experimental generator
  + ??
* For the simulation generator
  + Run simulation
  + Pause simulation
  + Stop simulation

And the suggested generator hooks, which are used by the plug-in components, are:

* General
  + Add schedule
  + Remove schedule
  + Execute schedule
* For the experimental generator
  + ??
* For the simulation generator
  + Result verification

## Visualizers

A visualizer consists of a group of components which cooperate to transform data into a known data format for visualization purposes. This is done by transforming the data into one of the known visualization data types which can describe one of:

* 2 or 3 dimensional graphics
  + Contour path or surface information. Which can either be defined on a plane, on a surface or through an iso-surface.
  + Path line information, including information describing the twist of the line.
  + Vector field information, including possible error boundaries.
* Text, either as single values or as a table
* 2 or 3 dimensional graphs, including error information

## Schedule

A dataset contains one or more schedules. A schedule is a directed graph that describes the order in which certain data transformations need to be taken in order to obtain the desired result. In a schedule each node contains a set of data transformations which need to be invoked in sequence. This sequence can be very simple, i.e. a single transformation, or very complicated, e.g. a fluid flow solver. In order to allow computations that involve multiple physical models interacting it should be possible to define sub-schedules where a schedule node links to another schedule. This sub-schedule can then be executed when the execution of the parent schedule arrives at the linking node.

The edges of the graph indicate both the order in which the nodes must be visited and what the pre- and post-conditions are for entering and leaving a node. The figure below provides an indication of what a (rather simple) schedule can look like.

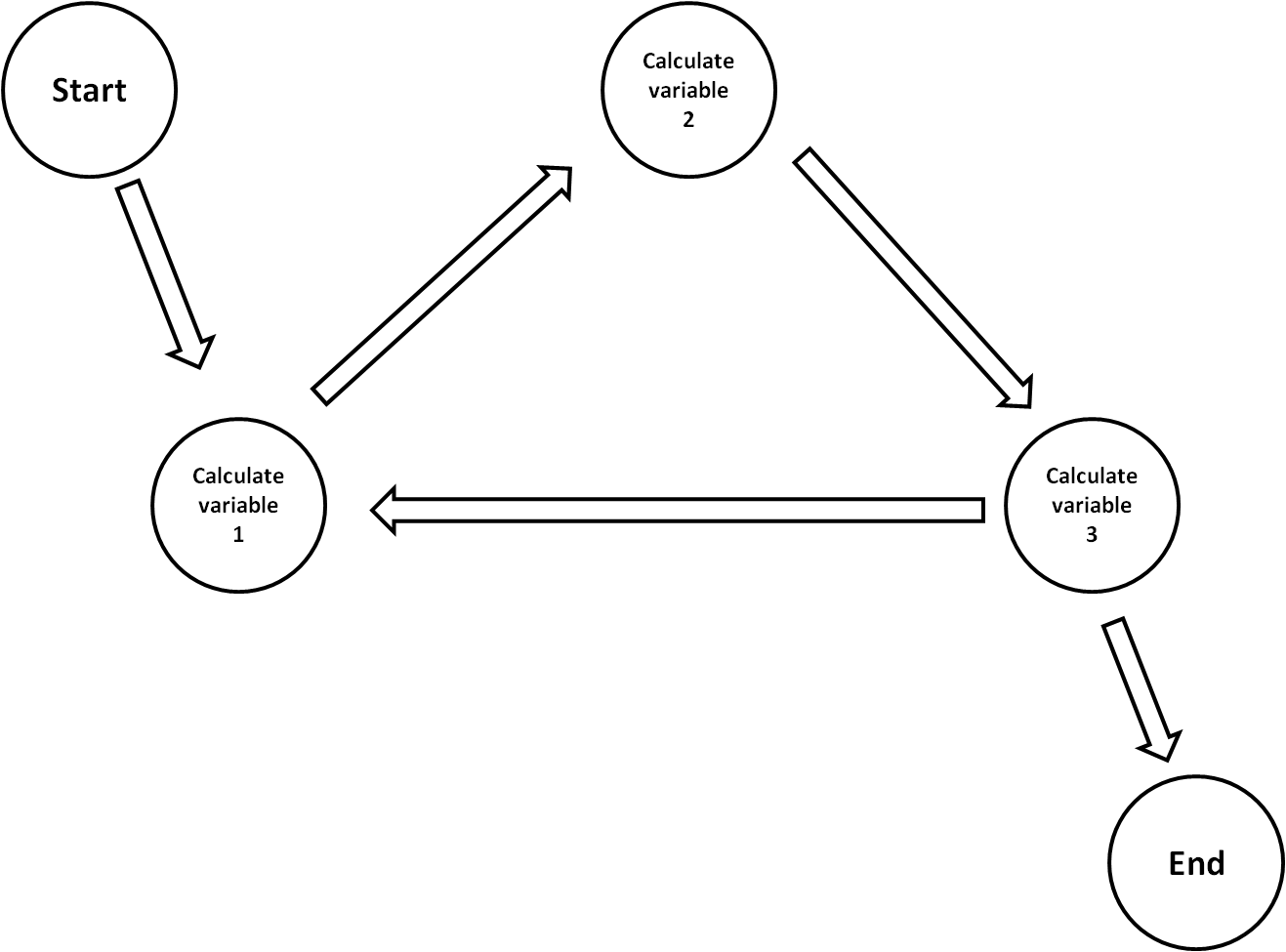


Figure : An example of a simple schedule

Schedules can be defined by a generator component or by the dataset application. When a component needs to perform calculations to obtain a value for a variable it can define the schedule to achieve this. In order to do this the component can use one of two ways. The first method is by indicating which data transformations must be grouped and how they are ordered. From this information a schedule can be created. Schedules which are created by this 'direct' method cannot be changed because it should be assumed that the order of the different transformations is critical in order to obtain a correct result.

The second way of creating a schedule is by defining the different actions that have to be taken to obtain a value for a variable. Once that is done the component can define which variables depend on each other. The dataset application can then create a schedule based on the dependencies between the different variables. Schedules created by this 'indirect' method can be reordered by the system in order to create a more efficient way of obtaining the variable values.

A single dataset may have multiple schedules defined at any given time. These schedules can only directly be executed based on the request of an internal component. The user only has indirect control. In order to ensure that the user can start the simulation the simulation generator marks one of the schedules as the main simulation schedule. The 'run simulation' command will start the execution of this schedule.

In order to prevent users and components from modifying a schedule while it is being executed it is possible to lock the schedule and the connected components. This lock is a global lock, i.e. when one schedule is executing then all components are locked for changes.

## Schedule Executor

A schedule just stores the information necessary to describe the action order and dependencies, it is not capable of actually executing the different actions. The actual execution is done by a schedule executor.

When executing a schedule the schedule executor first locks the schedule and all the components in the dataset. This prevents any component from making breaking changes to the schedule. Once the lock has been taken out the executor starts executing the schedule at the starting point. During a schedule run the execution can be stopped in several different modes:

* Immediate stop. This kills the current actions and immediately stops the schedule execution. The chances are that the data in the dataset is corrupted if this method is used.
* Finish current node. This allows the current action to finish. Depending on where the schedule execution was the data may be corrupt (e.g. if the execution stopped halfway into a loop)
* Finish current group. This allows the current set of actions to finish, e.g. loops will always run to the point where the loop conditions are verified etc.

Note that stopping and pausing the schedule execution is in fact the same thing. The only difference is whether the schedule execution is started from the point where execution was stopped or from the beginning.

Once the execution of a schedule is finished the executor marks the schedule as 'finished'. Any changes to the schedule will invalidate this status and will require it to be executed from the beginning.

The schedule executor will try to maximize the usage of the current machine by running schedule elements in parallel if allowed. While the schedule executor allows some parallelization of the schedule execution it will not allow distribution of schedule actions or components. The executor assumes that all actions will be executed in the current context (i.e. on the current machine, in the current application, in the current AppDomain).

Prior to executing a schedule the executor will perform a final validation of the schedule in order to make sure that the schedule is reasonable. This means that the executor looks for:

* Infinite loops created by two or more schedule nodes.
  + All schedules must have a designated start and end
  + It must be possible to move from the start to the end
  + It must be possible to reach the end from all nodes
* Sub-schedules cannot link back to the original schedule
  + All schedules must be completely defined before the start of the schedule execution
  + A schedule may not link back to a parent schedule

Even with all these checks it is still possible that the schedule ends up in an infinite loop or on a node that cannot be exited.

## Components

All modelling functionality in the Apollo application is provided by plug-ins. In order to use a plug-in the user can deploy it in a special location that is searched by the application when it starts. Plug-in detection is only done on start-up to reduce complexities. Each plug-in defines one or more classes which provide the system with some sort of capability. A plug-in can also define component templates which define a template that describes how of one or more objects and functions should be connected and used to provide some kind of useful capability to the application. The component definition provides information on how the different classes should be instantiated , which objects should be connected to which other objects, which methods form action executors or condition evaluators etc. etc.. None of the objects in a component block are ever shared outside of the block (at least not by the application, object internal code may obviously share information outside the control of the application). The component template may request that instances of external types, i.e. types external to the component plug-in, are provided to complete the object graph. The request to provide an external object is expressed through the request for a specific (base-)type with given characteristics. These characteristics are expressed through meta-data provided by each of the classes. Similar to internal types instances of the external types are never shared outside of the component. When objects are connected they will remain connected for the duration of their lifetime. If a link needs to be changed then the objects will have to be replaced. This simplifies the behaviour of the objects while they are performing their respective jobs.

In short the plug-in classes form the basic building blocks and the plug-in components define how the different plug-in classes should be connected in order to form bigger building blocks.

Plug-in classes can be divided into several groups:

* **Data storage** - These provide the storage of the data and consist of at least one class but potentially more. Examples of this kind of component are geometry and meshes.
* **Transformation** - These will usually be implemented as functions which transform one type of data into another type of data. The transformations do not store any data itself, other than the configuration values for the transformation. Examples are solvers and visualization algorithms.

Although transformations use data storage objects as input and output for the transformation function it is not desirable to connect the transformations directly to the data storage. The main reasons for this is that by not connecting the transformations to the data storage it is possible to replace the back-end of the transformation at (nearly) any point in time, thus allowing for changes to algorithms etc. without having to notify other objects.

When the transformation function is executed three different sets of components will be provided:

* One or more input data storage objects which contain all the data that will be transformed by the function.
* One or more output data storage objects which will store the result of the transformation.
* Zero or one settings objects which provides the configuration for the transformation.

In order to ensure that objects can easily be replaced it is necessary that only the data storage objects can store data directly. Transformation objects that handle data generation and visualization are not allowed to store persistent data themselves, although they can be provided with a state storage object if this is required.

Each transformation object can define a set of properties and events which are indirectly used by the component to communicate with the user interface. The events are raised when the object has changes which need to be reflected in the user interface. At the same time the user interface is capable of displaying the property values to the user and allowing them to alter these. Once the user has provided the correct setting values they are pushed to the settings object and provided to the owning object.

Finally classes can request access to parts of the application that relate to the loading and linking of classes, the instantiation of component templates and the creation and execution of schedules. The application provides one or more classes that provide limited access to these areas.

## History

An important part of the capabilities of Apollo is the ability to undo and redo certain actions. The actual undo/redo storage will be handled by the timeline service. The project system links to this timeline service and notifies it of all actions that happen. In response the timeline service will store a list of actions and the data these actions have changed.

Besides storing the differential changes due to the users actions in the timeline service the project will also provide the capability to store the results of schedule executions at certain points. This allows the project to revert back to a point in the schedule execution. The capability to revert back to a certain stage in the schedule execution can be convenient for cases where:

* The schedule execution fails for some reason. By going back to one of the previous schedule points it might be possible to execute the schedule with more conservative property settings.
* The schedule execution produces warnings regarding the use of a component, e.g. a component is being used outside the boundaries of what it has been verified for. In this case it would be possible to revert to a known-good stage and adjust the schedule to use another component which has been verified for the given conditions.