Apollo

System features

**confidential**

Revision number: 1

Date:

Name: P. van der Velde

System features:

Global:

* The assembly resolvers will be one of the elements in the security system. After all if we can't resolve an assembly we can't load it, and then we can't load any bad code in that assembly. All the systems will run in their own AppDomain so we'll need a few assembly resolvers. Define the directories for these resolvers as:
  + Main AppDomain : base directory, with the bootstrap assembly
  + Project AppDomain: Project directory
  + Experiment AppDomains: Project directory + plugin directories
* All components should be stateless. The only state should be kept in the data objects.
  + This also applies to the UI. The UI design problems in MCS were due to the fact that the UI had to track separate state.
* The system should support undo & redo on user invoked actions. Actions done by the system should not directly be undoable, although they may be undone when a previous user action is undone.
  + Note that this requires that we need undo of data deletes as well. This requires storing the data somehow.
  + Undo & redo can be done through a Version Control System (VCS)
* It should be possible to take snapshots of state at some points and return to these points, either manually or automatically.
* The core system should be based on a collection of services that run at different levels, e.g. plug-in scanner runs at a different level from project services etc. Services can run as background (no connection to global system other than data systems) or foreground (provides commands, other systems can use it).
* All foreground services can provide commands to the system which can be invoked by other services.
* Services have got security and licensing problems:
  + Run all services of a single generate in a separate AppDomain
  + Run all commands in AppDomain of the owning generator
  + Run generator in 2 AppDomains:
    - Generator globals (loaded before a specific project unit is loaded)
    - Generator object specific (loaded when a specific project unit is loaded)
* Most of the API for the system should be command based. By using commands it will be possible for the different parts to easily extend their capabilities by registering a new command.
* Commands could work by creating a tag and feeding that to the command system. The command system can store a dictionary with tag/command pairs. An example of this system is here: <http://geekswithblogs.net/robp/archive/2008/08/05/speedy-c-part-1-optimizing-long-if-else-or-switch-branches.aspx>
* Create a help system that allows components / sections etc. to provide help information for the user. Should be able to attach help to:
  + Components
  + Error types
  + Regions in the system (e.g. experiments)
* The help system should be able to combine information from different sources (maybe only as seealso links).
* The configuration file should have a default version which is stored in the binary. That way we’ll always have it.
* Do not store the framework directories in the configuration file. These will always be fixed so we might as well code them up in a settings class.
* Provide a –reset command line argument which allows resetting the system back to normal.
* Provide an auto-updating system. It should be able to be configured so that the user can determine if the auto-update looks at the default server or a local proxy (allows companies to do a single download but multiple auto-installs).
* Note that security of auto-update could be difficult. Check: <http://www.belshe.com/2004/01/22/security-for-self-patching-software/> and <http://www.codeproject.com/KB/security/xmldsiglic.aspx>

Meta-data:

* Suggested meta data types
  + Attribute – Adds meta-data element to a component
  + Constraint – Constrains and element to …
  + Condition – Element can only be used if the condition is true
  + Info / Description – Stores name, helptip, helpURL etc.
  + Tag – For random use
* Meta-data is needed to store:
  + Capabilities
  + Characteristics – good, bad, average, etc.
  + Advice – do use, don’t use, combine with, etc.
* Some meta-data is fixed at compile time and can thus be set by attributes or other means. Some meta-data is not known until run-time and should be added later.
* Components should be able to set meta-data on themselves and other components. Also it should be possible for components to retrieve that meta-data
* Allow proxying of meta-data. This allows components to forward meta-data and add their own. Furthermore this allows blocking or replacing meta-data.
* Allow meta-data use on:
  + Components
  + Schedule elements
  + Data elements
  + Generators
  + Method parameters
  + Coworkers
  + Settings
* Don’t allow adding meta-data to:
  + Meta-data elements
  + Projects
  + System core elements
* Do not make meta-data tags mandatory but do make it very easy to add them.
* Define specific meta-data markers for:
  + Experiments
    - Coworkers
    - Settings
  + Equations
    - Type – Hyperbolic, Parabolic, Linear, PDE, Differential etc.
    - Discretized form – Strong, Weak, etc.
  + Solvers
    - Stability
    - Robustness
    - Input – Equation, Non-Equation, etc.
    - Output
    - Boundary conditions
    - Initial conditions
  + Matrix
    - Smoothing – Good, Bad, Average
    - General Solver – Good, Bad, Average
    - Matrix types
* More general it should be possible for components to describe the kind of other components they would need. E.g. a physics solver can indicate it needs a PDE solver for specific types of PDEs
* Allow dynamic updates to the stored meta-data, even if the original information came from compile-time meta-data markers

Plug-in scanning:

* Store information about the assemblies that could not be loaded or did nasty things when loading. These should not be loaded again. Do note which assemblies were non-cooperative and provide users with the lists of these assemblies.
* It should be possible to specify constraints on which assemblies can be loaded, should not be loaded or should only be loaded. This might allow placing locks on the application and provides the ability to deactivate parts which are not licensed.
* The plug-in system is needed in different parts (project, experiments etc.) so it would be suitable to design a generic system and allow the different parts to deal with it in a different way.
  + Only run one tracker but allow multiple scanners
  + Allow queries on the stored data, e.g. search for Coworker WITH TYPE = ISilly AND SomeAttribute = GoodSmoother AND AllowsParallelization
* Store plug-in data in a version safe way. This might mean we need to separate the components from their assembly so that we can always find the components back.
* Provide a way to store exchangeable components, e.g. Rhino UI vs OpenNurbs components. This allows exchanging of these components when one of them is not available.
  + Also requires a way to write the information to the serialization stream
  + Also requires a way to load the components in the proper way
* The marking of plug-in classes is too complicated in MCS, this needs to be changed. Have a good look at MEF which should provide a good way of marking plug-ins. Furthermore the scanning probably needs to be improved.

Project:

* The project should have:
  + Generator runners (e.g. simulation) – because only the project knows how many other simulations are running.
  + Persistence
* Experiments should be able to request the different project level systems like runners, persistence etc.
* All project and experiment work should be scriptable.
* Experiments can by copied by the user
* Experiments can be created by the user
* The project should know if an experiment was created by the user or based on the request of a component in a parent experiment. User created experiments can be copied / cloned, system created experiments cannot be copied / cloned
* Experiments can create sub-experiments of their own in order to generate data.
* Create sub-experiments when:
  + Large amounts of data are involved
  + The data must remain available for later processing
  + Changing unique (singleton) data, e.g. geometry, which influences the entire system
* Sub-experiments can be hidden from the user, but the user should be able to view them if so desired
* Experiments can request data from sub-experiments
* Experiment can be run directly by the user or when another experiment asks for it. When system controlled experiments can be run
  + As batch
  + Directly
  + In a postponed fashion
* When dealing with long running sub-experiments it is possible to put the parent experiment in ‘sleep’ mode. The parent experiment is woken up when all the data is available
* Sub-experiments should be locked against user changes because they are controlled by other experiments. Users have nothing to do with these. This also allows users to copy an experiment and its sub-experiments
* Experiments can be linked in a Directed Acyclic Graph (DAG).
* The project defines the base geometry for all the experiments in the project. Individual experiments may introduce changes to the base geometry however no experiment will introduce a completely new base geometry
* The project is the main unit for the user. Users do not normally directly deal with experiments, although they may deal with the data of a single experiment.
* The project will store the base data, e.g. domain definitions, physics etc.. The individual experiments only store the data which is changeable and the resulting data.

Experiments:

* An experiment forms the base for all work done. An experiment consists of:
  + Data
  + Generator(s) - 0 or more
  + Visualization - 0 or more
* Experiments can be large so we should only copy the data if there are changes, i.e. copy-on-write.
* Experiments can contain a mixture of:
  + Real experimental data
  + Virtual experimental data, i.e. simulations
  + Theoretical experimental data
* Experiments should have the ability to be reset (to a specific starting point / save point).
* Changes in an experiment should be undoable. Undo steps should be logical from the UI point of view, not necessarily from the experiment point of view.
  + Note that this requires that we store all state in the simulation in the undo-system so that we can even undo/redo component creation etc.
* Experiments should be loaded in separate AppDomains. That way we can always unload them.
* Define plug-ins for:
  + Component handling
  + Variables & parameters
  + Serialization
  + Visualization
* Experiments should be able to be locked in several areas:
  + Global lock – No changes can be made to the experiment
  + Components – No changes can be made to the components
  + Specific – Lock specific components, variables etc.
* The system used for MCS allowed simple creation of custom simulation tools but it also mixed tool and simulation which has drawbacks in file transportability and just general setup. Apollo should prevent this mixing somehow. Suggestions are:
  + Provide predefined templates, e.g. auto-load a component based on the domain material etc.
  + Provide predefined tool sets, e.g. a collection of components that are always loaded.
* The experiment should store general information about itself:
  + Dimension
  + Goal (??)
  + Physics involved
  + Materials involved
  + Symbols
  + Etc.
* Experiments should provide a large amount of data describing their status, e.g. running, stopped etc. This allows the user interface to provide this information to the user.
* Provide monitors / probes which allow data gathering while an experiment is running. Data that can be tracked:
  + Start / finish times for calculations
  + Memory used
  + Number of components loaded
  + CPU time per component
  + Memory per component
* Experiments should be able to provide time estimates for their duration.
* Allow complex experiments to be used to calibrate more simple experiments. That way the user will be able to use the more simple setups for their problems, without unacceptable loss of precision.
* Experiments should contain sanity checking capabilities. Compare the values stored with some baselines so that it is quickly possible to identify if the values could be good or not. Examples of good boundary values are:
  + Length: Size of a molecule vs size of the galaxy
  + Velocity: Speed of light
* Sanity checker algorithms should be able to be applied everywhere, including input and output.

Experiments – Components:

* The co-worker system should be based on a graph. This allows specifying co-workers for many different combinations. Also provides a lot more control over the different connections.
* Co-worker connections should be requested in a different way, either request through a direct marker (e.g. an attribute/tag) or request specific capabilities (e.g. for data generators). This should allow indirect co-workers which are not directly passed to the component but operate on the same data or operate on required data. E.g. a solver would require a mesh generator without actually ever getting the reference to this generator.
* Need to be able to show information on the experiment and the components that are loaded. Provide:
  + Number of loaded components
  + Which components are loaded and how many of each
  + Connections between components
  + Total memory usage
  + Additional component information
* It could be possible to create proxy-components which are standins for the actual components. These proxies can pretend to be the real component until the component has to be used. At that point the proxy can call out to the knowledge base for the selection of the real component. This could provide the ability to select components based on the actual data in the experiment.
* Allow components to be hidden from the UI, other components and other parts.
* Components should be able to get information about the experiment they are loaded into.
* Components should be able to perform (limited) actions in the experiment:
  + Create / Load other components
  + Connection self / other components
  + Create / Edit schedule
  + Run sub-schedule
* It should be possible for components to have their own plug-ins (just like the mesh in MCS or the meshing algorithms). Users should not be able to load these, the components do.
* Components should be marked according to their group usage so that we can show only the most useful components. Also selections can take place based on these groups. Examples of groups are fluid dynamics / solid dynamics / optimization etc.
* Co-workers and users should be marked to indicate if they can be shared with other components or not. Sharing can be one of:
  + Sharing always allowed
  + Sharing only allowed with data readers
  + Sharing not allowed
* Allow multiple co-worker methods for co-workers of the same type. This allows named co-workers.
* Components should be able to indicate which other components can create the data they need. In some cases only a single component can create the data required.
* Components should be able to indicate which data values they create. That way we can load components based on the data they create.
* Components should define their verification levels. These indicate if verification of the final results are required. Levels could be:
  + Not required
  + Suggested
  + Required
* Allow the loading of all components of a specific type. This is useful if we normally would load those anyway, e.g. mesh algorithms.
* When loading multiple components from multiple assemblies we can load these in parallel and thus speed up the loading process.
* Allow component creator objects to have their own co-workers.
* Components could have a set of characteristics which describe how a component works / what it can do etc. It should be possible to inherit characteristics from other components(?) and that way get more characteristics without having to describe them explicitly (e.g. a PDE solver class inherits characteristics from the solver class, but does not necessarily inherit from the actual solver class.

Generators:

* Generators should be able to add system wide commands to the command set
* Generator commands are:
  + Experimental generator
  + Simulation generator
    - Run simulation
    - Stop simulation
    - Pause simulation
  + Theoretical generator
* The different generator types are stored in plug-ins
* Generators should be extendible by plug-ins to provide more capabilities
* Generators should be able to provide accuracy estimations / bounds on the data they processed.

Simulations

* Define extensions for:
  + Scheduling
  + Verification & Validation
* Simulations should allow verification & validation of the results. Suggested ideas are:
  + Bounds checks on the variables, either independent or dependent
  + Sanity checks on the models, e.g. checking that all values are within the simplification bounds of the model.
  + Standard numerical verifications, e.g. grid convergence, iterative convergence etc.
* Verification should be able to provide error bounds on the final solutions.

Simulations – schedule:

* Schedule elements are components too. This allows us to treat them in the same way as other components. Furthermore they can have properties etc.
* Scheduling system should be based on a directed graph, but not directly programming related.
* Allow multiple schedules to exist. These allow running of actions separately from the main schedule
* Only have one main schedule. This is the schedule that is executed when the simulation is run
* Allow running of sub-sets of any schedule
* Only allow creating sub-schedules if:
  + The resulting data can be stored in the originating simulation
* Allow automatic reordering of the graph to increase performance, reduce coupling etc.
* Schedule elements allow:
  + Specifying pre- and post-conditions
  + Breakpoints – Can be set to activate either upon reaching the element or just before leaving the element.
  + Additional actions taken when the element is first entered and left
* It should be possible to parallelize the schedule both in distributed and local manners.
* It should be possible for components to run their own schedules. This allows interactive running of calculations (e.g. object deformation, kinematics etc. etc.)
* The schedule should consist of different blocks. A block can either be fixed (e.g. the component ordering is predefined and can’t be changed) or flexible (component are executed in an order based on the availability of data). Problems with this last set are:
  + How to stop the sequence
  + How to deal with loops
  + How to indicate what is necessary and what is produced
* Each different block should have a controlling object (which could be the block object). Components should be able to be linked to these controlling objects. This also defines the scope for the different components.
* Schedule parts should be marked as:
  + Run
  + Not run
  + Running – Needed for loops etc. All the parts inside the loop will be marked as running when entering the loop.
  + Invalidated
* The schedule cannot be changed if it is being executed. Furthermore the entire experiment should be locked for changes
* Schedule elements should allow watches which can provide data about the schedule or the data created by the schedule run
* It should be possible to pause or stop the schedule at any point. From this point on it should be possible to resume the schedule if there have been no destructive changes to the section that has already been run.
* Components can mark their actions as ‘MustFinish’ in order to indicate that this action must be finished to prevent data corruption
* Have different levels of stopping / pausing a simulation
  + Immediate stop – Stops the simulation irrespective of any corruption
  + Finish current
    - Node
    - Block
  + Pause direct
  + Pause current
* The schedule should allow starting sub-experiments and waiting for the results of these experiments. Progress is based on the progress of the sub-experiments.
* Components should be able to manipulate the schedule, but only parts of the schedule ‘below’ themselves (i.e. in a sub-block, or happening later in the schedule).
* It should be possible to perform dry-runs on the simulation schedule in order to provide problem detection.
* Components should publish the variables they need and the variables for which they generate values. That way we can automatically schedule the components or component groups based on the usage of values.

User interface:

* Defines an API for providing feedback to the user. In MCS there is very little feedback which may lead the user to conclude that the system has stopped working.

User interface – Knowledge base:

* When selecting components have a preference order:
  + User selection / defined
  + PSE selection / defined
  + Default selection / Defined – Make sure this is always defined.
* Component selection could depend on the scenario that is running and the context, e.g. the selection of the mesh generator depends on the solver, but the selection of the solver can depend on the scenario.
* The knowledge base should be able to determine which physics models are applicable once the user selects a material for a domain. Furthermore an upgrade / downgrade path should be available.
* The knowledgebase should be able to determine which components to load based on the currently defined properties of the experiment.

Components

* Allow links between components:
  + Coworkers
  + Usage / Encapsulate, e.g. inside a data set where one data object encapsulates another data object
  + Construction links, e.g. one element relies on another for construction (not necessarily as co-worker), example: mesh – geometry
* Have several different component types. These may or may not be marked as special.
  + Data components
  + Generator components
    - Simulation schedule components
  + Visualization components

**NOTE:** We don’t want to have to check the different component types in order to perform actions on the components. That just leads to long switch statements etc. So we’ll need to find a way to filter out the different component types. Would LINQ help here?

* Remove the settings and parameters system from MCS. Replace this with properties and events. These should automatically be linked by property classes and event classes. Thus making it easy for developers to provide properties etc.
* Do not allow components to be replaced with other components once the connection is made. All co-worker links are immutable.
* Properties and events should be groupable. A single group can have meta-data describing what the group does. This allows the UI to display a grouped set of controls.
* When using the data-flow system do we still need co-workers? It may be more beneficial to specify the data to operate on during the operation?
* Allow components to be created and destroyed quickly so that there is less penalty on using one. This allows temporary components to be used/created.
* Non-data components should be interchangeable, e.g. we should be able to destroy one object and replace it with another object of the same type without consequences. This will mean that only data components should store state!
* Non-data components CANNOT store state!
* Algorithms, serializers, utilities and visualizers should be reduced to functions (or Action<T>/Func<T>). This promotes replaceability of components.
* Allow a single class to define multiple (non-data) components.
* Allow components to define if they are side effect free / can be shared safely:
  + CannotShare – The object can’t be shared without conflicts
  + CanShare – The object can safely be shared
  + ShouldShare – The object has to be shared as it stores communal state
* Allow components that cannot be removed from the experiment. These can serve as singleton components.

Generator components:

* In order for the schedule to be reordered it is necessary that:
  + Components specify which objects (co-workers, symbols etc.) they change
  + Components specify which objects they need

Data components:

* Allow data components to be invalidated when generating data is changed
* Data components can be grouped which allows them to be treated as a single entity. This enables actions to be performed on the group instead of on individual components
* Data components should indicate if they are singleton components. A change in one of these components will be noticed through the entire system, therefore singleton components may force the experiment to have to spawn sub-experiments if multiple operations on the data are required.
* Provide a hierarchy in the construction of data, e.g. mesh & geometry. This allows invalidation of the lower levels of data (e.g. the mesh) when the higher levels are changed (e.g. the geometry).

Data:

* Allow ghost data to exist. Ghost data is data that is stored but can’t directly be used by other components, except for specifically assigned components. Ghost data will mainly be used for visualization of data that will be created, e.g. mesh generation.
* Allow ghost data to be ‘upgraded’ to real data, that way we only have to create the data once.

Data – Variables:

* There should be a single, unified, way to handle variables in all parts of the system. For a variable we’ll need to know:
  + Type: Scalar, Vector, Matrix
    - Define how values are stored in the system
    - Define how we can access sub-variables, e.g. U[u] and U[v] for the velocity components
  + Number type: Field, Global, Constant, Temporary etc.
  + Define the methods to get and set the number
  + Define the range
    - Define if the range is controlled by expressions or just static. If expression controlled then there should not be a setter, only a getter.
  + Define the default value
  + Define if the variable is available to the user. Some variables should be, other should not be.
  + Define if a variable is free or not. A free variable is one that is not controlled by an expression. Note that variables can be free in some points (e.g. boundary of a domain) while they are not free in other points (e.g. the middle of the domain)
  + Define the unit for the variable
* Define how components can register a variable with the system. The system will only know about registered variables, although components may use unregistered variables.
* Differentiate between a parameter and a variable. A parameter defines a range of valid values; a variable defines a single value.
* Allow variables to be activated and deactivated. Deactivated variables are not used by any components but can be activated. Only user defined variables should be deactivated, component defined variables can just be removed if they are no longer used.
* Variables, and sub-variables, can be linked to other variables, either through a transformation expression or just straight. This allows for aliasing and for conversions of for instance coordinate directions etc.
* Variables can be linked to equations to allow for simplifications, e.g. coordinate transformations
* Allow linking of variable instances to other variable instances, effectively creating a single variable instance. This is useful in periodic boundary conditions etc.
* Provide parametric access to free variables. This allows parameter studies to be done easily. The access consists of:
  + The variable that needs to be accessed
  + The valid range of value for the variable
  + The current value of the variable
  + The step size of the variable steps

Data – geometry:

* Geometry should allow multiple coordinate systems in one ‘model’
* Geometry is stored as virtual geometry in background so that it is possible for components to manipulate this geometry which out changing the global geometry. This also allows the system to only have the useful geometry stored and not all of the geometry (including construction geometry etc.)
* Do geometry transformations, which are immutable, apply to geometry based data, e.g. meshes?
* Geometry should know how to deal with constraints on geometry elements. E.g. two surfaces can be constrained to be mated, thus indicating that they are actually one surface.
* Geometry elements should be immutable

Physics – Models:

* Upgrading and downgrading of physics models in the experiment should be possible. There should be a migration path.
* Combining models should be possible in order to form multi-physics experiments
* Both fluid dynamics and solid mechanics are part of continuum mechanics so it may make sense to start at that level and work down.
* The selection of a physics model determines which boundary conditions are legal for a certain domain. This might also allow automatic assigning of boundary conditions if we create some kind of tagging system that allows users to indicate what certain parts of the geometry are, e.g. wall, inlet etc.

Physics – Materials:

* Material information can be stored in different locations, however it should be transparent to the user where to it is stored.
* Material properties can be just numbers or they can be equations.
* Materials have the following characteristics:
  + Name
  + Symbol
  + State
    - Properties for each state
    - State transitions
  + Chemical make up
* Material properties can be either values or expressions
* Material characteristics are ??????
* It is likely that a material is not a single object but a collection of multiple objects, maybe with a proxy for ease of use.

Numerical

* It should be easy to create surrogate models and use these in calculations just like a normal model. Surrogate models should be able to be re-used / re-purposed. Can use surrogate models for:
  + Initialization of solvers etc.
  + Parameter studies
  + Optimization
* Define a standard convergence checking component. This should be able to:
  + Threshold
  + Stepcount
  + Threshold for x iterations

Numerical – Solvers

* Allow solvers to provide their own accuracy handling systems. A solver should (nearly) always be able to provide accuracy estimation for the values it produces.
* Solvers should have meta-data indicating what the best ways are to accelerate them, e.g. multi-grid, simpler solvers etc.. This allows for the creation of experiments that use the maximum acceleration.
* Solvers should support multi-fidelity approaches

Numerical – PDE solvers:

* Suggested requirements for one (or more) PDE solvers
  + Mesh free kernel, preferably adaptable in radius and shape
  + Moving mesh, possibly based on a mesh moving function
  + Overlapping / chimera mesh
  + Multi-grid
  + Adaptable in both space and order
  + Higher order
  + Capable of multi-scale / multi-resolution calculation
  + Adjoint enabled
* Simulation capabilities should be:
  + Discontinuities – crack formation, shock waves etc.
  + Advection / convection
  + Solution directionality (flow direction makes a difference to the solution stability)
* Should define constants, independent and dependent variables. These can be used to schedule solve orders etc.
* The independent variables should have a level number which indicates solve order, e.g.:
  + - Time = lvl 1
    - X, Y, Z = lvl 2
  + Thus the iteration runs over X/Y/Z for each time step

Numerical – Boundary conditions:

* Boundary conditions should have meta-data describing their assumptions (e.g. internal flow) and the conditions under which the BC is valid (e.g. du/dn = 0).
* Boundary conditions need to indicate how they operate on variables, e.g.:
  + Which variables they need
  + Which variables they limit
* Variables for boundary conditions need much more information than they provided in MCS. For instance:
  + Variable groups – Allows grouping variables by area, e.g. turbulence
  + Unit
  + Aliases – Allows users to fill in the variable they are familiar with. This may also require additional UI components to be created.

Numerical – Mesh

* Mesh elements can be based on geometry elements so that the mesh elements only need to store the relative position on the geometry element. This would allow automatic deformation of the mesh (partly)
* Normal mesh users shouldn’t have to deal with interpolations, they should be able to get the data directly from the mesh (or mesh proxy). In order to achieve this we’ll need to ensure that the solvers which wrote to the mesh provide the correct interpolators.
* There should be a read interface and a write interface for the mesh. Normal users only use the read interface while data producers use the write interface
* Can define mesh decorators, these form mesh proxies thus allowing more capabilities in the mesh. For instance:
  + Moving meshes can be done through a decorator that has links to a mesh generator or a moving algorithm
* Mesh movement can be done on a force-field based strategy where the nodes move with the force-field. The field could have different strengths in different places which can be prescribed by the user or the adaption algorithm.
* The mesh should probably be query enabled. Thus allowing queries to be used to find things.
* Allow mesh users to block certain parts of a mesh from changing. This allows them to work with the mesh without having to check for changes. All changes will be placed in a change buffer and applied once the block is removed.
* Would it be useful to have sub-meshes which have control over their own elements.
* The mesh should be able to deal with both adaptive meshes and overlapping meshes
* In theory a mesh is a model which combines a set of data:
  + Values based on equations solved by the solver
  + Describes the geometry in discrete form
  + Describes the physics in discrete form

Numerical – Mesh generation:

* Mesh generation should be able to include or exclude features of size smaller than x% of a mesh cell in the area.
* Mesh generation should be able to specify that mesh cells are x% of the size of the region
* Allow clustering of mesh elements based on the geometric or expected physical properties of the region, e.g. curvature, shocks etc.
* Base the generation of the boundary meshes on the types of boundary, e.g. put a boundary layer on walls, nothing on symmetry etc.
* Allow mesh generation based on previously generated data, e.g. from simpler simulations, experiments etc.
* Allow marking of specific mesh elements for adaption
* Store mesh generation info on the mesh so that we can always get it back.

Post-processing:

I/O:

* Make the file storage super stable
* Files need to be safe for version changes in the components. In MCS changes in the components can invalidate serialization. E.g. a version change could mean that a component goes from serializable to non-serializable.
* Make sure we write the version number of the assembly file when writing out an experiment to disk. Note that this needs to be the file version number, not the assembly version number because we care about serialization, not component versions.
* All components that write to disk (a.k.a. the data-components only) should use special serialization objects. These objects will be specialized per version.
* All components that write to disk should have a text based serializer for debug and crash rescue purposes.
* Create a single serialization layer through which all data serialization is performed. That allows components (and the system) to not have to worry about serialization procedures and errors. This allows them to focus on the serialization format only.

There is a persistent question on this newsgroup about why Rhino 3dm files are so big. There is a simple answer. We designed Rhino's file format based on the assumption that the information in the files is valuable and storage space is inexpensive.

RHINO FILE DESIGN:

Around half the bytes in the Rhino file are devoted to storing information that makes robust recovery possible. Every single item (layer, point, curve, material, ...) in a Rhino file is isolated in a "chunk" with its own CRC (cyclic redundancy check). When a Rhino file suffers minor damage (random bad bytes from a failing disk, removable media left too close to powerful magnetic fields, USB memory sticks dropped in the toilet), Rhino can still read the file and Rhino even tells you that it detected corrupt items in the file so you know the file has been damaged.

Let' say the file is really trashed by a seriously failing disk. Take a file with 50,000 objects in it, remove the first 20% of the file, and then randomly change another 5% of the values in the remainder of the file. The Rhino "Rescue3dmFile" command can dig through this shredded trash pile and still recover most of the geometry in what is left. (Try this experiment with your brand "X" cad files and you'll get nada. Then contact brand "X" customer support and ask what they can do to help recover the file's contents - you'll get nada and probably be charged for the support call.)

COST OF STORAGE MEDIA:

Rhino 3dm files are stored on delicate media that fail and inconvenient times. A 250GB (giga byte) disk costs around 100USD. That's about 0.0004 dollars/MB (mega byte). A 50MB Rhino file is using about 2 cents worth of storage space.

EVEN MORE FOR YOUR MONEY:

Think about your software investment long term. The value is in the information, not the product. Your 15 year old version of Rhino 1.0 is now a worthless floppy disks in a landfill. However, the files you created 15 years ago with Rhino 1.0 may still be extremely valuable. We give away, for free-no-strings-what-so-ever-attached, robust, portable, C++ source code for reading every version of Rhino file ever created and writing version 2,3, and 4 Rhino files. Anybody in the world can go to http://www.opennurbs.org/ and get this source code. We provide free support to developers, including our competitors, who incorporate this source code into their products. We release a new opennurbs toolkit BEFORE we ship the initial releases of new versions of Rhino so anybody who wants to can read the latest and greatest Rhino files. In short, this means any application written by competent programmers can read and write Rhino 3dm files created by any version of Rhino we have shipped or are currently shipping.