

Application Development Guide for MCRunjob

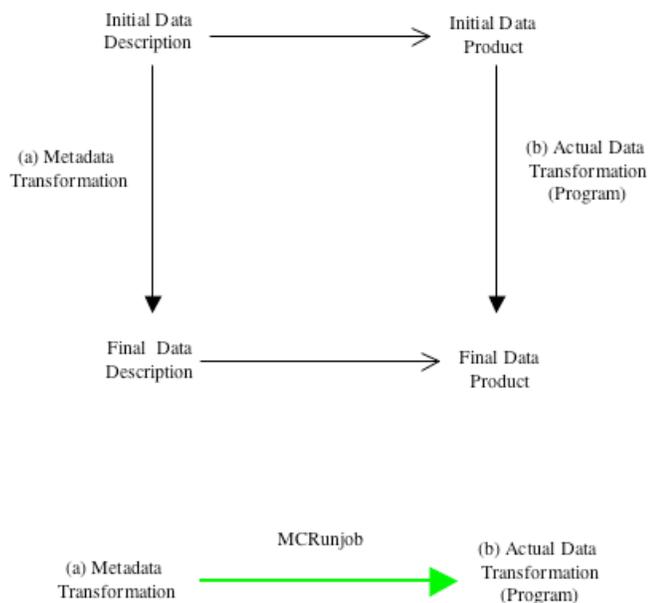
Greg Graham, Fermi National Accelerator Laboratory, Batavia, IL, USA

Introduction

This paper is intended to be a high level guide for MCRunjob developers. It will describe the primary use cases addressed by MCRunjob, the strategy MCRunjob has to address these use cases, and the tools provided within MCRunjob that the developer can use to address the use cases. It is not intended to be a reference to the MCRunjob macro language nor a detailed compendium of APIs or method footprints within MCRunjob. This information will appear in the Application Development Reference for MCRunjob.

The Tao of MCRunjob

The classical CMS use case for MCRunjob is to build jobs for Monte Carlo production. The underlying model is that transformations on real data can be modeled as transformations on metadata descriptions which can be modularized and decomposed.



The job building must take input from a variety of sources, mainly the RefDB for physics parameters and a local file catalog. Furthermore, the jobs must be built in a consistent way across many different execution environments. Finally, a tracking mechanism capable of storing jobs prior to execution, keeping track of submission status, and job

resubmission should also be provided. In general however, MCRUnjob also provides useful tools for modeling complex trees of applications. These include tools to describe application processing steps in terms of application metadata, tools to represent multi-application workflows in terms of their dependencies, tools to synchronize metadata among multiple applications, tools to represent different execution environments, and tools to embed metadata references and metadata dependencies in composable contexts.

The basic constructs used are:

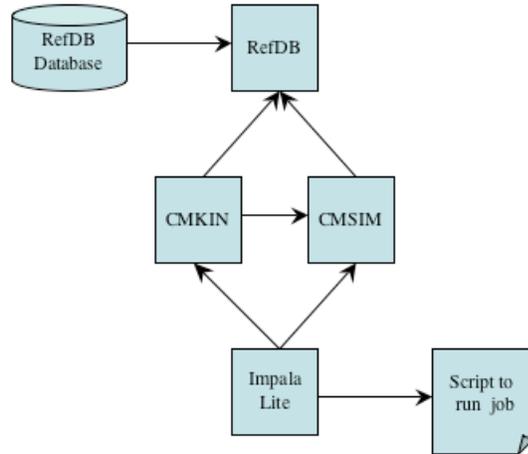
- 1) The Configurator: A Configurator is a data structure that has a key/value pair dictionary, a description/name label, and a list of other Configurator description/name labels that establish a dependency. The Configurator responds to FrameworkCalls (see below) in order to make transformations on the metadata and create programs that accomplish corresponding transformations on the data. (Note: The Configurator is also typically used to hold metadata coming from an external parameter database or to hold metadata abstracting a particular running environment.)
- 2) The Linker: The Linker is a data structure which maintains a list of Configurators in some serial order that satisfies their dependencies. The Linker also generates FrameworkCalls. (See below.)
- 3) The ScriptObject: An object which holds metadata summary of a job instance. This is like a sandbox specification. Scripts generated by MCRUnjob are pointed to by the scriptObjects, but they can hold more generic information also.
- 4) The Framework: The Framework establishes an order on a set of FrameworkCalls, or tasks, to be accomplished by the Configurators.

The following scenarios should help visualize these.

Scenario I: Typical Run of CMKIN and CMSIM

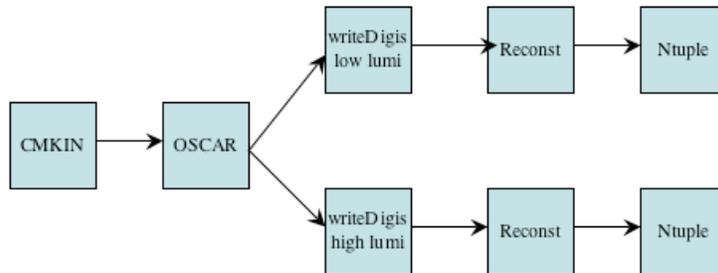
A particular Monte Carlo production job in CMS circa 2002 consists of event generation (CMKIN) followed by a detector simulation using CMSIM. Since this is an official CMS production job, a Configurator is added to the Linker to get job input parameters from the RefDB, and an ImpalaLite Configurator is added to create standard CMS production jobs. The User provides a RefDB Assignment ID and runs a standard framework script. During the “PreLink” FrameworkCall, parameters are retrieved from the RefDB describing the applications to be used, and CMKIN and CMSIM Configurators are added at this time. These are given initial configuration information, such as dataset names and file names. They are also given dependencies on the RefDB Configurator so that they can read the metadata there, and CMSIM is given a dependency on CMKIN since it must be able to read its input file name from the output file name of the CMKIN Configurator. Actual scripts are generated by the ImpalaLite Configurator, so it gets dependencies on CMKIN and CMSIM

in order to read metadata to build jobs. The framework then executes job building calls in a loop until all of the jobs requested by the RefDB have been created.



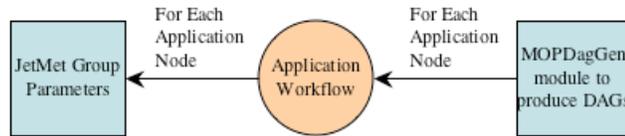
Scenario II: Comparison of Low Luminosity and High Luminosity Monte Carlo

A particular analysis task in the JetMet group requires eight application steps to be run in a certain order. There is CMKIN to generate Pythia events. These events are then run through OSCAR to simulate the detector response. Then the events are run separately through a low luminosity writeDigis and a high luminosity writeDigis. Each digi product is run through an instance of reconstruction, and then through an ntuple maker. This results in the following workflow:



The workflow is specified to be run in the JetMet environment for USMOP. In addition to any User options specified on the command line, the User specifies the following composition “JetMet.ctx:MOP.ctx” for the execution context. JetMet.ctx will add a JetMetGroup Configurator. And for every node given in the application workflow, JetMet.ctx will add a dependency and configure the applications to read parameters from the Configurator. Similarly, the MOP.ctx will add a MOPDagGen Configurator which will build a DAG node for each application node and

create a DAG for submission to DAGMan/Condor-G.



Note that the configuration of the applications should not depend upon the environment in which they are to execute.

MCRunjob Classes

In this section, we will explore the classes available to the developer in MCRunjob. We will attempt to do this from the bottom up, explaining simpler classes first and then build these into more complex classes. However, until they are defined, the classes Configurator and Linker will occasionally be referred to in a loose sense as described above. Familiarity with basic Python built-in classes such as dictionary and list is assumed.

Dependencies

The first classes of importance are the **RequirementSchema**, the **Requirement**, the **RequirementList** classes. The purpose of these classes is to cooperate with each other in order to define dependency relationships among Configurators, and the Requirement class is the main class here. The Requirement class is a key/value dictionary with methods to perform “masked matching”. When checking two key/value pair dictionaries for a match, one would in general check that the dictionaries match in the sets of keys

over which they are defined, and one would check for each key that the corresponding values match. In MCRUnjob, one wants to allow for the ability to specify directly the keys that are important for the match. This allows for comparison of objects that have only a subset of keys in common and for the comparison criteria to change with context. As an extra, the Requirement class also defines a wildcard value '*' which matches any value. If the mask is not given, it defaults to the schema of the object doing the match. Some examples follow.

Object A	Object B	Match Mask	Match?
A=1, B=2, C=3	A=1,B=2,D=4	A,B	Yes
A=1, B=2, C=3	A=1, B=2, D=4	Not given (A,B,C)	No
A=1, B=2, C=3	A=1, B=67, C=*	A,C	Yes
A=1, B=2, C=3	A=1, B=67, C=*	A,B	No

Objects of the Requirement class can be added to a special list called RequirementList which provides extra methods for answering the question of whether or not a Requirement is matched somewhere in a list, or if a list of Requirements is matched for each member in some other list of Requirements. For example, if a Configurator has a certain list of dependencies (Requirement objects), the Linker needs to check that the dependencies are satisfied by making sure that each one is satisfied in the list of Configurators already added (also a list of Requirement objects.)

Finally, a special data structure called RequirementDictionary was implemented in order to support lookup of objects in a dictionary where the keys are Requirement objects and the lookup supports masked matching as defined above. This mechanism is used to support contexts.

ConfiguratorDescription

A subclass of Requirement is the **ConfiguratorDescription** class. This class is special in that it defines a schema for Configurators and defines the string "null" as the default value for keys, but is alike in every other way to Requirement. The particular schema elements chosen are as follows:

- 1) Class: The Class should correspond to the class name of the Python class which implements the configurator in question. It is a system level element and should not be changeable nor removable by the user.
- 2) Version: The Version should correspond to the code version of the Configurator as defined by the ConfiguratorFactory at instantiation time. It is a system level element and should not be changeable nor removable by the user.
- 3) Alias: The Alias is the "name" of the Requirement object that serves to distinguish it from the other objects that may match in all other values. It is a system level element that should be changeable but not removable by the user.
- 4) DataTier: The DataTier is an arbitrary key item that is set by the user to represent a dataset type (ie- generated data, simulated data, digitized data, etc.) that is produced by the application being modeled.

- 5) AppFam: The AppFam is meant to denote the Application Family of the application being modeled. For example, CMKIN, CMSIM, etc.

The ConfiguratorDescription is meant to provide a rich set of tools for describing different kinds of nodes in an application workflow. Unfortunately, because of early mistakes made in the syntax of the MCRUnjob macro language, the DataTier and AppFam components became enshrined into the historical description of all Configurators. This was compounded by the lack of an interface to the ConfiguratorDescription class itself that allows for the easy addition or removal of schema elements where allowed. The first error has been remedied by the inclusion of namespaces into the macro language (see below.) However, the second issue remains to be addressed. This situation has unfortunately led to some confusion: for example, an InputPlugin for the RefDB has no “DataTier” nor “AppFam,” but these have been overloaded in order to work with the pre-existing macro syntax.

NameSpaces

In the “Tao of MCRUnjob” section, it was alluded to that each Configurator contains a metadata key/value dictionary for describing application steps and a ConfiguratorDescription object. As will be seen later, the Linker also imposes a constraint that elements in the collection of ConfiguratorDescriptions over all Configurator attached to the Linker must be unique within that set. In other words, you can’t have two Configurators that match in every element. The Alias is generally used to guarantee this uniqueness. Then the ConfiguratorDescriptions together with a metadata element name specify a unique value within the Linker. (CfgDesc,key) → value.

In order to simplify the process of referring to a metadata element, namespace aliases can be defined. Namespace aliases are different from the Alias element of the ConfiguratorDescription above. These consist of a mapping, maintained in an instance of the **NamespaceTable** class, of string names to ConfiguratorDescriptions and matching masks. For example,

Namespace MyNamespace Class=A DataTier=B AppFam=C
Will create a namespace called MyNamespace with ConfiguratorDescription given by DataTier=B,AppFam=C,Alias=null,Class=A,Version=null and matching mask of DataTier, AppFam, and Class. Subsequent operations using MyNamespace will be applied to all Configurators matching MyNamespace.

Developers who are familiar with MCRUnjob will be familiar with the notation

Define X ::DataTier:AppFam:Y

This notation doesn’t scale. Though it is retained for backwards compatibility, the ::namespace:Y notation is preferred. For convenience, namespaces defined only on the Class attribute of ConfiguratorDescriptions are pre-defined.

Value Lookup and Synonyms

Both the Linker and Configurator classes offer APIs to lookup a value based upon ConfiguratorDescription and key as described above. In the linker, this is straightforward. A Configurator however is constrained to have a dependency declared on the Configurator it is attempting to read from. In the following, let the C be a ConfiguratorDescription and Y be some metadata element being looked up.

While the pair (C,Y) uniquely identifies a value within the linker space, we still have a useful piece of information that is not yet used. In many cases, the value to be looked up will be assigned to a local metadata element X, as in $X=\text{lookup}(C,Y)$. We can use this extra information to define a synonym. Namely, if we store the relation $Y=(X,C)$ locally in the Configurator, (this is done in an instance of the SynonymTable class,) then the lookup can be done on C alone, as in $X=\text{lookup}(C)$.

The complete value lookup algorithm is as follows:

- (1) Specification of local metadata element X and an external reference R.
- (2) Namespace Lookup: Extract C from R, and check if C is a namespace or a straight up ConfiguratorDescription. If it is a namespace, then lookup the namespace in the Linker and replace C with its equivalent ConfiguratorDescription.
- (3) Synonym Replacement: If R is of the form “lookup(C)”, then attempt to lookup $Y=(X,C)$ in the local Synonym table. If this fails, assume $X=Y$. (Ie- just assume the local and target metadata elements have the same name.) If Y was given, then do nothing.
- (4) In order to check dependency, search the local dependency lists for a match for C. If there is a match, proceed. If C does not specify an Alias, then change C to C', which is C with the Alias from the matching ConfiguratorDescription in the requirements list. This is an important point: it allows us to model workflow in terms of abstract attributes and delay the Alias lookup until the last minute.
- (5) The complete (C,Y) reference is passed to the Linker, which either returns a value or throws an exception if the key is not found.

This behavior is implemented in the Configurator.GetValue method (steps 1-3) and in the **NameResolver** function (steps 4-5). The name resolver function itself is implemented as a read-access trigger on the Configurator metadata dictionary. (See section “TriggerDictionary”) below.

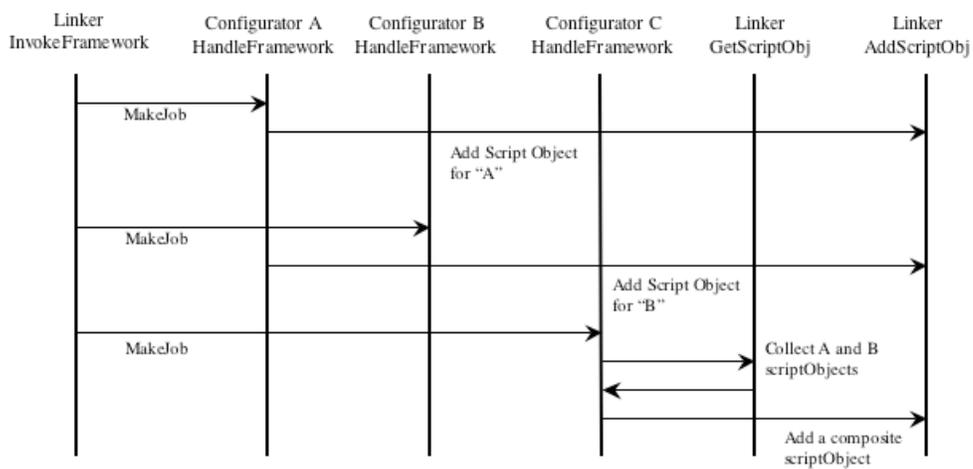
The Framework

A **FrameworkCall** merely consists of a unique string, though it is helpful if it is description of some task or stage in job creation. The Linker will generate FrameworkCalls and pass them to each Configurator during normal operation. The Configurator will handle each FrameworkCall through one of three ways:

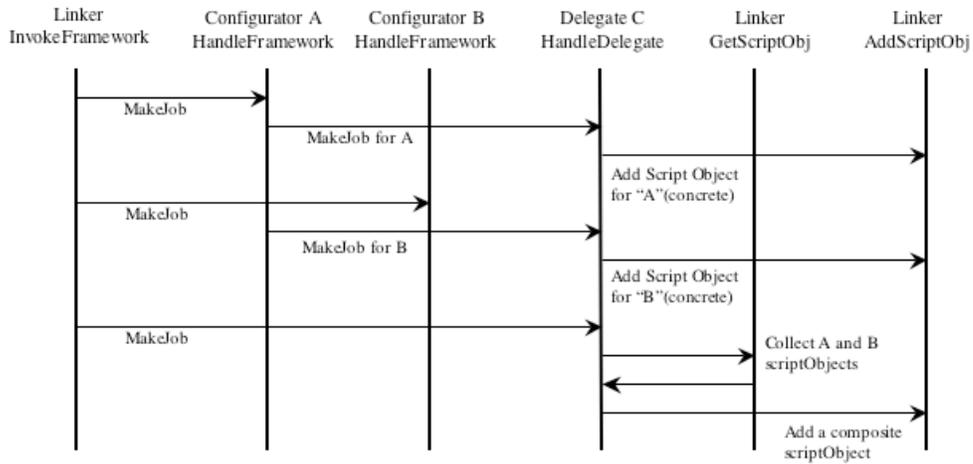
- (1) Through inheritance, the Configurator can modify its FrameworkCall method to specialize its response to a set of calls.
- (2) Through registration of handlers to the FrameworkCall interface, FrameworkCalls can be redirected to externally defined functions. This is more modular, and

allows the same code to possibly be reused in several different Configurators to handle cross-cutting functionality. This also implicitly contains case 1 in that a simple catch-all handler can be registered and defined to do exactly what the inherited function would have done.

- (3) Through explicit delegation of the FrameworkCall to a delegate Configurator. (This is called the **ScriptGen** interface, or **Delegate** in ShahKar.) The advantage of this scenario is principally that all framework handling functions that externally must satisfy certain constraints can be localized to a single module, for example ImpalaLiteScriptGen.



Classic case: Configurators (A,B) create Application Specific scriptObjects which are collected by a ScriptGenerator (C) into a composite scriptObject.



Delegate case: Configurators (A,B) call on Delegate (C) to create Application Specific scriptObjects which are collected by a ScriptGenerator (C) into a composite scriptObject.

A FrameworkCall handler has access to the Configurator that is handling the call and to the name of the call being executed. It can read the metadata from that Configurator, create scripts, XML, DAGs, etc. It can also contact external databases for input data or tracking purposes. A Delegation handler has access to both the Configurator handling the call and the Configurator for which the call is being handled. Also, the Delegation Configurator also traditionally gets the job of collecting scripts into a composite script object or DAG.

In addition to calling a handler function on receipt of a FrameworkCall, the Configurator also is able to store macro commands to be executed just prior to a handler call or just after. These stored commands make many tasks easier and are part of the overall customization of Configurators. (See the section on Macro Parsing below for an explanation of macro commandss.)

FunctionObjects

Externally defined functions can be registered to perform several duties, including handling of FrameworkCalls as described above. In general, any registered function is accessed through a **FunctionObject**. The FunctionObject class provides methods for adapting interfaces by maintaining lists of expected positional arguments and keyword arguments. It also provides an interface for specifying a function that is either defined in the same scope as the FunctionObject, in which case a function pointer is given, or a

function that is not in scope, in which case the module name and function name are given.

The ConfiguratorFactory

Configurators are instantiated by a **ConfiguratorFactory** class. The ConfiguratorFactory class contains a list of module names and class names that are indexed by the Class and Version attributes of a ConfiguratorDescription. When a particular Configurator is chosen, then the class is loaded from the desired module. This was done in order to facilitate dynamic loading of new Configurators. Later, in an environment that may contain a mix of different versions of Configurators, this class also may provide a central point for information on compatibility.

MCRunjob developers may have noticed that in addition to loading Configurators in this way, some special functions such as command parsers and name resolvers are also loaded this way. These are core pieces of MCRunjob, and the idea was to be able to change behaviors dynamically. This was not a good idea and will be discontinued in the future so that ConfiguratorFactory is once again a factory for Configurators only.

The Trigger Dictionary

The **TriggerDictionary** class is a class which extends the basic Python UserDict class by allowing the developer to add triggers to the normal `__getitem__` and `__setitem__` interfaces. The triggers are either global, in which case any access results in a call to the trigger, or indexed, in which case only access to specified elements cause the trigger to be executed. The NameResolver described above is implemented as a global read trigger. This provides for lazy resolution of inter-Configurator references.

The Macro Parsers - 1

The primary way in which a user interacts with MCRunjob is through a macro script. By “macro script,” it is simply meant that the scripts do not (yet) rise to the level of a real language and are just shorthand expressions for invoking methods in the Linker or Configurator APIs. Parsing is a complex two step process in MCRunjob and involves many classes. The aim of the parsers is to provide a basic look and feel to the macros while providing flexibility for Configurator developers to define their own macros locally without modifying a central module. Logically, all macro commands are interpreted by the Linker. The Linker determined whether the macro is to be executed directly by the Linker, or, if a ConfiguratorDescription is given, by all matching Configurators that are attached to the Linker. If it is the latter case, then the macro is simply passed along to the matching Configurator(s). A Configurator then has registered functions which will handle (or not) the macros from the Linker.

In the following section, the process of reading a macro script into an internal representation, first pass parsing of the text commands therein, and making the commands available in a structured way to the Linker and Configurators is described.

CommandObject and CommandParser

The **CommandObject** class is initialized with a text representation of a macro line. The **CommandObject** also contains a reference to an instance of **CommandParser**, which contains a method to tokenize the text command and draw out options, if any, to the command. The tokens are accessed through a stack-like interface while the options are accessed through a dictionary-like interface. Tokens are generally accessed through the `GetNextToken()` method which consumes a token off of the stack, or the `Peek()` method which returns the value of the token on top of the stack but does not consume it.

NodeIterator and TreeNode

The **TreeNode** class is a basic unit for building a tree where any node can have multiple children. It contains pointers to the parent and to an ordered list of children nodes. It implements the **NodeIterator** interface which adds methods to `Reset()` the node and iterate over children. Using the basic interface, a depth first search is implemented.

CommandTree

The **CommandTree** class is a container for **TreeNodes** that contain **CommandObjects** as payload. In addition to holding individual **CommandObjects**, the **CommandTree** adds block structure to the macros. It defines the following special tokens as beginning a block of code: `if`, `repeat` (also a looping block), `contextBlock`, and `frameworkCallBlock`. Blocks are implemented as subtrees and are therefore nestable. The only block ending token is `end`. The processing of macro scripts into **CommandTrees** proceeds as follows:

- (1) An empty tree node is created to be the root node for that script. This node is labeled the current node.
- (2) Lines of text from a macro file are cleaned up (comments, extraneous whitespace, etc) and individually wrapped in a **CommandObject** and **TreeNode**. The **CommandParsers** are invoked at this time to tokenize the text command.
- (3) If the **CommandObject** contains an ordinary token or a begin block token it is appended as a child to the current node. In the case that it is a begin block token, then the new child node additionally becomes the current node.
- (4) If the **CommandObject** contains the end token, the parent becomes the current token.

Each macro script or context script gets its own command tree.

When all text commands have been added to a tree, it can be reset and the **CommandObjects** come out of it using the `GetNextNode()` method. `GetNextNode()` implements a modified depth first search of the nodes that, without looping or conditional evaluation, merely returns the commands in the same order as they appear in the macro script file. However, this procedure is modified in two respects. First, all looping blocks are evaluated with respect to loop counters so that the contents of loop blocks are reset and returned as many times as called for in the loop constant. Second, conditional blocks

are evaluated with respect to an external Eval() function. In the case of MCRUnjob, Eval() is implemented in the Linker which has global knowledge. (NOTE: loops are also sent to Eval() to check if the Linker set a stop flag. This is useful when a loop constant is not given.)

Additionally, nodes inside of blocks are provided with something called “block context.” This is distinct from MCRUnjob context as presented below, and refers to the ability to pull node options and block commands of all parent blocks out of the command tree. This is merely a walk from the current node to the root node. However, CommandTrees executed from within other CommandTrees (ie- with the “source” macro,) do not remember the context of the calling tree.

As a brief review, the CommandTree and its helper classes provide for a uniform look and feel to the macro scripts. They also provide block structuring with conditional blocks, looping blocks, and block context.

Macro Parsing – II

The Linker opens MCRUnjob macro scripts and reads them into CommandTree objects, one per macro file. It then pulls CommandObjects out of the CommandTrees in the modified DFS order described above. Once the Linker has this CommandObject, it interprets the macro by invoking calls on it’s own interface or on the interface of one or more Configurators. This process is the focus of this section.

Parsers in General

A macro parser in either the Linker or a Configurator is a registered FunctionObject that takes a CommandObject and a reference to a Configurator as arguments and based on the contents of the tokens of the CommandObject performs some series of actions on the Configurator’s interface. When done, it must return 1 if the CommandObject was successfully handled or 0 if not. (It is logically and physically distinct from the CommandParser above.) The Parsers are generally prepended to a list in the Configurator constructor so that derived class Configurators “inherit” the parser functions of their parent classes while shadowing like behavior of the parents. The Linker has only one parser, though theoretically it could also have prepended parsers as well. When a CommandObject is being parsed, it is sent in turn to each registered parser function in the list until one of them returns 1.)Note that the current implementation is to send only the original text commands to the parsers instead of the full CommandObjects, but logically it is the same thing.) The LinkerParser is special in that it must contain a macro for sending a CommandObject along to a set of Configurators. The method for doing this is ToCfgrtr(cfgDesc, cmdObj, matchMask) which sends the CommandObject cmdObj to all attached Configurators that match (cfgDesc,matchMask).

A Configurator developer can thus inherit from the Configurator base class, extend the derived class Configurator interface, and provide a macro interpreter function that provides macros to use the extended interface.

Putting It All Together

The Configurator

The **Configurator** class consists of the following:

- (1) A class that inherits from TriggerDictionary. This provides the Configurator with its basic key/value pair dictionary for metadata and allows for system defined behavior in referencing values in other Configurators,
- (2) An algorithm for looking up metadata values in other Configurators, including a SynonymTable for automatic mapping of metadata elements to metadata elements in other Configurators.
- (3) A ConfiguratorDescription object
- (4) A list dependencies, implemented as a list of ConfiguratorDescription objects
- (5) A method to register functions to respond to FrameworkCalls in order to produce scripts with stored macro execution just before or just after.
- (6) A method to register functions to parse and interpret macro commands provided in a script by the User.

The benefits of a Configurator lie in its ability to describe an application or a task, order itself among other Configurator based on dependencies, and customize itself in order to accomplish some task in cooperation with other Configurators.

The Linker

The Linker consists of the following:

- (1) A Container class for Configurators with a method to add new Configurators.
- (2) A ConfiguratorFactory for instantiation of Configurators.
- (3) A NamespaceTable for easy addressing of Configurators
- (4) A method for specifying and dispatching Framework calls.
- (5) A tree system for organizing macro scripts into a block structured language and dispersing macro commands to Configurators.

MCRunjob Contexts

The Configurators and the Linker offer a rich array of options and flexibility in the face of complex systems. Experience has shown during CMS production that using macro scripts alone, while expressive enough to describe production jobs, is not sufficient for easily mastering the complexity. A number of ad hoc scripts have been developed over the past two years that basically do the following tasks:

- (1) Driver scripts which test for certain conditions and add/modify Configurators in response
- (2) Helper scripts which set Configurator options and metadata values to some sensible default.

(3) Helper scripts which maintained synonym relationships between any two Configurators.

The problem is that there is no structured way to combine all of the configuration options, synonyms, dependencies, etc. that need to be coordinated in order to pull off successful production job creation; so each environment typically gets its own suite of Driver/Helper scripts. What is needed is a way to organize the rich configuration options into data structures that can be combined with one another.

The Linker therefore contains a new data structure called the context. The context is an instance of RequirementDictionary as described above. The purpose of the context is to store configuration macro commands keyed by ConfiguratorDescription and matching mask. When a Configurator is added to the Linker, all matching configuration macros are retrieved and executed. The user interface to contexts is like an ordinary macro script. Configuration commands are specified in a context file and maintained by the Linker in a CommandTree object. Multiple context files can be specified at once however. Before the regular macro script is executed, each context CommandTree is executed in order of specification and loads the Linker.context data structure. This means that later contexts can shadow earlier ones. Also, context CommandTrees are also allowed to add Configurators of their own. For example, CMSProduction.ctx will add an ImpalaLite Configurator for the standard CMS Production environment. This can be combined with a following MOP.ctx which will add and configure a MOPDagGen module to enable the ImpalaLite scripts to run in the MOP environment.