

Simulation Review

Jim Kowalkowski, Marc Paterno, Pekka Sinervo

1 Introduction

This document contains a summary of our review of the CDF simulation software. We have concentrated on the plan for redesign summarized on the web by Pasha Murat, and presented at the review kick-off meeting by Chris Green. The plans for redesign include integration with the CDF geometry system, initialization of the GEANT3 geometry from the CDF geometry system, and the introduction of a framework for digitization. In conducting this review, we did not study details of the existing simulation code for the purpose of commenting on its content and structure. We have concentrated on understanding what the current simulation package does, and have considered what design modifications might enhance the maintainability and extensibility of the code.

In this document we cover, from a high level, the following “tasks” performed during simulation:

- the presentation of the CDF detector geometry to the physics simulation engine (currently GEANT3);
- the handling of callbacks from the physics simulation engine during the propagation of particles through the detector, to register the required “digitization” information in the detector elements;
- the finalization of the simulation of an event, including such tasks as smearing and the creation of raw data objects for each subdetector.

We have concentrated on making the execution of these tasks simple from the point of view of the implementor of the code for a subdetector, and secondarily from the point of view of the implementor of the simulation infrastructure. We have also concentrated on enhancing the speed of the callback process, since that is the most time-critical part of the system, and upon the ease of configuration of the entire system. We make few comments on the subject of recording “MC truth” information about the simulated particles, due to lack of time.

2 Overview

The old simulation system¹ contains several elements that are incompatible with the current CDF infrastructure code. The new simulation framework design addresses many of these incompatibilities. For this reason, our overview of the system and our proposals address the new system.

The new framework consists of two AC++ modules:

- *SimInitManager*, which is responsible for the initialization of the GEANT3 geometry, and
- *SimulationControl*, which is responsible for the digitization and readout of the detector elements.

This framework defines a common interface for digitizers that provides for common ways to digitize a hit and to populate the event with raw data objects.

During the course of the review, we tried to simplify the system by demoting *SimInitManager* from an AC++ module to a tool intended for use by one or more modules. In the end, we came to agree that *SimInitManager* should remain an AC++ module, because of the requirement that simulation and extrapo-

¹ In this document, we will refer to the body of code written primarily by Pasha Murat as the “old” system, that written by primarily by Chris Green as the “new” system, and the design proposed in this document as the “proposed” system.

lation use a consistent description of the CDF geometry. The most convenient method of assuring that the initialization is done once is to have a framework module control the initialization. The status of *SimInit-Manager* as a subclass of *AppModule* also provides for a talk-to interface for the configuration of the simulation geometry (*i.e.*, the declaration of each detector element as either “active” or “passive”).

We believe that Chris has valid concerns about the lookup time involved in locating the correct digitizer in the particle propagation procedure. We believe that improvements can be made in this area. These improvements will come at a cost; that cost being higher memory requirements and a more complex infrastructure. We believe the benefits will outweigh the costs.

2.1 Definitions

In this section, we present a few definitions that will be used throughout the remainder of this document.

Concept and **model**: These are *generic programming*² terms. In the context of this review, a concept can be thought of as a list of type requirements. A type *T* is said to be a model of a concept **C** if *T* satisfies all of the requirements of **C**.³

For example, one of the basic concepts introduced with the STL is the concept **Equality Comparable**. A type is **Equality Comparable** if it is possible to compare two objects of that type for equality using *operator==*, and if *operator==* is an equivalence relation (*i.e.*, it satisfies the requirements of identity, reflexivity, symmetry, and transitivity). All the primitive types of C++ are models of **Equality Comparable**. User-defined classes are models of **Equality Comparable** only if they have defined, either as a member function or as a free function, *operator==* with the required properties.

In our design proposal, we define several concepts for which the each digitizer class (written for each sub-detector) must be models.

Digitizer: a digitizer class is a class that models the concept **Digitizable**, described in Section 4.1.4. A digitizer is an instance of such a class. The purpose of a digitizer class is to turn the information generated by the simulation engine during particle propagation into hit information in the simulated detector elements. Additionally, each digitizer class must provide a series of typedefs required by the proposed system.

2.2 Assumptions

The design changes outlined in this document required us to make certain assumptions. We list the assumptions here, to make them explicit.

- A structural change in the CDF geometry tree must be followed by a complete rebuilding of the simulation geometry and the digitizer framework. By structural change we mean deletion or insertion of physical volumes or detector elements.
- Each CDF physical volume corresponds to exactly one physical volume of the simulation engine (*e.g.*, GEANT3). Each physical volume is contained within exactly one *CdfDetectorElement*; no physical volume spans two *CdfDetectorElement* instances.
- It is acceptable to have a one-to-one relationship between *CdfDetectorElement* instances and digitizer instances. This is different from both the old and new designs, which contain many *CdfDetectorElement* instances for each digitizer.
- Every digitizer class is associated with exactly one raw data class. This raw data class must be a *StorableObject*, so that it may be added to the event. The raw data class implementation is up to the developer. The calorimeter, for example, could choose to use the *CalorData* object (based on

² For an excellent introduction to generic programming, see **Generic Programming and the STL**, by Matthew H. Austern (Addison-Wesley, 1999).

³ Throughout this document, we will use *italics* for class names, and a **bold sans-serif font** for concept names.

tower energies) instead of using the D-Banks directly. The *CalorData* object produces raw data banks on demand from the tower energies. The silicon detector can use the *SiStripSet* in a similar fashion. It is also possible for one detector subsystem to have more than one kind of digitizer.

2.3 Required Modifications to Related Systems

The design changes outlined in this document require the system to behave in the following ways:

- Instances of subclasses of *CdfDetectorElement* must be able to return the name of their class. This feature is needed to support our proposal for configuration of the system.
- Factory registration of digitizers must be permitted. This registration will be similar to the method used by ROOT and the calibration database subsystem. The coupling of digitizers and the digitizer framework and simulation control module will be in name only. This implies registration using standalone object files, shared object libraries, or the presence of special entries in the users *AppUserBuild* method.

3 Major Areas of Concern

Our primary concern with the “new” system is the complexity of the task faced by someone responsible for the introduction to the simulation of a new subdetector, or that faced by someone responsible for introducing a new method of digitization for an existing subdetector. As a secondary issue, we were concerned with the efficiency of the mechanism that performed the lookup of callback functions during the process of particle propagation. Third, we did not understand how the system would be started except through hard-coding of the system configuration by end users, which we judged to be too inflexible. Finally, we thought that the relationship between one digitizer and one subdetector system was too rigid.

The design we propose addresses each of these concerns. As a side effect, the mechanism by which the raw data objects are produced is more complicated than in either the old or new designs. We think that the added flexibility is worth this added complexity.

In this section, we address each area of concern and describe how our proposed design deals with it. In Section 4 we present our proposal in more detail.

3.1 Configuration of simulation by user:

Since the simulation will be used by more than just its designers and implementers, care must be given to make it easy to run in a variety of situations.

- It must be easy to specify the values of the parameters (*e.g.*, smearing constants) for each of the digitizers.
- It must be easy to specify which digitizer class will be used for each subdetector components.
- It should be easy and efficient to run several digitizers simultaneously for a given detector element. This allows simple comparison between digitizer outputs; for example, to compare two different calorimeter energy smearing choices. This would allow a comparison to be done at the hit-by-hit level, rather than only allowing comparisons of distributions.

Our proposal describes the configuration of the system using a simple ASCII table, perhaps read from a file. It would be possible (though perhaps more complex, with little gain in functionality) to replace this table with a set of AC++ parameters, read in the form of a TCL script.

In addition to describing the large-scale configuration of the system, it is necessary to set the parameters of each individual digitizer. This should be handled by a standard mechanism. In our proposal, we indicate this by having each *SimElement* instance contain an *APPCommand**. In our proposal, these *APPCommand* pointers are managed by the *SimulationControl* module, and can thus be used in the configuration interface of that module. The exact mechanism through which this is to be done must be determined in consultation with the infrastructure group.

3.2 Creation and Integration of digitizers:

It should be easy for a developer to write a digitizer that will integrate easily into the system. To achieve this goal, we have introduced the concept **Digitizable**. **Digitizable** defines a uniform interface to which all the digitizers must conform, and which assures that the digitizer can be created, configured, and used by the system without modification.

3.3 Efficient Stepping in the Simulator:

3.3.1 Digitizer Lookup

We will call the process of particle propagation in the physics simulation engine *stepping*. The stepping procedure in the simulator be called thousands of times per simulated event, so it is essential that this process be efficient. One of the concerns raised during the initial presentation of the new system was that process of locating the appropriate digitizer might be too slow. Our proposal employs a larger number of digitizer instances, so without other changes the problem would be even more severe. For this reason, the digitizers must be organized in a data structure that permits a minimal amount of searching in the stepping procedure.

3.3.2 General Concerns

Because it will be called so frequently, it is important to avoid needlessly repetitive tasks in the stepping function. To give a specific example, in *gustep_simTest()*, we see something which we must be careful to avoid. The code fragment in question is the following:

```
// We're going to need the outer radius of the COT to decide whether
// or not we are interested in secondaries:
const CdfTubs* cotContainerShape = dynamic_cast<const CdfTubs*>
(( *CdfDetector::instance()->
getCotDetector()->
beginningPhysicalVolume()->
getLogicalVolume()->
getShape());
double cotOuterRadius=0,cotZHalfLength=0;
if (cotContainerShape)
{
cotOuterRadius=cotContainerShape->getOuterRadius();
cotZHalfLength=cotContainerShape->getZHalfLength();
}
}
```

It seems that the value retrieved from the CDF geometry is the same at every step. It would be more efficient to obtain the value of `cotContainerShape` once, and to cache the value. The value only needs to be recalculated when the geometry is changed (at which time the entire digitization system needs to be rebuilt).

Also, while the code above is careful to use a `dynamic_cast` on the pointer returned by `CdfDetector::instance()`, and to use it only if the `dynamic_cast` is successful, what action is taken if the `dynamic_cast` fails? The system should probably fail gracefully, indicating to the user that a configuration error has prevented proper initialization of the geometry.

3.4 Ability to swap out the simulator.

The ability to change to a new physics simulation engine such as GEANT4 in the future is highly desirable. In the new system, the scope and magnitude of code modification required to introduce a new simulation engine is unclear. In our proposal, we have isolated the dependency on the simulation engine to the concept **STEPDATA**. As long as the simulation engine proceeds by propagating individual particles by steps, and is able to identify the physical volume in which each step occurs, it should be possible to use the simulation engine in the proposed framework.

3.5 Monte Carlo Event Data Storage

The output from the generator and the particles generated by the simulator need to be recorded in objects compatible with the CDF EDM package. Because the simulator will need to add new particles generated by the simulation process to the event, and perhaps to modify particles in the original input, special care needs to be taken to deal with the immutability of *StorableObjects* in the event.

3.6 Initialization of geometry

The *SimInitManager* AC++ module must use the information in the CDF geometry system to initialize the physics simulation engine (GEANT3) geometry. Because *SimInitManager* is an AC++ module, it is probably not important for it to have the flexibility to be able to produce the geometry for another simulation engine (e.g., GEANT4). Instead, a different class (perhaps *SimInitManagerG4*) could be introduced when needed.

3.7 Control over the simulation.

The *SimulationControl* AC++ module must be able to initiate the configuration of the digitizer framework, initialize and feed the simulation engine (GEANT3) the particle information, and control the digitizer read out. In addition, it must be able to allow the user to interact with the digitizers and the simulator.

3.8 Raw Data Generation

SimulationControl must produce raw data objects (instances of subclasses of *StorableObject*), and insert them into the event. In the new system, this is done at the subdetector level. A more fine-grained approach, which allows for the generation of different formats for the raw data from each subdetector, would be preferable.

3.9 Random Numbers and Histogramming

During the first review meeting, it was stressed that a facility for generating random numbers according to an arbitrary distribution (described by a frequency distribution encoded in a histogram) is needed. It should be possible to gain this functionality at slight cost (perhaps by a slight enhancement of the ZOOM random number generator product). It should not be necessary to add the entire machinery of ROOT, merely to be able to use a subclass of ROOT's *TH1* for the generation of random numbers.

4 Design Recommendations

4.1 Overview

The design we propose is actually a framework for performing detector simulation. This framework makes extensive use of C++ templates, to be written by the core members of the CDF simulations group. This design allows the task of writing the detector-specific code to be as simple as possible, while retaining a great deal of flexibility.

This flexibility is achieved mostly by having the class responsible for the event processing (*SimulationControl*) define a series of functions describing the pieces of the tasks to be performed. The digitizer classes, to be developed by the subdetector experts, implement these functions. This allows new digitizers to be introduced without requiring widespread changes in the system.

We have chosen to base our design largely on the use of templates (generic programming) rather than solely in object-oriented terms, in order to gain maximum efficiency while simultaneously achieving loose coupling between classes and retaining strong type safety.

One of the goals we had in mind when laying out our proposed design was to reduce the number of abstractions that the developer is required to understand. Another goal was to reduce the number of levels in the inheritance hierarchy. We believe that a framework cast into a templated design fulfills many of these

goals. We also believe that it is easier to write high-level tools that utilize the framework elements directly than it is to write a single, high-level, generic object that works with a bunch of abstract elements.

4.1.1 Generator and Simulator Output

The collection of particles presented to the physics simulation engine and the collection that is a result of the processing done by the simulation engine must both be contained in classes inheriting from *StorableObject*, so that they may be stored in the event. Rather than inventing new Monte Carlo event classes, we recommend working with the Computing Division simulations group to develop the C++ version of STDHEP. If collaboration between the CD simulation group and CDF can be arranged, we will of course be ready to assist with the development of the design.

We recommend that the *StorableObject* output of the generator be placed into the event and not be changeable by the simulation. We recommend that the output of the simulator be an object that is placed into the event and follow the same format as the output of the generator. The output of the simulator should refer to data in the generator output object a CDF EDM compliant manner.

4.1.2 Geometry Initialization

The purpose of the classes in this subsystem is to initialize the CDF geometry and to present that geometry to the physics simulation engine. To achieve optimal efficiency in the stepping process, the digitizer framework design outlined in this document requires that a list of detector element / physical volume pairs be created. The digitizer framework configuration process is driven by names and IDs of detector elements and physical volumes. As the tools in the section are walking the CDF geometry tree and creating the simulation specific physical volumes, this large list must be created and stored in a place that simulation control can access.

4.1.3 Simulation Control

The class *SimulationControl* is the main coordinator. It has the job of constructing the elements of the simulation framework, and of configuring them. It must initialize and configure the digitizer framework. It must feed data to the simulator and round up output from the digitizers and the simulator.

4.1.4 Digitizers

Subdetector experts are the users of the simulation framework, in that the classes they define use the features of the framework to define the high-level *SimulationControl* class. We'll refer to these individuals as *developers*, to distinguish them from the *users* who will run the simulation, but do not develop new digitizers.

Developers create digitizer classes. In this system, there is no base class for a digitizer. A digitizer is a class that conforms to a set of rules, defined by the concept **Digitizable** (described in Section 4.2.4). Conforming to the rules allows the digitizer class to be plugged into the system. The specific rules are explained in the next major section of this document. A digitizer, in general, must describe four pieces of data that it will work with:

- The physics simulation engine stepping data structure. This is the simulator-specific data available at a given step. It could be as large as the entire suite of GEANT3 common blocks or as specific as a translated physical volume coupled with the energy value, or the appropriate class defined by GEANT4.
- The event level raw data structure that it deposits its energy or ADC counts into.
- The detector element type it is associated with.
- The AC++ command class used to configure it.

All these types must be defined as typedefs in the developer's digitizer class. A digitizer class has a special relationship with each of these types that it refers to. Some of these relationships have already been touched

upon in Section 2.2. Understanding the relationship between the digitizers and the classes it refers to is crucial to understanding how the system works. Here is a summary of the relationships:

- One digitizer instance is associated with precisely one instance of a subclass of *CdfDetectorElement*. For each digitizer type there is one associated subclass of *CdfDetectorElement*. This keeps the responsibilities of a single digitizer down to a minimum. It also allows for quick access of the correct digitizer for a detector element.
- Many digitizer instances are associated with one AC++ command class instance. It is assumed that many digitizer instances will share the same constants. For example, if the detailed calorimeter geometry describes a tower, and therefore has a digitizer type for each tower type, then all the towers for a given eta will likely share the same smearing constants.
- Many physical volume instances may be associated with one digitizer instance. Several of the sub-detectors will have detector elements that are composed of many physical volume “children”. The digitizer must know something about the physical volume structure associated with its *CdfDetectorElement*, because the stepping process will call on the digitizer for each particle step, in each physical volume known to the simulation engine, and the digitizer must respond appropriately.
- Many digitizer instances are associated with one raw data object instance. For example, if the detailed calorimeter geometry identifies each tower as a detector element and therefore each tower type has a unique digitizer, then a single instance of a raw data object could represent all the energy deposited in the entire central calorimeter. In this situation, the raw data object would span several tower types (digitizer types) and many instances of digitizers.

The developer of a digitizer is required to create a digitizer class that conforms to the digitizer rules. In addition to this class, the developer is required to create a class derived from *APPCommand* that will be used to configure the digitizer. The developer must identify the raw data object that will be used during the read-out procedure, this class is likely to already exist in the subdetector software as part of the reconstruction.

4.1.5 Configuration

The purpose of this subsystem is to present the user’s digitizer configuration to the simulation control. The user is required to supply a table that relates detector element names to digitizer class names. This is actually more complex than it sounds due to the fact that the relationships described in the digitizer section must be described in this file. Another complexity is that many digitizers can be active for the same detector element at the same time; this must also be described in the configuration file. The file has three white space separated fields: *detector_element_class_name*, *digitizer_class_name*, *set_name*. Digitizers are registered in a factory by class name, so that simulation control can create instances of digitizers given only the class name in the configuration file. The *set_name* defines a group of digitizers that all contribute to a single raw data object (as described in the digitizer section). The system will, at run time, insure that all the digitizers refer to the same raw data object type. All the digitizers in a set will contribute to a single instance of a raw data object that will be placed into the event. Here is a simple fictional configuration file:

<i>detector_element_class_name</i>	<i>digitizer_class_name</i>	<i>set_name</i>
CdfHalfLadder	G3PadovaDigi	SiliconSet1
CdfHalfLadder	G3PadovaDigi	SiliconSet2
CdfHalfLadder	G3NewMexicoDigi	SiliconSet3
CDF_CEM	G3CEMDigi	Cal
CDF_CHA	G3CHADigi	Cal

This example shows three important aspects of the system:

- It shows the ability to specify that two instances of one same digitizer class for the same detector element (presumably to run with different smearing constants) can be configured.
- It shows the ability to specify two different digitizers for the same detector element.
- It shows the ability to specify two completely different digitizers contributing to the same raw data object instance.

As mentioned in the Digitizer section, command objects are associated with group of digitizers. To determine what is the group of digitizer instances corresponding to a single AC++ command instance, we concatenate the *digitizer_class_name* together with the *set_name*. This insures that similar items are configured with the same constants by the same AC++ command object instance.

4.1.6 Raw Data Readout

From the *SimulationControl* level, raw data readout is straightforward – just poke the system at the right spot. Internally it is one of the more complex subsystems. As digitizers are created using the geometry and configuration file as a guide, they are assigned to readout groups by *set_name*, which corresponds to a single instance of a raw data object class to be produced. All the readout groups are recorded within the simulation control object. For any *set_name*, one can ask what instances of digitizers contribute to its raw data object. The readout group can be given an event and told to perform the readout. This procedure is defined in Sections 4.2.6 and 4.3.3.

4.1.7 Simulation Stepping

The design in this document reduces the search for the proper digitizer instance down to one data structure. This is a conscious tradeoff, using more space (memory) in order to gain greater speed. For each physical volume uniquely identified by the simulator, there is a corresponding entry in a lookup table that maps directly to a digitizer/detector element pair. When the digitizer is invoked it is given a detector element of the correct type; no casting is necessary. The design can easily be tweaked to include the CDF physical volume, so that the digitizer developer does not need to go to GEANT3 common blocks to read this information. The stepping procedure contains code that works with the generalized digitizer abstractions, which means it should never need to be changed when new digitizers are introduced or reconfigured. The stepping procedure relies on the data structures maintained by the digitization framework.

4.1.8 Digitization framework

All aspects of digitization are captured in the digitization framework. This includes initialization and configuration, raw readout control, and access within the stepping procedure. The framework maintains several data structures that allow various pieces of the reconstruction to efficiently navigate the digitizer instances. In our proposal, there are four maps maintained here. These allow the framework to efficiently perform the following tasks:

- Get a digitizer given a simulator specific physical volume identifier.
- Get a digitizer given a CDF geometry physical volume identifier.
- Get the readout group associated with a given set name.
- Get a command associated with a group of digitizers.

The framework owns the all the digitizer instances, command instances, and any required readout helper instances. Developers do not have to worry about resource management issues for any framework objects.

4.1.9 Factory

The factory allows for the creation of digitizer instances without undue link-time dependencies. Given a digitizer class name, the factory returns an instance in its generalized (abstract, interface, or base class) form. In this design, it is actually a wrapped, abstract form. The factory serves two important functions in

this design. It allows for name-only coupling of digitizers to the digitization framework libraries. This means that the framework support tools never actually know about or see the individual digitizer classes. The factory allows for the system to be configured using a text based table. This means that users can easily reconfigure and operate the digitizer framework with recompilation of the executable. The factory is used by the digitization framework during the initialization procedure.

The factory is the least defined part of the system. We propose a system similar to the one used in the calibration database system. This factory has additional requirements in that it must be able to create instances of objects associated with the digitizers, such as *APPCommands* and raw data generator management class instances. Extreme care must be used during the creation of this part of the system. It must be easy for the user to register a digitizer class with the factory. The registration of user classes must not introduce (or use) any templates or the KAI template mechanism can get confused (if standalone object files are used, as in the calibration database).

4.2 Description of Classes and Relationships

Accompanying this document is a series of class diagrams that show all the major classes required to create the framework outlined here.

4.2.1 MC Particles and Vertices

We do not present a proposed design for this subsystem. Instead, this design warrants review in its own right.

4.2.2 Simulation Control

- *SimulationControl*: This AC++ module must contain an instance of *SimDigitizers* (the digitization framework class). It must also utilize the instance of *IDPairList* produced by the geometry initialization module to initialize the digitization framework.
- **STEPDATA**: This is a concept definition. Many of the classes outlined in this document are templated on types that must be a model of **STEPDATA**. The purpose of this templating is to allow the framework to be reused with other simulators. In the GEANT3 case, the **STEPDATA** type could be simply a struct with pointers to common blocks in it. A digitizer will get passed a pointer to **STEPDATA** in the simulator stepping procedure. It is up to the stepping function to manage the **STEPDATA** instances and pass them to the digitizer framework.
- *IDPairList*: This is a list containing information for each CDF physical volume (PV) / CDF detector element (DE) registered in the simulator geometry. The list relates a simulator identifier to a PV/DE pair. This list is used to drive the generation of digitizer instances. It is also used to create the fast access map used in the simulator stepping function.

4.2.3 Geometry Initialization

- *GeometryManager*: This is the standard CDF geometry management module included with *FrameMods*.
- *SimInitManager*: This AC++ module produces the simulator geometry by walking through the CDF geometry tree. It should have the responsibility of creating the *IDPairList*. Passing the *IDPairList* to the *SimulationControl* module could present a problem since we do not want to store this information in the event.

4.2.4 Digitizers

The digitizer class must implement several methods that carry out the digitization process, and must define several typedefs. The concept **Digitizable** presents the requirements for all digitizer classes. **Digitizable** makes use of several other concepts, which we list below.

- **RAWDATA:** This concept defines what is required from all raw data types. Types that model **RAWDATA** must inherit from *StorableObject*. They must be able to append new “pieces” of data (e.g. calorimeter tower or silicon strip) which are accumulated during the digitization process to an existing instance. A type that is a model of **Digitizable** must present a typedef *raw_data_type*, which yields a type which is a model of **RAWDATA**.
- **DETECTORELEMENT:** This concept defines what is required from all detector element types. Types that model **DETECTORELEMENT** must inherit from *CdfDetectorElement*. A type that is a model of **Digitizable** must present a typedef *detector_element_type*, which yields a type which is a model of **DETECTORELEMENT**.
- **CONFDATA:** This concept defines what is required from all configuration data types (the classes which are used to configure digitizers). A type that is model of **CONFDATA** must inherit from *APPCommand*. A type that is a model of **Digitizable** must present a typedef *configuration_data_type* which yields a type that is a model of **CONFDATA**.
- **STEPDATA:** This concept defines what is required from all particle propagation information data types. A type that is a model of **STEPDATA** must provide all the information (specific to a particular physics simulation engine) concerning the result of a single particle step. A class that is a model of **STEPDATA** must also present a typedef *PhysicalVolumeID*, which yields the type of object (or primitive type) that is used by the physics simulation engine to identify an individual physical volume. A type that is a model of **Digitizable** must present a typedef *step_data_type* which yields a type that is a model of **STEPDATA**.
- **Digitizable:** This concept defines what is required from all developer-written digitizer types. A type that is a model of **Digitizable** must present all the typedefs listed above, which identify the types that the digitizer is designed to work with. A type that is a model of **Digitizable** must also provide the member functions shown for the type **Digitizable** in the accompanying class diagram.
 - *digitizeHit()* takes the specific detector element and the step data as arguments. This method may need to also take as an argument the physical volume and a flag indicating whether or not the simulator has entered, exited, or is stepping through the volume. Alternatively, the *digitizeHit()* method can be broken into three methods: one for entering a volume, one for stepping through a volume, and one for exiting a volume.
 - *appendToRawData()* is expected to place the energy or hit information accumulated so far into the raw data object passed in as an argument.
 - *clear()* should clear out any of the cached energy or hit information present in the digitizer.
 - *configure()* is meant to reset parameters for the digitizer. This method is likely to be unnecessary if the digitizer just uses the command that is associated with its instance to read out the parameters each time they are needed. If the parameters are to be read all the time, then the constructor will need to take an argument of *configuration_data_type*.

4.2.5 Configuration

- *ConfigFileReader:* This is a simple tool used to read in the contents of the digitizer configuration file. This class presents the contents of the file as a table or map of detector element names to a list of (digitizer name, set name) pairs. Each unique detector element name in the file can correspond to many digitizer/set name pairs. This data structure directly reflects this. Given a detector element name, this class returns the list of pairs associated with it.

4.2.6 Raw Data Readout

- *AbsGenerator*: An abstract base class that defines an object that can contribute to the generation of raw data. This class has no interface; it is only used to identifier classes that can fulfill raw data generation duties in their derived form.
- *Generator<RAWTYPE>*: This is a class derived from the *AbsGenerator* and templated on the type of raw data object that it contributes to. This class implements a method called *populate()* which is called when it is time to fill a raw data object.
- *AbsGeneratorElement*: This is an abstract base classes that represents a set or group of digitizers that all contribute to one raw data object. The method *addToEvent()* requests that the underlying set (implemented in the derived class) run through all the data generators that it holds and create a single raw data object that it will add to the event. The method *addGenerator()* requests that the underlying class add a generator to the list of data generators that it holds.
- *GeneratorElement<RAWTYPE>*: This is class derived from *AbsGeneratorElement* that actually holds the list of *AbsGenerator* instances. It is also templated on the raw data object. This class implements *addToEvent()* and *addGenerator()*. The method *addGenerator()* first converts the very basic *AbsGenerator* to the more specific *Generator<RAWTYPE>* form so that it can access the *populate()* method. The method *addToEvent()* creates an object of type *raw_data_type*, passes it to each of the *Generator<RAWTYPE>* objects it holds, and the adds the object to the event.

4.2.7 Digitization Framework

- *AbsSimElement*: An abstract base class used to indirectly describe any digitizer to the system. The digitizer framework holds on to the digitizer instances in this abstract form. The public interface of this class allows the system to digitize a hit and clear the digitizer. The interface also allows one to locate the *APPCommand* used to configure the digitizer.
- *SimElement<DIGITIZABLE>*: This is the central class in the digitizer framework. It is a wrapper around a developer's digitizer that allows the digitizer to plug into the system and be used correctly. This class is templated on *DIGITIZABLE*, which means that it has access to the typedefs in the digitizer class. This class has as data a pointer to the specific *APPCommand* used to configure it (Identified by the *configuration_data_type* typedef in the digitizer class. This class records the raw data generator set name that it belongs to. This class holds a pointer to the specific detector element that it is associated with (identified by the *detector_element_type* typedef in the digitizer class). This class is derived from *Generator<RAWTYPE>* and *AbsSimElement<STEPDATA>*. The derivation from *Generator<RAWTYPE>* allow it to be registered in the proper *AbsGeneratorElement* for readout. The derivation from *AbsSimElement<STEPDATA>* allows for generic access from the simulator stepping routine. This seemingly complex wrapper arrangement allows the system to hide how the system treats readout and digitizing hits from the user's digitizer classes. This class also removes the need for a dynamic cast in the stepping routine and the search for a corresponding detector element in the CDF geometry tree.
- *AbsSimElementList*: In the diagram is a class by this name. It is really just a list of pointers to *AbsSimElement* instances. This class exists to support running many digitizers for the same detector element simultaneously. For each detector element, there exists a list of digitizers that are active for that element. The configuration section discussed how the user specifies that several digitizers should be active for a given element.
- *SimDigitizers*: This class contains and controls access to all the major components in the digitizer framework. It permits high-level functions to be performed such "find the *AbsSimElementList* for this *PhysicalVolumeID*" or "generate all the raw data and put them into this event" and "clear all the digitizers" or "inform all the digitizers that we are done with the particle set for the current event". It holds following:

- A map of simulator-specific *PhysicalVolumeID* instances to *AbsSimElementList* instances. This is used in the simulator stepping routine to locate digitizer lists and invoke the hit digitization methods.
- A map from (*set_name+detector_element_class_name*) to *APPCommand*. This holds all the unique instances of *APPCommands* used to configure the digitizers. It is used during initialization for assignment of *APPCommands* to digitizers.
- A map from CDF physical volume identifier to *AbsSimElementList*. This is the main owner of all the underlying objects and lists. This map is used during initialization and cleanup.
- A map from *set_name* to *AbsGeneratorElement*. For each *set_name* there is a unique *AbsGeneratorElement* instance, this map holds all of these.
- An instance of the *ConfigFileReader* class.

4.2.8 Factory

- *SimMaker*: This class has three methods that are used to generate various class instances needed by the system. The method *makeDigitizer()* creates an instance of a digitizer and returns it wrapped in an *AbsSimElement<STEPDATA>* instance. The method *makeCommand()* creates and instance of *APPCommand* specific to the digitizer named in the argument. The method *makeGeneratorElement()* creates an instance of *AbsGeneratorElement* specific to the digitizer name given as an argument.

The factory mechanism used in the calibration database system, if reused here, poses a few problems. The calibration database factory relies on the fact that non-templated instances will be returned by the factory functions. This may not be the case here because of the templating of *AbsSimElement<STEPDATA>*. This can be fixed by introducing another empty abstract class from which *AbsSimElement<STEPDATA>* would be derived, and have the factory return instances of this class. The problem with this is that *SimDigitizers* would need to immediately *dynamic_cast* it back to the *AbsSimElement<STEPDATA>* that it is. Another alternative would be to do a slight redesign of the factory class registration mechanism. Further development of this topic is required.

4.3 Procedures

This section outlines a few important procedures that require further explanation, mostly in the form of pseudocode.

4.3.1 Digitizer Initialization

```

Locate IDPairList produced by the geometry initialization module.
For each unique DE in the list
  Locate list of (digitizer/set_name) pairs using ConfigFileRead
  Create an AbsSimElementList
  For each digitizer name in the list
    Concatenate digitizer name and set_name
    Use this name to look for an AppCommand in AppCommandMap
    If none found
      use the factory to generate an AppCommand
      put the command into the AppCommandMap
      locate AbsGeneratorElement in AbsGeneratorElementMap using
      the set_name
      if none found
        use the factory to generate an AbsGeneratorElement
        for this set_name, put it in AbsGeneratorElementMap
      use the factory to generate a new SimElement (digitizer),
      give the constructor the set_name, the AppCommand, and the
      detector element.
      Put the SimElement in the AbsSimElementList

```

```

    End for
  For each entry in the AbsSimElementList
    Dynamic_cast entry into an AbsGenerator
    Locate the AbsGeneratorElement in AbsGeneratorElementMap
    Using the set_name
    Add the AbsGenerator to the found AbsGeneratorElement
  End for
  Insert the AbsSimElementList into the AbsSimElementListMap by DE
  Insert the AbsSimElementList into the AbsSimElementListMap by PV
End for

```

When this procedure is complete, four data structure inside *SimDigitizers* will be populated: the map of simulator PVID to *AbsSimElementList*, the map of CDF geometry detector element to *AbsSimElementList*, the map of (*set_name* + *detector_element_class_name*) to *APPCommand*, and the map of *set_name* to *AbsGeneratorElement*.

4.3.2 Configuring Digitizers

Configuration can be done in two different ways. The first is to use the (*set_name* + *detector_element_class_name*) to locate the *APPCommand* used for configure a set of digitizers. The second is to use the CDF geometry detector element ID to locate a specific *SimElement*, then ask it for the *APPCommand*. It must be noted that the unique ID used to identify a CDF geometry detector element might need to be the detector element's physical volume ID.

It is envisioned that the *APPCommand* object associated with a digitizer will contain a simple group of data items that are set by the *APPCommand* handler.

4.3.3 Creation of Raw Data

Calling *SimDigitizers::generateRawData(AbsEvent)* will essentially do the following:

```

  For each entry in the AbsGeneratorElementMap
    Entry->addToEvent(AbsEvent)
  End for

```

The class diagram has notes that further outline the procedure for filling the event with raw data.

5 Questions Addressed - Part 1 (paraphrased)

5.1 CDF standard Generator Output Format?

During the review, everyone agreed that it is important to have standard format for generator output. It was also agreed that this format should be a *StorableObject* of the CDF EDM. In Section 3.5 we made a brief statement concerning the requirements for event generator output. This object would be both the mechanism through which simulated event information is presented to the detector simulation and the mechanism by which analysts gain access to the MC "truth" information.

5.2 How should new versions of GEANT be introduced?

The mechanism we propose introduces the physics simulation engine (currently GEANT3) to the simulation system in two ways:

- the simulation geometry;
- the particle "stepping" mechanism.

To minimize the effects from changing to a new simulation engine (GEANT3 to GEANT4, for example), the dependence on the physics engine must be encapsulated. Our proposal encapsulates these dependencies

in different ways. The creation of the simulation geometry is encapsulated in the existing CDF geometry system. We strongly recommend use of the “new” geometry system, and make a few recommendations for additions.

The particle stepping mechanism is encapsulated using “generic programming”, *i.e.* a template-based system. We introduce the concept of **STEPDATA**, which defines the behavior and typedefs the rest of the system expects from the physics simulation. **STEPDATA** encapsulates the information generated by the physics simulation. New physics simulations (or new versions of the simulation) would be handled by introducing new classes that are models of the **STEPDATA** concept.

In order to track minor version changes (for different releases of the GEANT3 product), it should be sufficient to tag the raw data objects with an *RPCID* that indicates that they were created by the simulation, and with what set of parameters — including the version of GEANT used.

5.3 How should output be organized?

Energy losses and radiation length integrated over trajectory? We make no comment on the physics content required, beyond stating that one of the required outputs is the “raw” data format of the subdetector. The exact definition of what is that “raw” data format is left somewhat to the discretion of the subdetector software leaders. In all cases, this “raw” data format should either be the *DBANKs* objects, or higher-level C++ objects from which other code can create the *DBANKs* objects.

5.4 Should hits be stored and how?

We have not addressed the issue of whether hits should be stored. If they are to be stored, it is necessary for them to be handled in a *StorableObject* of the EDM, and for this *StorableObject* to be designed in conjunction with the *MCEvent* class. The classes and tools designed by the subdetector groups should be reused for this purpose.

5.5 How should random number streams be handled?

This is an item largely independent of the simulation review. In Section 3.9, we proposed the development of a simple mechanism for the generation of random numbers. As we stated there, it should not be necessary to include all of ROOT merely to generate random numbers.

5.6 Interactive GEANT?

We have not addressed interactive use of GEANT in the simulation system.

5.7 What about MC truth information?

MC truth information should be handled in the *MCEvent* class and in related classes, such as *MCHit*.

5.8 What about objects that need to save themselves such as magnetic field?

This issue is very wide-ranging. Event data must be stored in the EDM. Other information could be stored in databases (perhaps using the calibration database API). Each case must be handled individually. Having objects save themselves is one implementation choice. Another choice would be to have such items as a magnetic field object contain an EDM object, which does know how to stream itself.

5.9 Is it possible to have one object output (list of primary interactions) instead of n-banks?

This should be deferred until the discussion of the Monte Carlo event class.

5.10 How should the particle DB be handled?

This should be deferred until the discussion of the Monte Carlo event class.

5.11 How does the input list of particles interact with the simulation output list of particles?

This should be deferred until the discussion of the Monte Carlo event class.

5.12 Is it possible to have a generic description of an MC particle and a description of the hits?

This should be deferred until the discussion of the Monte Carlo event class.

6 Questions Addressed – Part 2 (paraphrased)

6.1 How do we make the declaration of geometry and simulation as fast and efficient as possible while still keeping enough accounting information to allow for fast lookup in stepping?

Our proposal for modification of the *SimulationControl* class includes a mechanism specifically designed to optimize the efficiency of the stepping mechanism.

6.2 Can digitizers accessing the G3 specific common blocks be avoided?

Our proposal does not avoid the need to access GEANT3 specific information. We believe that it is best not to do so, because the digitization process seems likely to be intimately tied to the physics simulation engine — developers of digitization code are best served by allowing them access to the full complement of information provided by the physics simulation. Our proposal encapsulates the physics simulation in the **STEPDATA** concept, and makes use template-based programming.

One way to reduce the amount of common block access could be to pass the associated *CdfPhysicalVolume* instance in to the call to *digitizeHit()*. This information is then easily and quickly available.

6.3 Should CdfHitDigitizer be optional or obligatory? Is the CdfHitDigitizer completely unnecessary?

In place of an abstract digitizer base class (akin to *CdfHitDigitizer*), we have proposed a template-based solution. The closest relative of *CdfHitDigitizer* we propose is the **DIGITIZEABLE** concept, described earlier.

6.4 Can the hit digitizer part of the system be improved?

We believe that the template based solution we propose retains the strengths of the current design, while gaining still more flexibility. It also should make the work of those who will design and implement the classes for the digitization of each subdetector easier, since they need to design only two classes (the appropriate **DIGITIZEABLE** class, and the related **CONFDATA** class).

6.5 Can the dynamic cast in the digitizers from generic geometry element to specific element be eliminated?

This wish is one of the major forces that guided us toward the template-based design. In our proposed design, the *dynamic_cast* is only used during configuration of the system. The more time-critical “stepping” code does not need the *dynamic_cast*, yet retains the type safety required.

6.6 *How should change over from current simulation to a new system be managed? (Package and file collision problems)*

It is likely that the classes of our proposed design have few, if any, name collisions. It should be possible to build the entire structure below the *SimulationControl* class in both the current design and under our proposal simultaneous. *SimulationControl* could then be changed from the current implementation to our proposal when the underpinnings are ready.

The conditional compilation (`#ifdef`) system proposed by Chris Green will work, but care must be taken to assure that source code (*.cc*) and header (*.hh*) files do not contain a mixture of old and new code. One way to accomplish this is to rename the old code files with an identifying tag (such as OLD) in their names, and to include them in the file of the correct name if the proper preprocessor switches are set. These files can even be put into their own subdirectory.

7 Coding Recommendations

In the new design, there seem to be too many levels of abstraction and inheritance. Our proposal reduces this.

Since this document describes the design of a framework, we have no specific coding recommendations, except for the one in Section 3.3.2.

8 Physical Design

We would prefer that the sub-detector specific digitizers had name-only coupling with simulation control and the simulator.

9 Documentation

The current documentation is too ROOT-specific. It gives the impression that the only way to make the system go is by using ROOT.

10 Other Notes

The accompanying class diagram does not contain all the details required for a complete system. We suggest having a follow-up meeting to discuss the design, if the proposal is accepted.

It has been mentioned that it is important to be able to reuse code from the current simulation system and that a migration strategy would be preferred. We do not believe that the system here precludes reusing some of the already-developed digitizers. To do this could require a wrapper or adaptor that will be placed over the current digitizers to give them the look-and-feel of the new system and meet the system requirements. This will need further discussion.