# MINOS Software Review

**Jim Kowalkowski, Marc Paterno**

## 1 Introduction

This document contains a summary of our findings from the MINOS software review meeting of 25 September, 2000. This is not a detailed review; there was far too much material presented during the review meeting for us to be able to adequately respond to it all. Past experience with software reviews shows that it takes about three weeks to complete a review for a single subsystem or package; this review covered many such subsystems. We see a clear need for additional review, and suggest that several of the subsystems be the focus of a series of smaller, detailed reviews.

We also suggest that MINOS consider a more structured external review of the scope of the computing project. Such a review may help MINOS define priorities for the many tasks discussed in this review.

The structure of this document is as follows. In Sections 2 through 5, we present general comments regarding the software design process and the software design itself. In Section 6, we present comments specific to each presentation. These frequently contain references to the relevant sections earlier in the document. We have not commented upon a few of the presentation. This is not because they are unworthy of comment, but merely reflect our domain of expertise.

## 2 Software Decision Process and Resolving Issues

In this section, we address the most important issues we perceived regarding the software process in MINOS, including issues of scheduling and decision-making.

### 2.1 Software Process

We are concerned that there seems to be no clearly defined process for making general decisions. For example, during one of the presentations a comment was made about preferring the use of the Standard C++ collection templates (colloquially, the “STL”) to ROOT collection classes; this comment generated a somewhat heated discussion of this basic issue. We believe that it is critical for the experiment to have a forum in which such issues are discussed and resolved, and a mechanism for assuring that the software designers then abide by these decisions.

We are also concerned by the lack of a shared “software process”. We have seen other collaborations adopt the use of highly structured processes (requiring large amounts of formal documentation of requirements, detailed up-front design to be followed by methodical implementation, *etc.*). While we do *not* recommend the use of an overbearingly formal process (our experience has been that such processes are not well suited for the more “flexible” HEP community), we do think that some process needs to be defined.

Specifically, we recommend the use of the UML for documenting software designs, at a level of detail suitable for each task -- neither so little as to be useless in supporting design, development, and maintenance, nor too much, to be a burden beyond the benefits possible to gain.

We also suggest the development of a set of milestones useful for both the managers of the entire MINOS software, the managers of various software subsystems, and the developers producing those systems. An excellent reference we recommend is **Extreme Programming Explained**, by Kent Beck. The XP (extreme programming) methodology is particularly light-weight, and adaptable enough to allow users to choose the part they wish to use, without subscribing to the entire system. Of most importance in this context is what is called the “planning game”, which describes an excellent method for obtaining -- and maintaining -- a useful and *achievable* set of milestones.

## 2.2 Schedule

We note that the schedule for MINOS software development has two natural timescales: summer 2001, and 2003. We believe it is important to keep these two timescales in mind while planning milestones.

There is little time before the summer milestones. We suggest that the critical components required for the summer run be identified, and that short-term solutions (to be improved upon or disposed of later) be acceptable for meeting these milestones. The “planning game” mentioned in Section 2.1 may be of its greatest value here; without a list of milestones useful both to developers and managers, there seems to be little chance of success.

For the longer time scale, we recommend that MINOS take advantage of the experience gained during the completion of the summer milestones to determine the requirements of the long-term system. It should be expected that most, if not all, of the software developed during the initial phase will be replaced by the later systems. Because of the long expected lifetime of MINOS, greater emphasis should be place on the *maintainability* of the software for the long term. It is here that the benefits of a well-designed OO system will accrue.

Finally, we note that the GANTT chart presenting the schedule and deadlines for the MINOS software project seems to be of little use, because of two admittedly hard-to-fix failings: (1) many of the tasks are not concrete enough, making it impossible to determine if they have been fulfilled, and making it difficult to determine what tasks rely on previous tasks, and (2) the times associated with those tasks are largely fictional. We suggest that the procedures described in the "planning game" (see Section 2.1) be used to develop a useful time line.

# 3 Analysis

## 3.1 Requirements

Producing concrete requirements is essential to produce software that users need and also to have a measure of success. The goals of the software must be specified by the experiment as a whole. Guidelines and policies must be defined. From the presentation, we got the impression that this was on the list of things to do, but was not nearly complete yet. We also felt as though each subsystem had this as a task. Some of the requirement and goals for the software can be stated for the experiment as a whole and should be. Below is a simple example list of requirements that may be able to be answered by the experiment and would give direction and consistency to the software.

- What are the differences between reconstruction and analysis environments?
- How are the packages tested and validated?
- What are the interactivity requirement for analysis session?
- What is the degree which end-users are required to know C++ (and ROOT)?

A few example guidelines could include:

- OO Design Guidelines
- Coding standards
- Package organization standards
- Package documentation standards
- Use of consistent terminology (definitions)

In our experience, complexity of C++-based subsystems and packages is a big problem. Packages that have many rules, unintuitive behavior, or inconsistencies with other packages tend not to be used at all (bypassed whenever possible) and become the target of much criticism. Disregarding "best practice" as documented by experts in design such as Stroustrup, Sutter, Meyers, and Lakos can lead to this problem. This is especially relevant in regards to novice C++ programmers.

We highly recommend that testing be included as part of the development process. This includes automated testing integrated into the build system, where each package contains a series of component and system tests that get executed automatically by the build. Testing is critical for long term maintenance of the software and for catching errors immediately when lower-level packages change. Testing, if done properly, insures that the software is operating as it is supposed to operate.

Guidelines for Object-Oriented design will help produce uniformity amongst the various subsystems. The guidelines can be a short list with references to back them up. One example of such a reference would be Riel's **Object-Oriented Design Heuristics**. Having these guidelines could help prevent developers from making bad design choices. They can help with issues such as inheritance vs. containment and creating a well designed object. The guidelines will help developers produce code that is usable and maintainable.

### 3.2 Standards

The experiment as a whole should decide on what standard tools will be used. The decision here will dictate what users will need to learn about in some areas. It will also define a direction for the experiment and a plan for expansion in the future. *We would strongly recommend the use of industry standards whenever possible.* We heard reference to some of the technologies below, but saw no rationale or specification document that could be used by all subsystems to give uniformity to the software as a whole. The document should include choices for the following areas, including brief description of why the technology was chosen and how it will benefit the experiment know and in the future.

- Networking / Distributed Programming tools (CORBA, Raw Sockets, ROOT)
- Histogramming tools (AIDA, JAS, ROOT, various commercial tools)
- Graphics (HEPVis, JAS, Java, Python, ROOT, Wired)
- Data exchange formats (HDF, ROOT, XDR, XML)
- Databases and APIs (MySQL, ODBC, Oracle, TSQL)
- Scripting languages (CINT, Java, Python, various commercial tools)

The experiment should be cautious about adopting technologies that have been placed into ROOT, such as rootd. Are facilities such as this robust enough and will they be viable for the life of the experiment? Other products like NFS and AFS have been around for many years, have a huge user base and a development team that specializes in this type of technology. This is just one specific example; industry-wide standards should be taken advantage of.

## 4 OO Design Issues

This section only points out a few general areas of concern. There should be a clear understanding of why OO was chosen and what the benefits are. What is the purpose of OO design? Why did MINOS choose this methodology? Here are a few reasons for making the OO decision:

- Maintainability
- Flexibility
- Extensibility

- Encapsulation: Keeping concepts clear and makes the system easier to use and modular.

## 4.1 Benefits of OO design

The real benefits from OO design appear only when one makes adequate use of abstraction. This entails programming to the interface of a class (its public member functions) and not its implementation (data members). A class which contains little more than set/get methods for each data member is no better than a struct (no encapsulation).

A big problem area we have run across is classes that span too many concepts or features. A well designed class will have a single purpose and encapsulate a single concept. We see this problem in many of the class diagrams that were shown in the presentations.

We heard the term “self-describing data” mentioned in several talks. We felt that in the context of these talks, that the ROOT tree is has self-describing data and that there was a distinct advantage in having this feature. Object-Oriented programming means that the data is private (encapsulated by the class and an implementation detail) and that access is only available through the public interface (methods). An object included in a ROOT file is not useful without the C++ code for the object. The only proper way to interpret the data held within the object is through the object’s interface, which requires the C++ code for the object. If *structs* are stored in the ROOT file, then browsing the data makes sense, because that is what a struct is. Browsing a ROOT file that contains *class instances* does not make sense without the code for the class. The ROOT files do not store the code for the class in any form. A real self-describing data file will likely include more information that ROOT included. A format such as HDF is self-describing.

We believe that self-describing data should be investigated as a low-level persistency concept, but that the form of the data presented to C++ reconstruction and analysis code should be full-featured C++ classes, not merely structs with get and set methods.

## 4.2 OO implies design

To gain the benefits mentioned above, an OO system requires *design*, rather than immediate production of code. Typical scripting languages or macro languages do not lend themselves to such design. Maintainable code is not produced in an *ad hoc* fashion.

It is of great benefit to take advantage of design reviews *before* the production of a significant amount of code (more than a prototype) for a system. Having a solid design leads to far less time coding, and debugging, and is more likely to lead to a product that actually meets the requirements for the system. Early design effort is more likely to lead to correct software, and software which is more easily extensible and maintainable.

# 5 C++ Language Issues

We strongly urge that MINOS use Standard C++, and not a partial dialect nor vendor-specific “enhancements”. In this section, we discuss several of the reasons for this for this recommendation, and the ramifications of accepting or ignoring this recommendation.

## 5.1 Forbidden language features

There were several presentations that discussed forbidding (or discouraging) use of some features of Standard C++. We discuss each of these major features in turn.

### 5.1.1 Exceptions

The use of exceptions is a part of Standard C++. A decision *not* to use exceptions means a decision to make use of some vendor-specific extension to the Standard, and is inherently an obstacle to portability.

Furthermore, exceptions have been designed into Standard C++ as part of the method of producing robust systems. For example, it is not possible for a constructor to return a status indicating that it has failed. Standard C++ would use the throwing of an exception to indicate such a failure. If non-Standard extensions were used, preventing the throwing of such an exception, then the object would have to be returned in an invalid state. This in turn leads to a design in which one must be aware that any object may be in an invalid state, and so to produce robust code, one must insert error checking in many places. Thus the “performance penalty” associated with the use of exception handling may be less painful than the performance penalty associated with robust code in the *absence* of exception handling.

The design and coding guidelines adopted by MINOS should make a clear statement of the experiment’s policy with regard to exception handling. The design of each subsystem should also include a description of what exceptions or error conditions can be produced, and how they are handled.

### 5.1.2 RTTI

More than one of the presentations discussed not using C++ Run Time Type Identification (RTTI), because ROOT provided such a facility (unfortunately also called RTTI). As is exception handling, RTTI is an inherent part of Standard C++; it can only be “turned off” with a non-Standard, vendor-specific, and inherently non-portable language extension.

A basic feature of C++, the dynamic_cast mechanism, can not work without RTTI; all implementations we have seen that allow the “turning off” of RTTI also disallow use of dynamic_cast when RTTI is turned off. The dynamic_cast mechanism is the only method by which type-safe downcasts (casting a pointer-to-base-class to a pointer-to-derived-class) can be performed in C++. Losing this ability is a major blow, and can have devastating impact on the reliability of a C++ software system.

The RTTI provided by ROOT is of a different nature. It is a part of the data dictionary, and allows the answering of questions like “is class *A* related to class *B****?”.*** This is not, however, a replacement for the ability of dynamic_cast, and as far as we know, dynamic_cast is the only mechanism for safe downcasting, especially in the presence of multiple inheritance (used heavily in many systems, for example ROOT).

### 5.1.3 Templates and the Standard Library

Unlike a decision to use non-Standard extensions such as disabling exceptions or RTTI, a decision not to use templates or not to use the Standard Library does not mean writing non-Standard C++. However, such a decision means that MINOS will not be taking advantage of one of the major advantages provided by C++.

First, we note that a decision not to use templates must also mean a decision not to use the Standard Library, because almost all of the Standard Library is template based. The C++ *string, iostream* (*cout* and *cin*), collections (*vector, list, map...*), iterators, and algorithms are all template based. It also, of course, means prohibiting the use of MINOS-defined class and function templates, and removing one of the major tools (generic programming) provided by Standard C++.

A frequently overlooked but very important part of the Standard Library is the algorithms presented in the header <algorithm>. These algorithms provide methods for looping, filtering, selecting, merging, and sorting (among others). These algorithms have been produced by professionals for whom this is a specialty, and are unlikely to be improved upon by non-experts. Prohibiting use of templates also means prohibiting the use of these function templates -- and so these things would have to be reproduced by MINOS programmers. Experience shows that this will be the case, resulting is a waste of manpower for the production of an inferior product.

We heard two major objections to the use of templates, and address each of them in the following subsections.

#### 5.1.3.1 Poor compiler support for templates

Several years ago, the state of compilers was generally poor, and few had adequate support for templates. Today, the situation is much better. There are several multi-platform compilers available with adequate template support, and many of the vendor-supplied compilers are adequate. The situation is also rapidly improving. We do not believe that lack of compiler support (from current compilers) is a valid reason to avoid the use of templates.

#### 5.1.3.2 Poor support of templates by ROOT and CINT

This becomes relevant in two different domains: (1) support of templates for ROOT I/O, and (2) support of template classes at the CINT prompt.

We would argue (as use at CDF has shown) that it is possible to use template classes in design, and still make use of ROOT I/O with those classes. While this would be simpler if ROOT I/O were to better understand templates, the amount of work necessary to use ROOT I/O with custom templates has been less onerous than would have been the lack of templates entirely. Furthermore, several of the class templates of the Standard Library (the collection templates) are already understood by ROOT I/O.

It is currently not possible to make reasonable use of class templates from the CINT prompt. See Section 4.2 for our discussion of OO design principles, and Section 5.2 for our statements in regard to CINT versus C++. We do not think that the lack of this ability is of sufficient importance to consider making the use of class templates outlawed by MINOS.

## 5.2 C++ is a multi-paradigm language

C++ allows programming in more than one style. It allows *procedural* programming, as a “better C”, it supports *object-oriented* programming, and it supports *generic* (template-based) programming. It also allows mixing of these paradigms.

Not all C++ code (even when using “classes”) is object-oriented. Code that manipulates basic data types, or structs of such types, does not gain the benefits described in Section 4. Many of the talks presented design ideas that, while executed in C++, were not object oriented, because they did not exhibit the properties discussed in Section 4.

We strongly believe that interactive use of C++ will not lead to *any* design, much less a solid OO design. Code produced as “macros” is generally of poor quality in terms of clarity and maintainability. The perceived advantage of the CINT prompt is speed of development. We believe this can be achieved using a modern development environment that allows compiled code to be integrated dynamically into the running system.

## 5.3 Standard types

Several of the talks presented designs using the types *Int_t* and *Bool_t*. It is unclear why these types are used. *Int_t* (presumably defined by ROOT, as a signed 4-byte integer) is useful when it is important to explicitly fix the size of an integer -- but this should be necessary only under special circumstances. Such cases of use should generally be hidden by the public interface of a class. Widespread use may lead to inefficiency. *Bool_t* is an ancient artifact, left over in ROOT from pre-draft-Standard days. The built-in type *bool* has been a part of Standard C++ for many years, and should be used directly.

## 5.4 Memory management

Our experience with other reviews has taught us that the issues of memory management are many times difficult for new C++ programmers, especially programmers coming from the Fortran world. We strongly recommend a coding policy that makes ownership of objects obvious at all times. Some examples of techniques that can help are:

- pass-by-const-reference rather than passing pointers;
- use of the Standard Library class template *auto_ptr*.

Furthermore, MINOS should invest in licenses for leak checking tools such as purify or insure++. These tools have proven to be very valuable at D0 and CDF in solving difficult problems quickly.

## 5.5 Iterators

The Standard Library provides a model for iterators that is clear and concise. It is the result of many years of work from people that study solving this problem correctly. The iterators of the standard library are extremely efficient in almost all circumstances in which they are used. This also a good solution to separating containers and contained objects and allowing algorithm to operate on ranges of contained objects.

Care should be taken when developing iterators that differ from the standard ones. The interfaces should not model the standard ones unless they are extensions of the standard ones. The standard iterators always exhibit uniform behavior across the collection types and guarantee a certain level of performance. Many experiments create there own iterators that have similar interfaces to Standard Library iterators. In all these cases, the custom iterators do not yield the necessary performance or have behavioral quirks when used in certain circumstances. All the custom iterator packages that we have seen will have maintenance problems. Custom iterators should have an interface of their own and be distinguished from the standard library iterators.

We strongly recommend that the Standard Library iterators and iteration techniques be used whenever possible. We further recommend that developing custom iterators should be avoided, unless they provide significant enhanced functionality.

# 6 Presentation Comments

This section presents a series of comments targets at each of the talks. The comments are presented as bullet items for brevity.

## 6.1 Offline Analysis Requirements

- Concrete numbers for speed of reconstruction are needed. A concrete time budget will allow asking the question “are we fast enough”. This is, of course, as important number; it could have a direct impact on C++ design decisions.
- Concrete numbers for speed of simulation are also needed.
- Care should be taken when selecting network service daemons (see Section 3.2).

## 6.2 MINOS Fortran Software

- The new system should not be limited or constrained by what the existing Fortran software does.
- The old system should be used to help define requirements. Functionality and features that were good should be repeated and enhanced. Features that were lacking or poorly implemented should be provided by the new system.

## 6.3 Data Model

- What software needs to run on Windows NT/Windows 2000? Is the software that runs on Windows separable from the offline software? Is SRT support for Windows required? Does any of the offline software need to be compiled under Windows, including the “data objects”? If so, is gcc coupled with Cygwin adequate? If purchased software runs under Windows, does it need to be extended (code added to it that uses its libraries)?
- It was mentioned that data collection must continue even if the database is down or inaccessible. What does “Database is Inaccessible” mean? Is this calibration data needed to start a run? Databases such as Oracle ODBMS (for example) are very reliable when managed properly. The chances are that the code written by MINOS will not be more reliable and robust than the Oracle Database software. The integrity of long distance network links is another problem and the data collection must be protected against this. Freeware databases or freeware APIs to databases are also another issue.
- Why is there a requirement that detector status be available without referencing the database? We see no reason for monitoring detector status without the database. The talk did not give an adequate justification for this statement. Our opinion here could be a result of our vague understanding of the problem.
- The talk referred to objects owning their data and granting access. Does “Grant access” mean reading only or is it writing also? Under what conditions will

read access be denied. If grant is for writing also, how would this be achieved? This item needs to be more specific.

- Sounds like completely primitive, simplistic data objects. Asking the data what detector it is associated with typically means data that is arrays of floats or ints and that the object is covering several concepts. Asking for the type of data could mean arrays of bytes that are really arrays of floats or ints. We are concerned that this is a completely non-OO design, and subject to maintenance problems and accidental improper use.
- The actual data model needs a review all by itself. We strongly encourage a review of the data model and its interactions with other parts of the system.

## 6.4 OO Talk

- Simplicity of the detector - does this really make the software infrastructure any easier then other bigger experiments? We do not believe this makes the infrastructure any easier. The requirements for the bigger experiments are similar and the number and type of software subsystems is similar.
- The software seems to be very ROOT centric. Is this due to the perception that ROOT has most of everything done for you already and you just need to add a few bits and pieces? We suspect much more is needed.
- Minfast: This is not an OO project. This is a procedural project involving direct manipulation of structures and data (see Section 4). OO experience will not be gained by this project, only simple C++ syntax and procedural programming experience.
- Why have ROOT containers been chosen over Standard C++ (see Section 5.1.3)?
- The statement “Flexible data format with abstract interfaces” was made in III.2.B. Abstract interfaces do not lend themselves to browsing expect through the interface itself. Again, it appears that your objects are not objects, but structs. Creating an abstraction comes at a price, one such price being performance. Having many abstractions is proving to be complex for uses to follow and understand at CDF and D0. Many times the intention is to have many specializations for the abstraction, but only one concrete implementation appears. They is overhead and complications in storing and using objects in there abstract form. Returning objecting in their abstract form and expecting users to convert them back to their actual form is poor practice. Can objects in their abstract form be mixed that should never be mixed (a parameter set for jet finder paired with a tracking algorithm because a algorithm handle is an abstract parameter set and an abstract algorithm pair)?
- Lack of manpower (III.C.1) and turnover means that it is critical to have a good OO design - this is the strength of OO (see Section 4).

### 6.4.1 Database Issues

- (III.B.4) MINOS needs to understand in detail the requirement for analysis at all institutions. This does not appear to be a short term requirement. Sufficient care should be given to this problem to make sure the solution will work and be maintainable in the long term. For example, we immediate wanted to ask the following questions. How should data be delivered? By FTP? Tapes by mail? DVD? Direct access of files over AFS? Should it be abstracted so it can evolve?
- It was mentioned that Minimarts (MySQL) will be used for information such as calibration, alignment and run conditions. Does this include the event catalog? Is MySQL adequate? MINOS should specify requirements for the database, such as concurrency, rollback, backup, etc. instead of choosing a database technology first.
- Need to define remote/local database requirement clearly. Such requirements could be that updates to MySQL remotely are allowed, but data must be submitted to central system. Are remote institution requirements different then local ones? Will the database technology be different local and remote?
- What are coding standard that apply to database standards (see Section 3.2)?
- CD, D0 and CD have valuable experience in the area of databases; both have been working on the distribution and access problems for quite some time.

### 6.4.2 ROOT

- (IV.B.1) CINT macros are a drawback when it comes to coding in C++, not a benefit (see Section 5.2). Again, a good development environment could reduce the need for macros.
- (IV.B.2.a.1.c) The self-describing data format of ROOT is mentioned here. How is this feature important to MINOS? If the idea is that the data can be interpreted without code, then this is incorrect (see Section 4.1). If an object is stored, then the code for the object is required to interpret the data.
- (IV.B.2.a.1.a-b) How is sequential or random access or full or partial access to events managed or handled? Is ROOT going to give you this ability for free? If so, how?
- (IV.B.2.e.1) Dynamic loading is part of the OS and fairly easy to take advantage of. Versioning of the libraries immediately becomes important in order to record the relationship between the data that was produced and the code that produced it. CDF and D0 are exploring this problem right now.
- (IV.B.2.b.4) Abstract interface to graphics is not always best solution. Many times you get least common denominator of all packages or quirky, limited behavior on certain platforms. This is a tough problem and generally the classes are not at all easy to use. Again, D0 and CDF have been working on event displays using many different technologies, including ROOT, MINOS should talk with them.

- (IV.B.2.e.2) RTTI of root being used instead of the one built into C++ (see Section 5.1.2).
- (IV.B.2.d) How robust and modern are the networking tools and solutions of ROOT compared with the standards from industry? See Section 3.2.

### 6.4.3 Language Features

- (IV.C.1.b) Poor old compiler support of templates is not a good argument for not using them, especially since the old compiler is not used (see Section 5.1.3).
- (IV.C.1.a) Use of an important language features should not be limited because of CINT deficiencies. Not all code or class needs to be available is its raw form at the CINT prompt.
- (IV.C.2.a) ROOT directly supports vector and list, can they be used instead of root container classes? In what circumstances do they not work properly?
- (IV.C.3) A decision should be made about a standard or main iterator model. Having a secondary ones is fine, as long as they do not pretend to be STL or pretend to be STL compliant (see Section 5.5).
- (IV.C.4) CPP macros are a poor substitute for templates. They are usually unmaintainable, awkward to use, and produce difficult to follow error messages from the compiler. They also do not work well with language features such as typedefs.
- (IV.C.5) Asserts go away when the optimizer is turned on. They are only to be used for finding problems like logic errors - not runtime data validation. It is a mistake to ignore exception handling and error checking, especially now at this early stage. These rules should be spelled out now for error handing.
- There are many products from the HEP communities and commercial sector for GUIs and data manipulation. A disadvantage of using ROOT classes for everything, including RTTI, is that incorporating these third party products can become very difficult. With the long lifetime of MINOS and the current rate at which products are evolving, we feel it is a mistake to disregard them.

### 6.4.4 Philosophy and Goals

- (V.B) Rules should be established for the organization of SRT packages. This must include directory structures, testing, and documentation procedures.
- We recommend that MINOS use the Fermilab SoftRelTools.
- CDF has substantial experience (Art Kremer of the PAT group) in distributing code and releases using CVS and SRT, make use of this experience. We do not recommend CDF’s nightly build strategy, but prefer the test release build procedures from D0.
- Clarity of the two time scales is good. Focus should be on what needs to be done, not how it will be done.

- Long term goals: Mentioned assess, rewrite, redesign. Should also be reassessing the requirements, and should not be surprised if large portions need to be thrown away.
- General requirements: Should have been made already; even in a short, simple form. Now MINOS needs something for real; not just a simple feature sheet (see Section 2.1)

## 6.5 Framework

The talks in this group seemed to be overlapping; they also seemed to be addressing similar things in different ways. Trying to figure out their relationship and how they work together was not clear. The framework itself and JobOptions are in need of a separate detailed review. The Candidate package appears to be partially in the event model area.

### 6.5.1 Candidate and Algorithm Packages

- There appears to be an undesirable dependency graph, where data objects depend on algorithm objects. Data objects depending on algorithms is unlike any other use that we see. MINOS must discern between quickly changing code (algorithms) and slowly changing code (a muon object). The physical dependency tree must reflect this. Never make the slowly changing code depend on the rapidly changing code. An example would be that one cannot make a transverse momentum distribution of muons from already reconstructed data without including all the muon reconstruction code. If there are many algorithm types, then all the various algorithms need to be pulled in also.
- Universal Object IDs are a well-known technology, examples of places where they appear are in CORBA, COM-DCOM, and DCE. This is difficult business and you should consult books on the subject before jumping in and decide if it is really necessary. Typically they IDs are large and associated with objects that do a large function, such as manipulation of a spread sheet. There is usually assignment and scaling problems associated with these IDs.
- What purpose does the abstract algorithm serve? Why does it exist?
- The design looks very complex, which is not going to be good for users, especially ones that are not good with C++. Experience from CDF/D0 shows that users will not want to learn complex structures and will try to bypass them.
- Implementing this design without templates will pose a maintenance problem (see Section 5.1.3).
- (II.A.2) The use of Set/Get methods in a class to manipulate data members implies no encapsulation, just composition. (see Section 4).
- (II.A.3) You must carefully address the problem of labeling data objects when more than one instance can exist in the event, produced by more than one algorithm. Be very careful when you think there will be just one instance of a particular data object, it is likely to be an incorrect assumption.

- (II.A.4) Data object history (version of code, package, library version, parameter set, derivation from other objects in the event) is critical for reproducing results and must be managed properly. Quality control measures should be put in place (code inspection) at the data object level to make sure that history is populated correctly.
- The event will need to record “world state” information, such as alignment/calibration set identifiers.
- The concept of separation of persistent and transient store is missing. Reconstruction and analysis modules should communicate with a transient store. At one place in the framework, the decision of what goes to persistent store is made and how it gets there is know.
- (II.7) There is a requirement that one algorithm cannot modify an object without disturbing other algorithms that are running. Copying the object is the only way that we know of. There needs to be a strong statement and policy made about modifying objects in the event - with reproducibility as a key concept.
- (III.B, III.E) A “clone” method is needed to copy objects that are manipulated in an abstract form. Is cloning/copying deep or shallow? This is a complicated question. Maintaining associations becomes a serious problem. Do the associated objects get copied (implied by a deep copy)? Do the associated not get copied, which means that you cannot navigate to them and modify them?
- Reverse associations are very difficult to manage correctly and should be avoided at all costs. They should be handled externally from the objects themselves in “associative collections” (similar in concept to the CLEO Lattice package). If the Lattice package manages connections between objects (relationships), why would pointers from daughter to parent be needed at all?
- (II.9) The generic concepts of “overlap” and “equality” is not valuable. They terms only have meaning for specific types matched against the same types.
- What is the lifetime of Candidate objects? If the event is the lifetime, then this is easy, because we know when the object need to be cleaned up. If it is not the event, how is lifetime managed and what does it mean? The reason for an event (frame) object is to facilitate this. Having these objects float around system is bad.
- The talk makes a good statement on objects being created in proper state. All objects should be constructed like this.
- Parameter setting in algorithms should never have defaults coded directly into the algorithm. These parameters will be difficult to trace.
- Reference counting with not-obvious or strange rules (copying when ref count > 1) will cause problems and confusion for users (issue of managing complexity). Usually rules like this are needed because of design problems. If performance is an issue, then we should remember the point from an earlier talk about “performance is not an issue now”.

- This is a package that needs to be reviewed at the same time as the JobOptions / Module package.

### 6.5.2 JobOptions and Modules

- Methods without arguments or return type usually indicate that data needed to operate comes from global areas. This is generally a bad design choice and should be looked at in the review of this subsystem.
- Is driving the framework event loop from the ROOT prompt a requirement? This may put significant constraints on the framework design.
- Other experiments at Fermilab have ways to configure their frameworks and modules. Why are they not adequate?
- Appears that Module class is a “super” object, with many concepts within one class. This is bad practice and leads to inflexible code and makes it more difficult to use.
- The interface from the ROOT prompt shown in the sample session is very unintuitive. We have seen several frameworks; it is not immediately obvious how this one works and how to manipulate it.

## 6.6 MINOS Persistency

- One of the things this project was based on was ATLFast++. This project (ATLFast++) is being converted to the Athena - a framework based on Gaudi, which is similar to the event store concepts at CDF, D0, and BaBar. This type of event store allows for multiple persistent back-ends.
- What about schema evolution and automatically generated streamers? Does ROOT directly support what is needed by MINOS? Will MINOS need to write the streamers by hand to perform of schema evolution, as does CDF? Has MINOS demonstrated how schema evolution will work in regards to ROOT and the MINOS data objects?
- Self-describing object format of ROOT may not be what you want, see Section 4.1.
- Simple pointers in objects may not work as expected in ROOT, especially in regards to split file mode.
- Are users to be aware of all the branches and streams? How do users navigate though the TTree? We recommend a event model that provides tools to do this navigation and hind the TTree and directly manipulation from the users.
- ROOT does not claim to be an object database. Distributing data over multiple files is difficult and carries several restriction that should be understood. The ROOT web site and experts in this area should be consulted to understand the limitations and restrictions.

- Package objectives should be completed as soon as possible (requirements). The problem should be understood first, then figure out what technology will satisfy the requirements.
- This package and plan needs a review of its own.

## 6.7 Geometry

- A single source for all geometry was mentioned in this talk. This single source would be used for Geant3/Geant4 geometry initialization, and for reconstruction. This is a very good idea. How will this be realized? CDF is doing this and should be consulted on the matter. We have heard that ROOT also has an abstract interface to Geant3/Geant4 that could also be explored. Thought should be given to using the Geant4 geometry as this single source.
- We would like to look over the design for the magnetic field management classes. This seems like it could be a bottleneck for reconstruction and an area of performance problems.

## 6.8 Event Display

- The MVC architecture is good, it breaks down the problem properly.
- Separation of the event display GUI and framework event loop controls is a good concept and should be explored further.
- The idea of being able to use tools other then the ROOT GUI is good. MINOS will be around for a long while, it is good to be able to adapt to new technologies.
- C++ based GUIs are known not to be the easiest tools to use; many scripting languages have much easiest to use GUI toolkits.
- MVC facilitates proxies to remote event display servers. The idea of separating graphics manipulation into separate processes is good.
- Development of this system will be a lot of work. Be sure to have a clear list of objectives and goals (features and requirements) before you start to ensure that a product will be ready (see Section 2.1).

## 6.9 Navigators

- We strongly discourage the use of CPP macros in place of C++ templates (see Section 6.4.3).
- What is the overhead associated with using this package - in terms of memory and CPU? Is incrementing an iterator a virtual function call? This could severely impact performance in a negative way.
- The iterator and container concepts here are very different than the Standard Library. (see Section 5.5).

- The NavKey class contains overlapping concepts and inappropriate use of inheritance. These features will cause the class to be used incorrectly and will be the source of maintenance troubles.
- It is unclear what the advantage is of this package. It appears to be complex to use - a feature that users to not like at all when it comes to C++. There may be other, easier ways to manage this problem using combinations of Standard Library algorithms and containers.
- Is this a substitute for a data model? It appears that the CLEO Lattice is tool used in conjunction with a data model, used to relate objects (bidirectional if necessary). It is unclear that MINOS has the same goal. For MINOS, it appears to be a tool used to temporarily relate objects for analysis use in a single job. The use of C++ templates make it clear how CLEO Lattice is to be used - associations between objects are expressed through templates. This is a sophisticated and powerful concept.

## 6.10 Database API

- MINOS should be talking with D0/CDF - they are doing similar things with relational databases. MINOS should be using CD database resources to help design tables.
- Hiding database access through interfaces in C++ is a very good thing (database independence - no database structure visible in C++ - users should not need to learn or see SQL). We recommend consideration of the ODBC standard.
- Oracle replication is being tested and used at CDF and D0. MINOS should consult with them - they have important experience with this feature.
- Who will design the tables that contain the data? Who will design the management tables? It would be good to see database table layouts soon, to verify that the system will manage data properly. We would like to see the mapping between data administration tables and C++ objects.
- How are new database tables introduced and schema evolution managed? Will it be through SRT releases, or by a separate management tool? How will access code be synchronized with the database tables?
- Cascading sounds like a powerful idea. However, it can cause confusion for users. There must be a way to determine from which database each set of constants came. Documentation must be clear and easily available.
- Do not use a “SuperObject” to manipulate data from all the databases. Make a separate class for each database interface and for each table. The making of SuperObjects is bad design: it causes physical coupling problems, and maintenance problems.

## 6.11 Event Generator

- MINOS should talk with the Fermilab CD/PAT group about use of HepMC, instead of StdHep++.
- It is a very good idea to include all information relating to testing and validation within the package that is being tested. This concept should be propagated to other MINOS software packages.
- This subsystem should have its own detailed review.

## 6.12 Supported Compilers and Platforms

Many of our comments relevant to this talk are in Section 5.

- The most recent gcc compiler (2.95.2) has reasonable Standard Library support, with the exception of some of the streams classes.
- We strongly suggest avoiding non-standard C++. Turning off RTTI and turning off exceptions or using “long long” are examples of non-standard C++. You should think long and hard before doing this, the experiment is going to be around for a long time and compilers and libraries are quickly improving.
- We agree that circular dependencies are bad, and should be disallowed; it is good that this talks points this out.
- We suggest that MINOS organize the build system to do layered releases, in which packages are placed into a hierarchy, and in which each level in the hierarchy can be released separately.
- Testing is essential. Each package should supply enough tests that will verify that the package is operating correctly -- within the context of the package and the system.

## 6.13 Schedule

Our comments on the schedule can be found in Section 2.2.

# 7 Conclusion

This document has covered only the more obvious issues brought to our attention during the review, and has not gone into sufficient detail for the many parts of the system presented during the talks. We see a clear need for a series of more detailed focused reviews. These reviews should be both smaller in scope, and longer in duration, than this initial review. It is also important that these reviews have some continuity in the review committee, because these subsystems must be integrated, and so their designs are to some extent inter-related. We have suggested reviews for at least the following subsystems:

- Data model

- Persistency
- Framework
- Event generator
- Magnetic field management

We are concerned about MINOS's lack of developer manpower. It is clearly necessary to make the most out of the available talent; toward this end, we have made the following recommendations:

- MINOS should adopt a "software process", including the establishment of design guidelines, coding guidelines, milestones of sufficient concreteness to be useful to both management and developers, standardized testing procedures, and reviews (as mentioned above).
- MINOS should use established standards whenever possible: Standard C++ (including the Standard Library), standard data formats, standard database APIs, standard networking APIs, *etc*.

In addition, MINOS should take more advantage of the experience within CDF and DØ, and within the Fermilab Computing Division (especially the PAT and Special Assignments groups). MINOS may also benefit by combining efforts wherever possible (for example, in producing a reconstruction and analysis framework, data model, and persistency scheme) with some of the smaller Fermilab experiments that have a compatible timescale.

We are concerned about the complexity of some of the proposed subsystems presented during the review. This complexity comes in several varieties: ease-of-use for non-experienced people, difficulty of maintenance when original designers move on to other projects, scalability and performance, and integration with other subsystems.