CDF/DOC/COMP_UPG/PUBLIC/3518

The Trybos Document

R. D. Kennedy
Rutgers University

Version 2.2
22-Mar-96

Abstract

Trybos is an Object-Oriented C++ package to read, write, and manipulate data in CDF YBOS format and is intended for use by C++ programs. This package is also used to explore the definition of a general user-data interface which is independent of the data persistence mechanism and yet still provides access to Run 1 data. The currently implemented user-data interface model involves accessing data elements in CDF Run 1 YBOS banks with inlined C++ accessor functions. This model can evolve to allow users to manipulate higher-level objects whose I/O is implemented through the use of YBOS banks.

Table of Contents

List of Tables

1 Introduction

Trybos (Totally Revised YBOS) is an Object-Oriented C++ package to read, write, and manipulate data in CDF YBOS format [1, 3] and is intended for use by C++ programs. Files written by Trybos are completely indistinguishable from, though not quite identical to, files written by the current CDF YBOS software package[1]. Trybos is written entirely in Standard C++ and uses no pre-existing CDF offline software. Trybos is not intended to be a "pin-out" replacement for the existing YBOS package, and it does not replace by itself CDF Offline Bank definitions and support. In particular, the memory management functionality of YBOS is not completely reproduced by Trybos since that functionality it provided by the C++ programming language.

This document describes version 2.1 of Trybos. It treats the current design in detail and the current implementation in general. To describe the Object-Oriented design of Trybos, a variation on the Object Modelling Technique (OMT) [4] is used. It is expected that this document will be filled out with examples and modifications as they are needed or as the package evolves.

1.1 Design Clusters

The design of the Trybos package is organized into four clusters[2], or tightly-knit sets of abstractions. Each of these is implemented by a number of C++ classes. They are:

- Bank — a collection of data fields of varying fundamental types
- Record — a collection of distinct Banks
- File — a collection of Records
- Dataset — a collection of Files

1.2 Usage Models

The Trybos design is in part driven by three usage models which represent the extremes of performance requirements: non-reconstruction processing, user analysis, and event reconstruction.

[1]The "next bank index" field in the YBOS bank headers will generally contain different values when output by YBOS than when output by Trybos, but this is invisible to the user. The files written by either package can be read by either package.

[2]The term *cluster* comes from the Eiffel programming language where it means a tightly-knit set of modules or classes. The term *module* is used in OMT for this, but this term has a very different meaning in many OO programming languages and is thus avoided.

1.2.1 Non-Reconstruction Processing

In non-reconstruction processing, one reads a Record from a File, finds a few numbers in Banks located near the beginning of the Record, and makes a processing decision based on those numbers. One might insert Banks into the Record, but only at the end of the Record. One does not generally erase Banks from the Record, though the operation is still supported in unoptimized form.

As an example, consider the operation of the Level 3 Event Reformatter programmer. It receives a set of data blocks which it interprets and then generates a small set of YBOS banks containing the needed fields from these blocks. After receiving the data blocks and clearing a Trybos Record, it creates a number of Banks outside of the Record. It sets the data fields values in the Banks and inserts the Banks at the end of the Record in the desired order. It then notifies the next Level 3 program that the Record is ready for further processing.

Another example is the Level 3 Dispatcher program. It receives a Record from the Level 3 Reconstruction program, and must decide whether or not that Record (containing a full-fledged CDF event, by now) meets any event requests that have been received from event consumers. To do so, it must find a specific Bank, the EVCL Bank, and extract the trigger bit information. It also finds the LRID Bank in order to extract the run number and other diagnostic information. Both of these Banks happen to be located in the beginning of a Record in the CDF "WIN" record format, so elaborate search optimization is unnecessary. If requested to do so, the Dispatcher program will drop specific "WIN blocks" of many contiguous Banks before sending events to consumers, something which only the Dispatcher program does.

1.2.2 User Analysis

In user analysis, one reads a Record from a File, loops over all instances of a particular Bank within a Record, finds a few numbers in each Bank, and makes a processing decision based on those numbers. One might insert Banks into the Record based on evaluation of the existing Bank data. One might also drop specific Banks deemed unneccessary for a particular analysis in order to reduce the output File size. Occasionally, one needs to find a specific instance of a Bank referred to by another Bank.

1.2.3 Event Reconstruction

In event reconstruction, one reads a Record from a File, executes a long series of independent processing packages or "modules". Each module is roughly modelled on the description of user analysis above. All information passed between modules is stored exclusively in Banks in the Record under consideration. As such there is an increased emphasis on adding Banks to a Record and on finding a particular instance of a Bank.

1.3 Assumptions and Requirements

The Trybos implementation requires a fundamental assumption about integer sizes which is captured in the header file `trybos_package.hh`. The type representing a 32 bit signed integer for each supported operating system is given the name `integer4`, since the YBOS data format specifically treats the storage of 32 bit integers. The exact size of integers, however, is not bound to a particular keyword in the Standard C/C++ languages. This type definition allows package code to be portable with only that header file to be altered when moving to a new system.

Eventually, that header file will contain similar definitions for system byte-order and floating point representations to isolate and localize these differences as well. XDR is not yet used in Trybos. Currently, only big-endian/IEEE[3] representation and little-endian/VMS[4] representations can be read and only big-endian/IEEE representation can be written. The big-endian/IEEE representation sub-set used by CDF just happens to be the same representation as XDR encoding. Thus, big-endian/IEEE representation files already supported by Trybos will remain readable once XDR is used by Trybos. Because XDR cannot generally handle little-endian/VMS representation (in other words, CDF Run 1A data) on non-little-endian/VMS machines, a set of data representation converts are included in Trybos in the collection of functions Trybos_Data_convert. Eventually XDR will be used to produce truly transportable, machine-independent data files.

Standard C library functions are used extensively by Trybos. The Standard C++ library [6] is not yet used extensively; only the iostream class is used. The Standard C++ string class may be used in the future. The Unix fstat() call is used only in the Trybos_Disk_File method setpos_near_end() and even then only as a last resort.

1.4 Style

The public interface for each class is defined in a header file bearing the class name and given the extension ".hh". Classes intended for public use are in the `include` subdirectory; classes intended only for internal use are in the `private` subdirectory. The inline implementation for each class, if any, is kept in a file bearing the class name and given the extension ".mm"[5] and are in the subdirectory `inline`. The compiled implementation files for all classes are in the subdirectory `src`. Their names vary, though all include the class name with the leading Trybos stripped to help archiver programs which cannot handle libraries with long element names.

[3]The big-endian/IEEE representation is the so-called "Unix" representation used by CDF in Run 1B

[4]The little-endian/VMS representation is the so-called "VAX" representation used by CDF in Run 1A

[5]The ".mm" extension is inspired by the '.m' extension used by Objective-C for class inline implementation files.

Trybos class names are capitalized, while methods are all lowercase. In the design documentation, the leading 'Trybos_' in class names is dropped to save space. Underscore characters are used to separate words or word combinations in class, method, and variable names.

Trybos classes generally take on an extension of the *orthodox canonical class form* [2]. They each declare at least a default constructor, a copy constructor, a destructor, the operation `=`, the operation `==`, and the operation `!=`. If one of operations is not intended to be in the public interface, then it is declared private.

All integer constants are defined as C++ enumerations to make them truly constant. Preprocessor definitions are not used except to control compilation. All constants are removed from the global namespace by embedding them in a class or a structure. Structure-embedded constants are always referred to by using the scoping operator `::` as if they were in a class.

1.5 History

Trybos is the third generation of the BOS data management package[6]. Trybos traces its own origins back to work by R. D. Kennedy to implement a fast, independent set of Standard C functions to read YBOS files for the CDF Run 1B Level 3 Trigger. Out of this grew Toybos (note the subtle name variation), a package by R. D. Kennedy in Standard C which supports reading and writing of events as needed in the CDF Level 3 Trigger as well as several bank search algorithms. It did not support all needed bank manipulations since it was intended to serve as a demonstration in principle that YBOS alternatives were possible with reasonable effort. Finally, as part of the CDF Run 2 Computing Upgrade project, a need for a C++ package to read Run 1 data was identified and work on Trybos began.

2 Bank

A Trybos Bank is a collection of data fields of possibly varying, but known, fundamental types. Banks within Records are referred as Stored Banks, otherwise they are referred to as Unstored Banks.

[6]Actually, Trybos was to be called Oybos, but that means "old ybos" in Unix culture. Oyveh was the next candidate, but I failed to come up with a meaningful phrase to go with the acronym. The current choice, Trybos, means the third generation BOS or the third choice for a name for the new YBOS package. The CDF YBOS package is counted as the second generation of BOS.

2.1 Design

The Bank cluster is broken down into a three different individual abstractions.

- Bank Key — the Bank name and number, unique within a Record
- Generic Bank — knows about Bank header and type information, used for generic algorithms on Banks
- Specific Bank — knows about individual Bank data fields

2.1.1 Bank Key

A Bank Key is a means of uniquely identifying a Bank, usually a Bank within a Record. It consists of two parts: the bank name and the bank number. All banks with the same name have the essentially same structure and are said to belong to the same bank family. Their data field values can vary, and in some cases the structure can vary in an orderly manner. Bank numbers are used to distinguish between Banks in the same family. Bank numbers are non-negative and do not, in general, follow any particular order within a Record.

2.1.2 Generic Bank

A Generic Bank is any YBOS Bank. Operations can be performed on Generic Banks that make use of the bank header and bank data type information, such as reading, writing, data representation conversions, printing, and other generic bank manipulations. This is not meant to be an abstraction that the user uses directly, in general. The Generic Bank is simply a means of collecting together all the common properties and behavior of YBOS banks for other specific bank abstractions to re-use.

Generic Banks know about an extra bit of information hacked into the bank number in the YBOS Bank format. This hack is orthogonal to the abstractions described here. A Squeezed[7] Bank has a bank number of at least 100, and an unsqueezed Bank has a bank number less than 100.

2.1.3 Specific Bank

[7]Squeezing is CDF terminology for compressing the data section of a bank using an algorithm optimized for the data fields in a particular Bank family

A Specific Bank is a YBOS bank with a well-defined set of data fields, a means of accessing those data fields, and possibly a means of setting those data fields. In the design, each Bank family is called a Specific Bank.

2.2 Implementation

The Bank abstraction is implemented by a set of classes, with each class representing one of the abstractions described above.

2.2.1 Trybos_Bank_Key Class

The Trybos_Bank_Key class is a simple class implementing a (bank name, bank number) number pair. Checks on the value of the bank name and number are made at the appropriate time to insure that they always lie in the defined range of validity. Bank names are required to consist of four ASCII printable bytes, or one of a few special codes to allow very limited regular expression in matching. Bank numbers are required to be non-negative or one of a few special codes to allow very limited regular expression in matching.

An OMT chart describing the Trybos_Bank_Key class interface is in Table 1 on page 16.

2.2.2 Trybos_Generic_Bank Class

The Trybos_Generic_Bank class describes and implements the generic properties and behaviors of YBOS Banks.

Stored and Unstored Banks must be treated differently in a number of methods, but both are described for now by Generic Bank. For instance, it would make no sense to erase an Unstored Bank since this means to remove it from a Record in which it is not stored. There are also issues of memory management to be considered.

One of the clumsy points in the implementation of Trybos is the treatment of non-native representation data while it is being converted to native data representation. In the Object-Oriented model, the Generic Bank should always have a well-defined state. Non-native data, however, does not since it cannot be interpreted as is. This somewhat academic conflict manifests itself when one attempts to place the "convert" algorithm somewhere in the class hierarchy. If it is placed in the Generic Bank implementation, which it is now, then care must be taken to convert the bank header and one type descriptor word first to define the state of the Bank, and in a separate method convert the remaining type description and data fields. The alternative used in an earlier version of Trybos put the convert method completely in the Basic Record class. This simplified the Generic Bank implementation, but required that the Basic Record class know the details of YBOS Banks which Generic Bank was expected to capture and encapsulate.

An OMT chart describing the Trybos_Generic_Bank class interface is in Table 2 on page 17.

2.2.3 LRID_Bank - A Specific Bank Example

Specific Bank classes are publicly derived from the Trybos_Generic_Bank class. At least a minimal set of methods are expected to be defined, as listed in Table 3 on page 18. There is expected to be at least one class per YBOS bank family, with newer versions of Banks defined by derived from older versions of Banks when possible. It may be useful for a single Bank family to have several different interfaces (constant versus mutable, raw versus validated), each of which would be implemented by a different class.

LRID_Const_Bank is an simple example of a Specific Bank. It is only an example and is not meant to be the default LRID class for the CDF Run 2 Computing Upgrade project. An OMT chart describing the LRID_Const_Bank class interface is in Table 4 on page 19.

3 Record

A Trybos Record is a collection of Banks, each Bank with a Bank Key that is unique within the record. A Record is the smallest unit of data that can be read from or written to a permanent storage device.

A Trybos Record is called a Logical Record in YBOS terminology. YBOS also contains the concept of a Physical Record. In the Trybos package, however, the concept of a Physical Record is hidden from the user within the Trybos_Disk_File class implementation.

3.1 Design

There are four abstractions in the Record cluster. There is one Record Iterator abstraction which is used to point to Banks within Records. There are three Record abstractions which vary in Bank storage and Bank finding mechanisms. These different mechanisms allow different trade-offs between CPU overhead per I/O operation and bank finding efficiency and bank manipulation efficiency.

- Record Iterator — points to Banks in Record, can move to next Bank

- Basic Record — No Bank indexing maintained
- Hashed Record — Hash table and Next Bank indices are maintained
- Listed Record — Hash table and search list are maintained

3.1.1 Record Iterator

A Record Iterator points to Banks in a Record and knows how to best advance to the next Bank. It is similar to the iterator concept embodied in the C++ Standard Template Library, where iterators are pointers to elements of a collection of some sort. The special iterator value `begin()` is defined to point to the first Bank in a Record. Also, `end()` is defined which points to the Bank *after* the last Bank in a Record. The Record Iterator can be advanced to the next Bank in a Record until the end is reached in which case `end()` is always the result.

A Record Iterator can also be used to perform those operations which only require the information present in the YBOS bank header, such as the Bank Key and the bank size. Since the creation of a Generic Bank involves some consistency checks on the header and type information which a Record Iterator does not perform, a Record Iterator is a faster means of simply locating or moving from Bank to Bank within a Record.

3.1.2 Basic Record

A Basic Record performs no special activities to help optimize bank searches. It is optimal for situations in which raw I/O rate is more important than Bank finding efficiency. This situation occurs in a number of Level 3 Trigger programs which only need to find one or two banks per event and these banks are intentionally placed at or near the beginning of events. The Basic Record I/O mechanism is re-used by all other Record abstractions.

3.1.3 Hashed Record

A Hashed Record maintains a Hash table and a set of Next Bank indices in order to speed up Bank find operations. The Hash table contains a table of First Bank locations and number of Banks, all organized by a hash of the Bank Key. In addition to this, the Next Bank index field in each Bank header is maintained. A Hashed Record is optimal for situations in which bank finding efficiency is somewhat more important that raw I/O rate. A Hashed Record is optimized for sequential bank finding within bank families in the pattern of find_first/find_next, but not for finding a specific bank or for inserting and erasing banks. This trade-off is probably appropriate for most user analysis software.

3.1.4 Listed Record

A Listed Record maintains a Hash table and a Search List of Banks in order to speed up Bank find operations and Bank insert/erase operations. The Hash Table contains a table of First Bank locations and number of Banks, all organized by a hash of the Bank Key. The Search List is a doubly-linked doubly-threaded list with threads for Next/Previous Any Bank and for Next/Previous Similar Bank. A Listed Record is optimal for situations in which find and insert/erase efficiency is much more important that raw I/O rate. A Listed Record is optimized for bank finding in the pattern of find_first/find_next, for finding a specific bank, and for inserting and erasing banks at the expense of some CPU overhead for each read and write operation. This trade-off is probably appropriate for the Level 3 Trigger and Production reconstruction executables and perhaps also some user analysis.

3.2 Implementation

The concept of a Record Iterator is implemented by a number of classes Each Record class has a nested class called simply Iterator which implements the Iterator functionality in an optimal manner for that particular Record class. To tie these nested Iterator classes together and establish some polymorphism, the abstract class Trybos_Abstract_Iterator is defined.

An alternative scheme was explored in version 2.0 of Trybos in which a single Trybos_Bank_Pointer class was defined. It could not iterate itself; each Record class provided a few optimal iteration methods for it. This left the Trybos_Bank_Pointer class non-virtual and as such improved performance by a few percent. The design was also a little cleaner than the current version.

The current design presents to the user an interface more like the Standard Template Library (STL) interface [5, 7] and is capable of being extended to provide nearly exact STL conformance. Exploring the possibility of using the STL algorithms for Bank manipulations, in my opinion, justifies the small degradation in performance of the current implementation. Furthermore, this design will be more intuitive to a user learning C++ and the STL for the first time. In any case, preliminary tests show that both versions of Trybos greatly outperform the current CDF YBOS package.

3.2.1 Trybos_Abstract_Iterator Class

The Trybos_Abstract_Iterator class is a small abstract class which exists only to provide some degree of polymorphism between the Record Iterator classes. The `where()` method in particular must allow polymorphic usage in order to permit use by the Generic Bank implementation, without tying Generic Banks to any one of the Record classes.

An OMT chart describing the Trybos_Abstract_Iterator class interface is in Table 5 on page 20.

3.2.2 Trybos_Basic_Record Class

The Trybos_Basic_Record class provides basic I/O functionality, including calls to other classes to perform bank search initialization and data representation conversion.

OMT charts describing the Trybos_Basic_Record class public and non-public interfaces are in Tables 6 on page 21 and Table 7 on page 22, respectively.

3.2.3 Trybos_Basic_Record::Iterator Class

A Basic Record Iterator uses a simple *bank walk* algorithm to iterate from Bank to Bank. No initial set-up is required for this algorithm. The bank walk involves simply adding the size of the Bank to which the iterator points to the iterator value. While this is relatively efficient for a small number of Bank searches, it can be very inefficient for a large number of distinct, and possibly repetitive, searches over hundreds of Banks.

An OMT chart describing the Trybos_Basic_Record::Iterator class interface is in Table 8 on page 22.

3.2.4 Trybos_Hashed_Record Class

The Trybos_Hashed_Record class re-uses the Trybos_Basic_Record class and adds bank search optimization at the expense of some initialization and maintenance.

OMT charts describing the Trybos_Hashed_Record class public and non-public interfaces are in Tables 9 on page 23 and Table 10 on page 24, respectively.

3.2.5 Trybos_Hashed_Record::Iterator Class

A Hashed Record Iterator uses an algorithm similar to the current CDF YBOS package. When a Hashed Record is read in, a hash table is filled with the bank names seen and the location of the first bank within a bank family. In addition, the "next bank index" field of each bank header is used to store the relative location of the next bank within a bank family[8], if any. Searches are thus optimized for the find_first/find_next pattern of Bank

[8] This differs slightly from the CDF YBOS package which stores the absolute location of the next bank. Storing the relative location simplifies the Trybos implementation by removing the need for Record infor-

usage. To find the first Bank within a family, the Hash Table is consulted and the location determined quickly. If the next Bank is sought, then the "next bank index" field is used. If a random Bank is sought, the first Bank in the family is considered, and then a walk over the Banks within the desired family is performed using the "next bank index" field. While not optimal, the CPU used to find Banks in this way is significantly less than using a Basic Record Iterator bank walk if many distinct, and possibly repetitive, searches are performed over hundreds of Banks.

An OMT chart describing the Trybos_Hashed_Record::Iterator class interface is in Table 11 on page 24.

3.2.6 Trybos_Hash_Table Class

The Trybos_Hash_Table class implements an open-address hash table with traced quadratic re-hashing. This hash table also aids directly in the initialization of the "next bank index" fields. Each insert operation involves hashing the bank name, inserting the current bank location at the appropriate place in the hash table as a previous location (storing the old value for later return), inserting the current bank location as the first bank location for this family if it is the first bank in this family seen, and returning the (stored) location of the previous bank of this family seen. For now, these two distinct roles for the Trybos_Hash_Table class are combined for implementation efficiency. In the future, one class for each of these roles may evolve from this one class.

The hash function used is maintained by the Trybos_Bank_Key class since it must be appropriate for the Bank Key private data members. The rehash function, however, is independent of the Bank Key representation and is maintained in the Trybos_Hash_Table class.

An OMT chart describing the Trybos_Hash_Table class interface is in Table 12 on page 25.

3.2.7 Trybos_Listed_Record Class

This implementation is not yet under development.

3.2.8 Trybos_Search_List Class

This implementation is not yet under development.

---

mation in the "next bank index" field maintenance, namely the Hash Table class.

4 File

A Trybos File is a collection of contiguous Trybos Records stored in a plain computer file.

4.1 Design

The Trybos File design cluster currently only includes support for local disk files. The design can be extended to support remote files, tape files, staged tape files, etc.

4.1.1 Disk File

A Disk File is a plain, binary computer disk-based data file.

4.2 Implementation

4.2.1 Trybos_Disk_File Class

The implementation of Trybos Files is more complicated than the abstraction since YBOS Records are actually broken up and stored in "physical records". Only the Trybos File classes know how to treat physical records, and the existence of physical records is completely hidden from other classes. Because of this physical record layer, more information than the `FILE` structure for a particular operating system must be maintained in the class in order to read and write Trybos Files. This extra information is hidden in the Trybos_Disk_File class.

The `setpos_near_end()` method may be unfamiliar to many users. The method will set the file position, if possible, to a point one or more events ahead of the end of a file. This method is extremely useful for the CDF Run 1B Level 3 Trigger package in simulation mode, allowing it to quickly find an End-of-Run record no matter the size of the input file.

An OMT chart describing the Trybos_Disk_File class interface is in Table 13 on page 26.

5 Dataset

A Trybos Dataset is a collection of Trybos Files. Presumably these files have something in common, such as having passed the initial requirements of a analysis topic. No design or implementation work has been done as yet on the Trybos Dataset. The concept is simply a placeholder for future work.

6 Other Elements

There are a few items that are not easily assigned to any one design cluster.

6.1 Trybos_Data

Eventually Trybos may evolve to treat the concept of a single data field as a first-class design element in order to encapsulate data representation differences between supported systems and data files. This has not yet been done. In the meantime, two sets of functions provide generic data field support: Trybos_Data_converts and Trybos_Data_Prints.

The Trybos_Data_converts functions convert data from little-endian/VMS representation into big-endian/IEEE representation.

The Trybos_Data_Prints functions define how to best print generic data values found in Trybos_Generic_Banks. The formats chosen can be revised by a Specific Bank.

6.2 Trybos_YBOS_Format

The Trybos_YBOS_Format header describes the details of the YBOS File, Record, and Bank formats by defining a number of integer constants. Most of these constants are simply array indices to particular elements in the YBOS format.

7 Tests and Examples

7.1 Trybos Regression Tests

There are a number of simple programs in the `test` subdirectory intended to test much of the package functionality after each revision. The programs are not intended to simulate physics analysis. The programs use a few data files stored in the `dat` subdirectory.

7.2 Trybos Release Examples

There are two programs in the `examples` subdirectory which simulate a simple physics analysis, a search for $J/\Psi \rightarrow \mu^+\mu^-$. The programs are identical, except that one uses a Basic Record and the other uses a Hashed Record.

7.3 CPU Comparison to CDF YBOS

A simple test of the CPU efficiency of Trybos relative to current CDF YBOS. In the `compare` subdirectory, there is a Fortran/Expand/Analysis Control version of the J/Ψ finding example programs. The comparison is made between the CDF READFILE module CPU statistics as reported by Analysis Control for the Fortran program and the total CPU consumed by the release example programs. The input file used contained about 7050 CDF Run 1B inclusive muon BPAD records filtered by a B meson semi-leptonic decay search and written in Run 1A VMS data format. When using the Trybos_Hashed_Record class, Trybos appears to accomplish in 25 CPU seconds what the READFILE module (CDF Offline 7.12) and a COMPARE module performing similar toy analysis take 45 CPU seconds to perform on an SGI Indigo running IRIX 5.3 with a MIPS R4000 CPU running at 100 MHz. Both programs were compiled with "-O2 -mips2" optimization. The times given are the average of at least three trials.

Great care should be taken in the interpretation of these numbers. No break-down has been made of where the CPU savings are actually occurring, whether it is from the file read, the data representation conversion, from the bank finding, or from some combination. Because these programs only search for a few Banks per Record, they do not fully exploit the optimization of bank finding provided by CDF YBOS and the Trybos Hashed Record class. Furthermore, comparing CPU efficiency for a Fortran program compiled with the vendor compiler to that of a C++ program compiler with the perhaps less optimal GNU compiler clouds the interpretation of the results. Nevertheless, the difference in CPU used is so large that I conclude that Trybos is already noticeably faster than YBOS for the functionality they have in common.

8 Commentary

Several familiar YBOS abstractions are not implemented yet. Bank Lists and Bank Subsets, used to filter which Banks are treated by Record I/O operations have not yet been considered for inclusion in Trybos. Work on tape file support has intentionally been avoided since there are many extraneous issues involved in accessing data on tapes, as demonstrated by the many software packages CDF uses to fully implement tape support for YBOS files.

This document is still rather incomplete, obviously, but it will grow in time as needed. See the examples and regression tests for demonstrations of what Trybos usage looks like in source code. See the slides in the presentation "Introduction to Trybos: Part 1" for OMT class diagrams and design discussion. The postscript file containing these slides should be in the Trybos `doc` subdirectory.

The top priorities for the next release of Trybos will be to include interface changes per suggestions, use XDR to support Intel-based Unix systems, improve the modularity of the implementation, and begin more thorough functionality and performance testing.

Table 1: Trybos_Bank_Key Class Chart

Trybos_Bank_Key: public interface
none
Trybos_Bank_Key(*default*) Trybos_Bank_Key(*copy*) Trybos_Bank_Key(integer4, integer4) Trybos_Bank_Key(char *, integer4) ~Trybos_Bank_Key() operator =(): Trybos_Bank_Key & operator ==(): bool operator !=(): bool operator <<(): istream & is_same_name(): bool is_inexact(): bool is_inexact_name(): bool hash(integer4 range): integer4

Trybos_Bank_Key: private interface
bank_name: integer4 = any_name bank_number: integer4 = any_number
none

Table 2: Trybos_Generic_Bank Class Chart

Trybos_Generic_Bank: public interface
none
Trybos_Generic_Bank(*default*) Trybos_Generic_Bank(*copy*) ~Trybos_Generic_Bank() operator =(): Trybos_Generic_Bank & operator ==(): bool operator !=(): bool operator <<(): istream & (FUTURE) is_stored(): bool is_squeezed(): bool (FUTURE) bank_key(): Trybos_Bank_Key next_bank(): integer4 bank_size(): integer4 print(): void

Trybos_Generic_Bank: protected interface
p_head: integer4 * = NULL p_type: integer4 * = NULL p_data: integer4 * = NULL
Trybos_Generic_Bank(integer4 *)

Trybos_Generic_Bank: private interface
data_type: integer4 data_groups: integer4 group_words: integer4 stored_in_record: bool
Trybos_Generic_Bank(integer4 *) convert(): bool convert_mixed_bank(): integer4 convert_mixed_block(): integer4 convert_mono_block(): integer4 print_mixed_bank(): integer4 print_mixed_block(): integer4 print_mono_block(): integer4

Table 3: LRID_Bank Minimal Interface Class Chart

LRID_Bank: minimal public interface
none
LRID_Bank(*default*) LRID_Bank(*copy*) LRID_Bank(Trybos_Abstract_Iterator &) LRID_Bank(Trybos_Generic_Bank &) ~LRID_Bank() operator =(): LRID_Bank & operator ==(): bool operator !=(): bool operator <<(): istream & (SUGGESTED) print(): void (SUGGESTED)

Table 4: LRID_Const_Bank Class Chart

LRID_Const_Bank: public interface
none
LRID_Const_Bank(*default*) LRID_Const_Bank(*copy*) LRID_Const_Bank(Trybos_Abstract_Iterator &) LRID_Const_Bank(Trybos_Generic_Bank &) ~LRID_Const_Bank() operator =(): LRID_Const_Bank & operator ==(): bool operator !=(): bool operator <<(): istream & is_consistent(): bool print(): void run_number(): integer4 run_type(): integer4 run_modifier(): integer4 partition_number(): integer4 run_unused(): integer4 experiment_type(): integer4 record_number(): integer4 raw_filename(): string record_type(): integer4 computer_id(): integer4 process_id(): integer4 process_release(): integer4 process_version(): integer4 creation_time(): time_t
LRID_Const_Bank: protected interface
none
LRID_Const_Bank(Trybos_Basic_Record &, integer4) LRID_Const_Bank(integer4)

Table 5: Trybos_Abstract_Iterator Class Chart

Trybos_Abstract_Iterator: public interface
none
Trybos_Abstract_Iterator(*default*) ~Trybos_Abstract_Iterator() is_null(): pure virtual bool bank_key(): pure virtual Trybos_Bank_Key next_bank(): pure virtual integer4 bank_size(): pure virtual integer4

Trybos_Abstract_Iterator: private interface
none
where(): pure virtual integer4 *

Table 6: Trybos_Basic_Record Class Chart - Public Interface

Trybos_Basic_Record: public interface
none
class Iterator: *see Table 8 on page 22*
Trybos_Basic_Record(*default*)
Trybos_Basic_Record(*copy*)
~Trybos_Basic_Record(): virtual
operator =(): Trybos_Basic_Record &
operator ==(): bool (FUTURE)
operator !=(): bool (FUTURE)
operator <<(): istream & (FUTURE)
find(Iterator &, Iterator &, Trybos_Bank_Key &): Iterator
find_first(Trybos_Bank_Key &): Iterator
find_next(Iterator &, Trybos_Bank_Key &): Iterator &
find(Iterator &, Iterator &, Trybos_Bank_Key &): integer4
insert(Iterator &, Trybos_Generic_Bank &): bool
append(Trybos_Generic_Bank &): bool
erase(Iterator &): bool
clear(): bool
is_empty(): bool
size(): integer4
begin(): Iterator
end(): Iterator
print(): void

Table 7: Trybos_Basic_Record Class Chart - Non-Public Interface

Trybos_Basic_Record: protected interface
p_buffer: integer4 * = NULL p_record: integer4 * = NULL buffer_size: integer4 record_size: integer4 record_offset: integer4
index(Iterator &): integer4 index(Trybos_Generic_Bank &): integer4 refresh_internal_state(): virtual bool bank_new(): virtual integer4 * bank_delete(): virtual void

Trybos_Basic_Record: private interface
none
convert(): bool

Table 8: Trybos_Basic_Record::Iterator Class Chart

Trybos_Basic_Record::Iterator: public interface
none
Iterator(*default*) Iterator(*copy*) ~Iterator() operator =(): Iterator & operator ==(): bool operator !=(): bool operator <<(): istream & advance(): Iterator advance_same_name(): Iterator is_null(): bool bank_key(): Trybos_Bank_Key next_bank(): integer4 bank_size(): integer4

Trybos_Basic_Record::Iterator: private interface
p_head: integer4 * = NULL p_parent: Trybos_Basic_Record * = NULL
Iterator(integer4 *, Trybos_Basic_Record *)

Table 9: Trybos_Hashed_Record Class Chart - Public Interface

Trybos_Hashed_Record: public interface
none
class Iterator: *see Table 11 on page 24* Trybos_Hashed_Record(*default*) Trybos_Hashed_Record(*copy*) ~Trybos_Hashed_Record(): virtual operator =(): Trybos_Hashed_Record & operator ==(): bool (FUTURE) operator !=(): bool (FUTURE) operator <<(): istream & (FUTURE) find(Iterator &, Iterator &, Trybos_Bank_Key &): Iterator find_first(Trybos_Bank_Key &): Iterator find_next(Iterator &, Trybos_Bank_Key &): Iterator & find(Iterator &, Iterator &, Trybos_Bank_Key &): integer4 insert(Iterator &, Trybos_Generic_Bank &): bool append(Trybos_Generic_Bank &): bool erase(Iterator &): bool clear(): bool / is_empty(): bool / size(): integer4 / begin(): Iterator / end(): Iterator print(): void

Table 10: Trybos_Hashed_Record Class Chart - Non-Public Interface

Trybos_Hashed_Record: protected interface
p_buffer: integer4 * = NULL p_record: integer4 * = NULL buffer_size: integer4 record_size: integer4 record_offset: integer4
index(Iterator &): integer4 index(Trybos_Generic_Bank &): integer4 refresh_internal_state(): virtual bool bank_new(): virtual integer4 * bank_delete(): virtual void
Trybos_Hashed_Record: private interface
none
none

Table 11: Trybos_Hashed_Record::Iterator Class Chart

Trybos_Hashed_Record::Iterator: public interface
none
Iterator(*default*) Iterator(*copy*) ~Iterator() operator =(): Iterator & operator ==(): bool operator !=(): bool operator <<(): istream & advance(): Iterator advance_same_name(): Iterator is_null(): bool bank_key(): Trybos_Bank_Key next_bank(): integer4 bank_size(): integer4
Trybos_Hashed_Record::Iterator: private interface
p_head: integer4 * = NULL p_parent: Trybos_Hashed_Record * = NULL
Iterator(integer4 *, Trybos_Hashed_Record *)

Table 12: Trybos_Hash_Table Class Chart

Trybos_Hash_Table: public interface
none
Trybos_Hash_Table(*default*) Trybos_Hash_Table(*copy*) ~Trybos_Hash_Table() operator =(): Trybos_Hash_Table & operator ==(): bool (FUTURE) operator !=(): bool (FUTURE) operator <<(): istream & (FUTURE) find_first(Trybos_Bank_Key &): integer4 * count(Trybos_Bank_Key &): integer4 insert(Trybos_Bank_Key &, integer4 *): integer4 * erase(Trybos_Bank_Key &): bool clear(): bool print(): void

Trybos_Hash_Table: private interface
p_bucket: Hash_Table_Node * n_buckets: enum n_entries: integer4
is_constructed(): bool is_empty(): bool is_full(): bool hash(Trybos_Bank_Key &): integer4 rehash_function(integer4, integer4): integer4

Table 13: Trybos_Disk_File Class Chart

Trybos_Disk_File: public interface
none
Trybos_Disk_File(*default*) Trybos_Disk_File(*copy*) (FUTURE) ~Trybos_Disk_File() operator =(): Trybos_Disk_File & (FUTURE) operator ==(): bool (FUTURE) operator !=(): bool (FUTURE) operator <<(): istream & (FUTURE) open(char *, Io_Mode): bool rewind(): bool setpos_near_end(): bool flush(): bool close(): void read(Trybos_Basic_Record &): bool write(Trybos_Basic_Record &): bool is_open(): bool is_OK(): bool is_at_end(): bool

Trybos_Disk_File: private interface
plain_file: FILE * save_first_record_offset: integer4 save_physrec_number: integer4 words_in_physrec_buffer: integer4 save_first_record_offset: integer4 *
read_first_physrec(*long arg list*): integer4 read_subsequent_physrec(*long arg list*): integer4 write_first_complete_physrec(*long arg list*): integer4 write_subsequent_complete_physrec(*long arg list*): integer4 write_incomplete_physrec(*long arg list*): integer4

Bibliography

[1] K. Biery *et al.*, "The CDF Event Structure", CDF Internal Report CDF/Note/152, August 1993.

[2] James O. Coplien, *Advanced C++: Programming Styles and Idioms*, (Addison-Wesley, Reading, MA, 1992), ISBN 0-201-54855-0.

[3] D. Quarrie and B. Troemel, "YBOS Programmer's Reference Manual - Version 4.0", CDF Internal Report CDF/Note/156, January 1996.

[4] James Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, (Prentice Hall, Englewood Cliffs, NJ, 1991), ISBN 0-13-629841-9.

[5] A. Stepanov and M. Lee, The Standard Template Library, 07-Feb-95 (postscript is available from butler.hpl.hp.com:/stl or at http://www.cs.rpi.edu/~musser/doc.ps).

[6] M. Vilot, "The C++ Standard Library", *C++ Report* **7** (2), 28 (February 1995).

[7] M. Vilot, "An Introduction to the STL Library", *C++ Report* **6** (8), 22 (October 1994).