Software for DAQ Systems

Bob Jones
CERN, Geneva, Switzerland

Abstract

This paper describes the major software issues for data acquisition systems for high energy physics experiments. A general description of a DAQ system is provided and followed by a study of a number of its components in greater detail. For each component its requirements, architecture and functionality are explored with particular emphasis on the real-time aspects. The paper concludes with an attempt to indicate what will be the most significant changes for DAQ software between the present day and start-up date of the LHC detectors.

1 Introduction

When one thinks of the size and complexity of the proposed LHC experiments, it is clear that a rigorous and long-term research effort is required if the array of new detectors are to be capable of handling the previously unheard of data rates and volumes. It follows that new hardware devices, not yet conceived during the LEP era, will find their way into these detectors, DAQ systems and computing centers. What does not automatically spring to mind is that a similar revolution will happen to the software as well. Why? Because our existing techniques for developing and maintaining software that can keep such experiments running efficiently, day after day, will not scale to the proposed proportions. As the number of people, processors, file systems and interconnects increase so does the effort required to keep them all working together. In response, a number of research projects have put an emphasis on the software aspects of their work. In particular, the RD13 project at CERN has concentrated a significant part of its time on the software that dominates DAQ systems.

The aim of this paper is to describe the major software issues for DAQ systems. We start with a general description of a DAQ system and proceed to study a number of its components in greater detail. For each component its requirements, architecture and desired functionality are explored with particular emphasis on the real-time aspects that affect its design. Some aspects of a DAQ (such as detector calibration) are not addressed since, from a real-time point of view, they are of less importance. Throughout the paper the RD13 DAQ is used as an example implementation in the hope that its requirements and design decisions will help the reader to consider such issues in a more practical and fundamental manner. The tour concludes with an attempt to indicate what will be the most significant changes for DAQ software between the present day and start-up date of the LHC detectors.

Due to time constraints, there is no list of references in the paper. However, the reader may browse the RD13 WWW server (URL: http://rd13doc.cern.ch/welcome.html) or contact the author directly (by email jones@vxcern.cern.ch) who will happily supply pointers to the relevant sources of information.

2 Overview of a DAQ system

The most general description of the purpose of a DAQ system will not change as we approach the LHC era and so Sergio Cittolin's description of the UA1 DAQ system is still valid:

"The main purpose of a data acquisition system is to read the data from the experiment's instrumentation and to store to tape that corresponding to physically interesting events. This must be done with the maximum of efficiency and while providing a continuous monitor of data validity and the detector performance."

He goes on to indicate what effects the characteristics of the detector have on the design of the DAQ:

"The high trigger rate and the large data volume of the detector information demand a powerful and versatile data acquisition system. The complexity of the equipment requires continuous monitoring with programmable means of error detection and error recovery. The data have to be reformatted and compacted in size and the event rate has to be controlled and reduced by programmable filters. So a large amount of computing power has to be made available as close to the detector as possible, a high degree of parallelism has to be implemented and sequential processes have to be pipelined as much as possible to limit the dead time and to maximize the effective use of the experiment at run time."

With this in mind, let us look at how one particular project intends to approach these issues for LHC experiments.

3 The RD13 DAQ system

The RD13 project was approved in April 1991 for the development of a scalable data taking system suitable to host various LHC studies. The basic motivations come from the conviction that, being too early for a 'top-down' design of a full DAQ architecture, a lot can be gained from the study of elements of a readout, triggering and data acquisition and their integration into a fully functional system.

3.1 Main goals

The time scale for LHC experimentation and the inadequacy of the existing readout and DAQ technology make a 'top-down' design premature. A more appropriate preparation for LHC is to spend some time and resources in investigating system components and system integration aspects. The investigation is more effective if done in a realistic environment,

such as the data taking phase of a detector prototype at a test beam. Such a setup has the double advantage of serving the increasing data taking demands from the evolution of the detector readout and triggering electronics on one hand and, at the same time, helping the smooth evolution of the DAQ system by forcing a continuous integration of newly developed components into a working environment.

A further motivation drives RD13, the conviction that the traditional standard High Energy Physics methods for online software developments would fail, with the obvious disastrous consequences for LHC experimentation, given the undeniable complexity of the required systems. Much has to be done to find suitable methods for complex online system designs and much has to be learned in the area of software engineering tools. The ability to be constantly evaluating and experiencing in an environment close to the real one, is the great advantage of a 'down-to-the-earth' learning ground, as proposed by the RD13 collaboration.

To solve the problems which have motivated our project, four major goals were envisaged in the RD13 proposal and have constituted the working plan of the RD13 collaboration in its first phase.

1. The core of the project is the construction of a *DAQ framework* which satisfies requirements of *scalability*, in both number of data sources and performance (processing power and bandwidth); *modularity*, i.e. partitioned in functional units; *openness*, for a smooth integration of new components and extra features. It is necessary to point out that while such features are easily implemented in a well designed hardware architecture, they constitute a big challenge for the design of the software layout. This is our first goal.
2. Such a DAQ framework is an ideal environment for pursuing specific DAQ R&D activities, following a 'bottom-up' approach, for the clear identification and development of the building blocks suitable for the full system design. The fundamental R&D activity is the investigation of the use of *Real-Time UNIX operating systems* in RISC-based frontend processors, to assess their combined potential, to converge towards operating system standards and to reach a full platform independence.
3. The control of all the aspects of a sophisticated DAQ system demands specialised software to complement UNIX with additional services and facilities. Other aspects of DAQ R&D, more directly dependent on a modular architecture, involve the integration of suitable *commercial products*.
4. The last project goal is the exploitation of *software engineering* techniques, in order to indicate their overall suitability for DAQ software design and implementation and to acquire expertise in a technology considered very powerful for complex software development but with which the HEP community remains unfamiliar.

3.2 Prototype DAQ

A test beam read-out facility has been built for the simultaneous test of LHC detectors, trigger and read-out electronics, together with the development of the supporting architecture in a multiprocessor environment. The aim of the project is to build a system which incorporates all the functionality of a complete read-out chain. Emphasis is put on a highly modular design, such that new hardware and software developments can be conveniently introduced. Exploiting this modularity, the set-up will evolve driven by progress in technologies and new software developments.

One of the main thrusts of the project is modelling and integration of different read-out architectures to provide a valuable training ground for new techniques. To address these aspects in a realistic manner, the group collaborates with detector R+D projects in order to test higher level trigger systems, event building and high rate data transfers, once the techniques involved are sufficiently mature to be tested in data taking conditions. The complexity expected for the software online system of LHC experiments imposes the use of non-traditional (within HEP) software development techniques. Therefore, the problems of data acquisition and support software are addressed by the exploitation of modern software engineering techniques and tools.

3.3 Test-beam activity

The first prototype DAQ was used at a test-beam of the SITP (RD-2) detector R&D in November 1992 for 10 days during which nearly 5 million events were recorded. This version of the system is based on VMEbus, using the VICbus (Vertical Inter-Crate) to link VME crates and to integrate backend workstations (sun SPARCstations and HPs) with the frontend processors (MIPS 3000 based CES RAID 8235 boards) via the SVIC (Sbus to VIC interface). All the processors (front and back end) run UNIX as the common operating system. A Real-Time version of UNIX (EP/LX, a port of LynxOS to the MIPS architecture) has proved extremely suitable for use in the frontend RISC processors. Software engineering techniques have been applied to the development of a data flow protocol and a number of commercial products have been used to develop a run-control system, database applications and user interfaces.

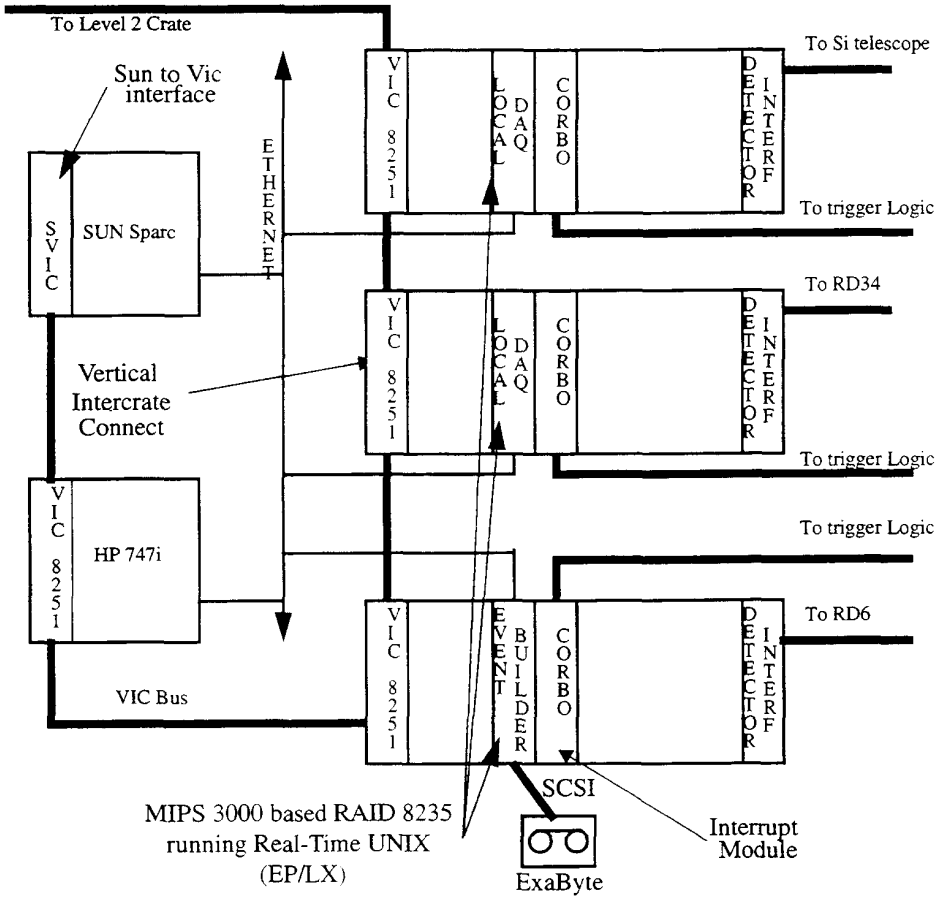
A second implementation has been developed and was used with the TRD (RD6) detector during 1993/1994 test-beam periods. The RD6 test beam required higher performance due to the read-out of 2 TRT sectors (512 straws each) via a HiPPI link into a HiPPI destination module acting as a high speed local acquisition during the SPS spill. The control and monitoring of the frontend electronics was performed via a slow control (SCM) module. A second level trigger memory could optionally be read out at the end of the spill. Some 400 MB of TRT events (540 bytes each) have been stored on tape by the RD13 acquisition and analysed by RD6. The performance of the RD13 DAQ was more than adequate for the task: when running with an intense beam, some 6000 events could be stored in the HiPPI destination memory in less than 0.5 seconds, read-out into the RD13 DAQ (without using the DMA facility) in some 2 seconds and recorded at exabyte

8500 speed (400 KB/sec). Indeed the limiting factor was the size (3.5 MB) of the HiPPI destination memory which allowed for only a fraction of the SPS spill to be sampled.

A combined detector run including RD6, RD34 (hadron calorimeter) and RD21 (Si telescope) is being prepared for September 1994 using a test-beam setup similar to the above (see figure.)

Given the very positive results obtained from the project, activities will continue with further test-beam setups and developments around Real-Time UNIX including the introduction of a multi-processor version. Software engineering applications (including the introduction of object-oriented technology) will enhance the functionality of the current DAQ and ensure platform independence (both front and backend) by evaluating new workstations and processors. The system will also be used for event building studies and the evaluation of high throughput R/O modules.

FIGURE 1 DAQ Hardware setup at ATLAS test-beam 1994



4 Real-time operating systems

A modern operating system environment (OS) for data acquisition applications must have features which span from fast real time response to 'general purpose' DAQ tasks (run control applications, monitoring tasks, etc.). The real-time environment, therefore, requires a compromise between *real-time performance* (i.e. a deterministic response to external events) and *standardization* (i.e. platform independence) in a system which supports *distributed processing*. Real time UNIX systems seem, today, the best candidates for achieving an advanced level of uniformity and compliance in OS standards with real time capabilities.

4.1 Why UNIX

UNIX characteristics which make of it a widely spread operating system (or better still the choice for almost any new computer platform) can be summarized in the following points:

- UNIX is a generic operating system (i.e. its architecture is not targeted to any proprietary hardware architecture) and is mostly written in C. Hence it is easily adaptable to different computer architectures (in a sense it is easier to write a C compiler than to write a new operating system).
- Although several flavours of UNIX exist, they all have a large common base. In addition the two forthcoming UNIX standards, System V release 4 (SVR4) and OSF/1, do provide an external interface which is an enhanced synthesis of existing flavours.
- Current efforts towards operating system interface standardization (in particular the POSIX activities) are very close to UNIX. Although POSIX interfaces will also be available on non UNIX systems (e.g. VMS).

In this respect UNIX provides, with certain restrictions:

- Platform independency: this is both from a general point of view (the choice of a workstation becomes less important) and from the application point of view (having different UNIXes in the backend and frontend may not be a problem from the application point of view).
- Programmer portability: people can move from a platform to another (i.e. from an operating system to another) with minimal retraining.
- The best path towards operating system interface standardization.
- A flow of enhancements which is less controlled by the computer manufacturer.

UNIX does present also a number of drawbacks, particularly in a real time environment such as ours:

- It is known to have a long learning curve, from the user's point of view: commands are cryptic and the user interface dates from the days of teletypes. Yet window (typically X11) based user interfaces and desk top managers are now available which provide users with graphics based, Macintosh-like, access to the operating system.

- UNIX is not suited for real time applications. The typical list of grievances is a superset of the following:

  The Unix kernel is not pre-emptable: an interrupt may be delayed for an indefinite amount of time is it comes while the kernel is running.

  There is no priority base scheduling, time-sharing is the only scheduling policy available.

  There is no asynchronous I/O nor provision for asynchronous events (signals are a poor substitute for the latter).

  There is normally no provision for direct I/O and/or connection to interrupts.

The question may arise of why not choose a combination of UNIX in the backend and a real time executive (e.g. VxWorks) in the frontend. This choice would not meet our requirements for at least the following issues:

It would be difficult to maintain uniformity in the overall data acquisition system.

Using a real time exec means dealing with two systems: development on a UNIX workstation, running on the real time exec.

Applications may need more functionality than is normally available in a RT exec.

UNIX is the future and a high degree of platform independence is required. While execs are somehow UNIX-like and may evolve towards POSIX, they are currently not UNIX.

4.2 Real time needs

A, non exhaustive, list of real time applications in our environment would include:

- Hardware read out driven by interrupts. Typically for

  Final data acquisition read out, at the bottom of the system after event building.

  Read out at the sub-detector level.

  Slow control applications (reaction to events happening in the overall system).

  Here we need a quick response to an external interrupt and easy access to the external (typically, but not exclusively, VME) I/O resources. Also primitives for efficient inter-process communication and management of multiple event sources in an asynchronous way are necessary.

- Drive and/or monitor specialized hardware, fast response to interrupt and quick access to I/O resources are here, as well, of primary importance.

- Data Acquisition (DAQ) applications such as:

  Event Distribution

  Event Monitoring and Analysis

  Recording

Run Control

Here too efficient IPC and treatment of multiple, asynchronous external (to a process) events (not to be confused with physics events) are necessary to design and assemble a DAQ.

A major question in real time computing is that of performance. An important issue, in particular, is that of determinism: how long an external event (interrupt) can be delayed, because of other activities in the system, without making the system worthless. It is clear that, in our environment, determinism is not an issue per se: normally (with the possible exception of slow controls) data acquisition (i.e. read out of an event) is a process which is relatively long compared to interrupt handling. What we really need is:

- To service an interrupt efficiently: this means not only sheer speed (i.e. 10 usec. maximum interrupt latency) but also flexibility to write the software to catch and treat the interrupt (interrupt routines available inside a user program, synchronization primitives between interrupt routine and main thread, etc.).
- Low overhead for process scheduling and context switching: threads would be welcome.
- Low overhead for IPC calls: for example semaphore handling, in particular semaphore switching without waiting or rescheduling, should be efficient.
- Efficient extension to multi-processing environment.

4.3 The EP/LX Real-time UNIX operating system

A survey of the operating system market has led us to choose LynxOS as currently the best kernel for our requirements. This choice combines the VME RISC processors (RAID 8235) as a frontend platform with the port to the MIPS architecture (EP/LX) of the LynxOS kernel marketed by CDC. EP/LX includes a real-time kernel; the MIPS port of LynxOS 1.2 which is a complete UNIX system fully compatible with both System V and BSD 4.3 and some additional real-time features not found in LynxOS, in particular access to the VME/VSB busses and to the RAID 8235 card hardware facilities. EP/LX also implements a scalable multi-processor model where multiple CPUs, while maintaining their own copy of the kernel, co-operate by the extension of their inter-process communication primitives (shared memory, global semaphores, named pipes) to a cluster of processors communicating via VME or VSB. Future versions of EP/LX will include LynxOS 2.0 features, including POSIX compliance. A large number of UNIX "intensive" applications have been ported to EP/LX such as ISIS, VOS (a heavy user of UNIX system calls), Motif applications, the Network Queueing System (NQS) and all the major CERN libraries. Additional products include QUID generated code, the Artifex run-time library and the client part of an Object Oriented Database system (ITASCA).

5 Modelling the architecture of a DAQ

The DAQ systems for a detector at a future collider like LHC will have to cope with unprecedented high data rates (~10 GByte/s), parallelism (100 to 1000 processors) and new technologies (e.g. SCI, ATM). Simulation of different architectures, algorithms and hardware components can be used to predict data throughput, the memory space and cpu power required and to find bottlenecks before such a system will be actually constructed. Therefore one needs a modelling framework with a high level of description and a clear mapping between the system to be built and the system to be modelled. The framework has to be modular and scalable to allow simulations of the different configurations from simple systems up to full DAQ systems for big detectors.

5.1 A DAQ simulation framework

The modelling framework presented in this paper is written in MODSIM II which is an object-oriented language for discrete event simulation and has its own graphics library.

The modelling framework itself consists of a library of generic objects for the DAQ simulation (DSL, DAQ Simulation Library), a graphical user interface (GUI) and a tracing facility for off-line analysis of the simulation results.

The package has been developed in the RD13 project and is still evolving while a working version is available. It has been used for small applications and is used for event building studies and used for DAQ simulations by the ATLAS collaboration.

5.2 The DAQ Simulation Library (DSL)

The main idea of the DSL is to use the smallest indivisible (**"atomic"**) processes that can then be used to build up any DAQ system. A dozen **"atomic"** processes have been defined and make the core of the DSL.

The DSL has a generic level consisting of objects for a generic description of DAQ systems, and a user level where inheritance is used to combine the generic objects with user dependent features. Thus the DSL contains the possibility to refine the objects and to include hardware dependent features.

As an example of an application of the DSL the readout of the combined RD6/RD13 testbeam in November 1993 has been simulated. This setup consisted of a single chain of data flow using a HIPPI link and had a total data rate of 1.5 MByte/s. This example was used as a proof of principle: it showed the easy mapping between reality and simulation and the consistency between the values measured and the values simulated. The simulation could then be used for changes of parameters and extensions of the setup.

FIGURE 2 RD6/RD13 testbeam setup

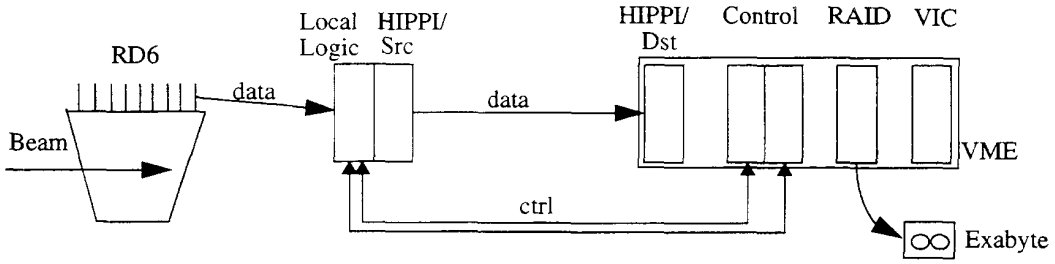
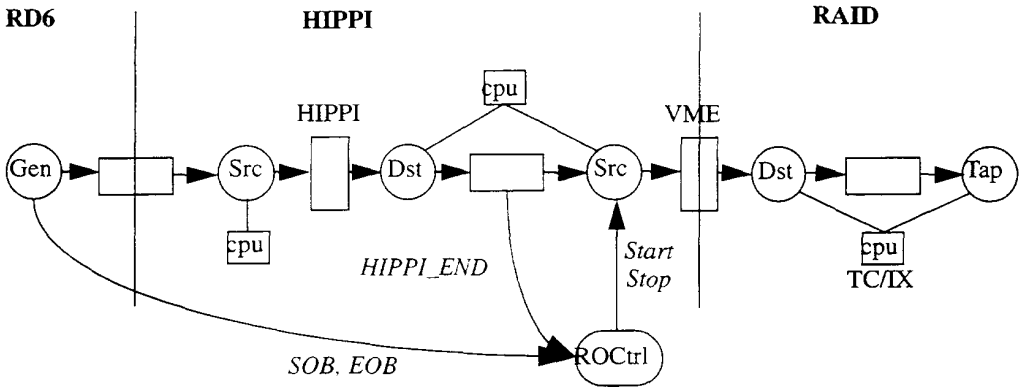


FIGURE 3 The Simulation Model of the RD6/RD13 Testbeam Setup



Using this model we are now in a position to replace the various elements (e.g. HiPPI with ATM) and observe the effect on the performance of the DAQ. Such work will continue in parallel to the testbeam activity during the years leading up to the final detector installation.

6 The flow of data

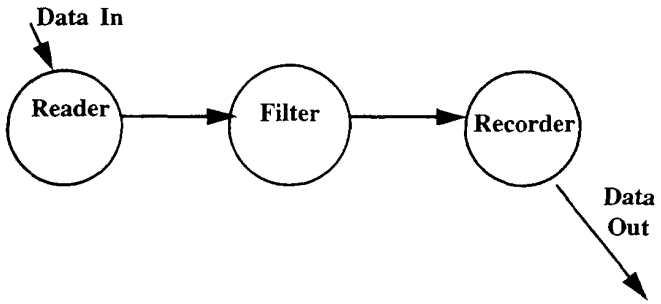
The dataflow is the data acquisition component responsible for moving and distributing data (sub-events or full events) to the various entities throughout the system. An entity is a generic component of the data acquisition system characterized by an input (from which data will flow in from) and one or more output (to which the transformed data will flow out to). We require that a common set of rules (the dataflow protocol in the following) be available to the various entities in the system to compose a data acquisition system (by interconnecting entities in a sort of network) and transferring data within the network from the detector to the recording medium, independently of the physical medium used to move the data. Flexibility is required to allow the data to be transferred over a number of media (busses, networks, shared memory etc.) and be scalable so that larger organizations of elements can be made.

6.1 System requirements

The basic component of the data flow is a DataFlow Element (DAQ-Module). It provides a service (e.g. transforms data), is implemented by a program and run by a process. A complete data acquisition chain consisting of an input element, 0 or more data transformers and 1 or more output elements is called a DAQ-Unit.

The data acquisition, from the dataflow point of view, is a collection of interconnected DAQ-Units. Shared memory is used to "transport" data within a DAQ-Unit.

FIGURE 4 A simple DAQ-Unit configuration



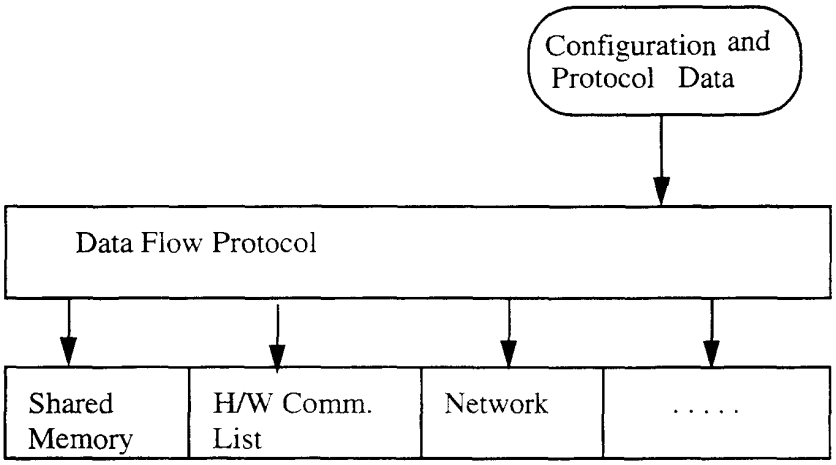
6.2 Dataflow protocol

A dataflow protocol controls the dataflow transactions with a library of basic primitives to implement the protocol within a program. Connection management, data transfer and synchronization are the functionalities required of the protocol.

The dataflow protocol handles 1(source) to 1(sink) connections, 1 (source) to n (sinks) connections (with possibility of selecting a policy for the distribution of the event, e.g. fan out the event, distribute the event selectively, etc.). The n (sources) to 1 (sink) (i.e. the case of event building-like functionality) or more in general the n to m case are lower priority (indeed event building protocols will be available inside event builders, or they could be expressed as sequences of operations to be performed on hardware modules) and their inclusion in the data flow protocol should be traded against the introduction of additional complexity.

The dataflow protocol should be independent of the particular medium used to transport data between a source and a sink of data. Provision must be made for control signals (e.g. run control commands).

FIGURE 5 Dataflow Protocol Layering



6.2.1 Constraints

The dataflow subsystem must coexist with other activities in a software module. That is a module waiting for an event to come should not block the reception of other signals such as run control messages. This suggests a multi-threaded approach to structure a data acquisition module. The design and development of an efficient and correct protocol is a complicated task for which we chose to use a software engineering methodology and related CASE tool.

6.2.2 StP

The first CASE tool we chose is called StP (Software through Pictures) by IDE (Interactive Development Environments). Stp is a set of CASE tools supporting analysis and design methods for software production. It offers diagram editor tools for data flows, data structure, structure charts, entity relationships, control specifications, control flows and state transitions. There is also a picture editor and object notation editor which allow the user to associate a set of properties and values with any objects in any editor diagram. A document preparation system allows designs to be printed in many formats.

To model the protocol we used a real time structured analysis method, as described in Ward/Mellor and Hatley/Pirbhai. Both variations of the method are supported by StP. The model is defined, specifying what the system must do in terms of hierarchy of control and data flow diagrams going from the context diagram down to the specifications. The application database and all the other data structures used to manage the protocol are defined according to the entity relationship data model, or the equivalent data structure. The protocol itself is specified in terms of state transition diagrams. These different modeling tasks are performed by means of the corresponding StP editor, the results are

then stored in the StP data dictionary and are available both to the StP editors and to user defined modules - e.g. code generator.

We had to overcome some of the StP deficiencies:

- Since a protocol is basically a state machine, the decomposition in state transition diagrams would make the definition clearer. Unfortunately StP does not support this feature. We made a special coupling of data flow and state transition diagrams that allowed us to use the DF nesting to define the missing ST hierarchy.
- More over, a code generator not supported by StP would be a great help to make the analysis-design-implementation cycle seamless.

At this stage of the project we decided that it would be interesting to redo the exercise with another tool which supports another method but has the features that we found missing in StP.

6.2.3 Artifex

The second methodology/tool that we tried is called PROTOB, an object-oriented methodology based on an extended data flow model defined using high level Petri Nets, called PROT nets, and its associated CASE toolset called Artifex, by Artis S.r.l.

Artifex consist of several tools supporting specification, modeling, prototyping and code generation activities within the framework of the operational software life cycle paradigm. Artifex fully supports system analysis, design, simulation and implementation through the same GUI.

- The analysis and design phase is done using the graphical formal language for the high level concurrent Petri Nets, that integrates sections of C and Ada code.
- The simulation generation and execution is done with two buttons only. During this phase the user can simulate, set break points and step through the concurrent task model.
- The emulation supports distributed code generation from the same model that was used for analysis and simulation. During emulation visual monitoring of the Petri Net is possible using the same GUI used for the two previous phases.

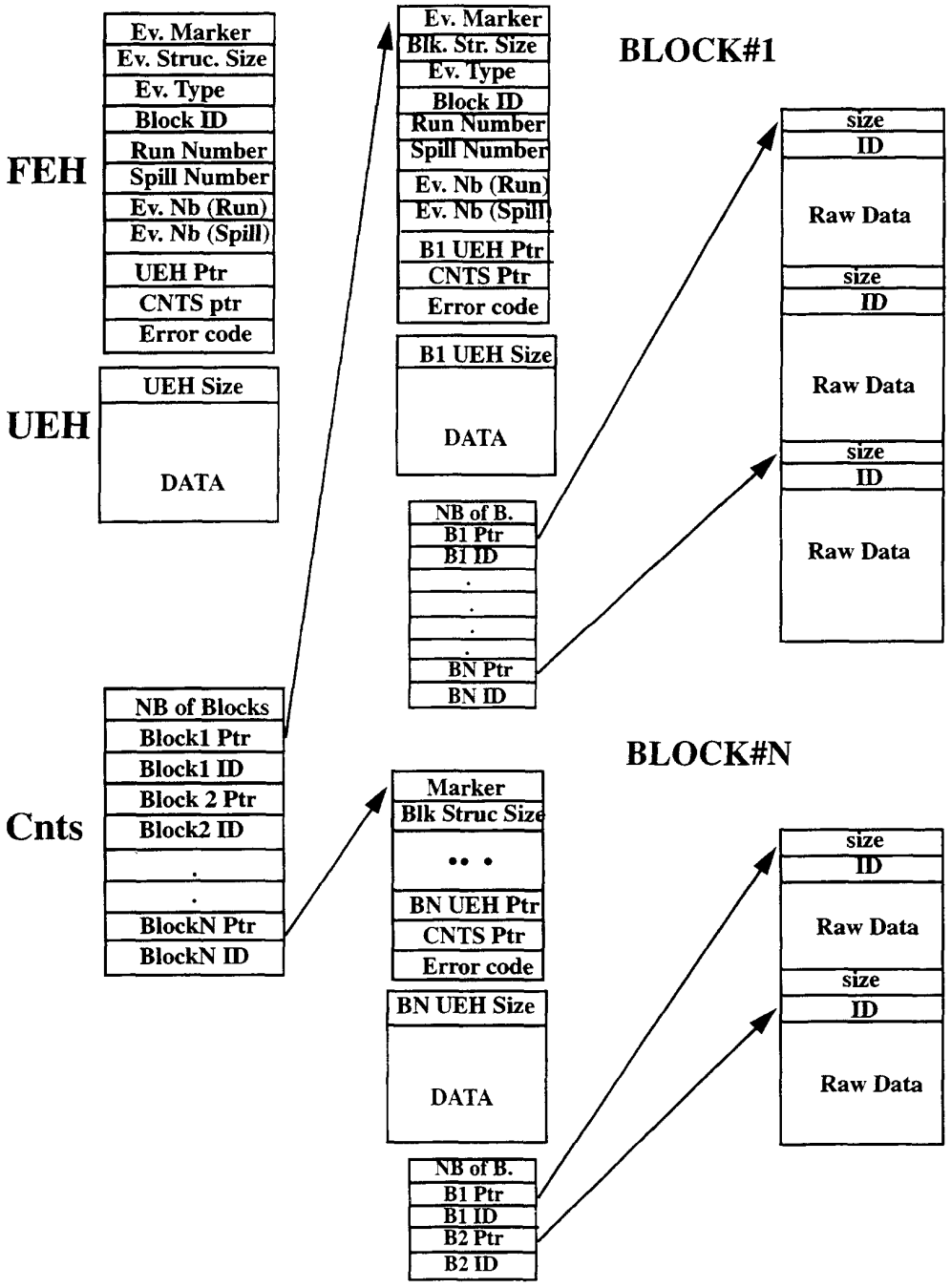
The protocol is specified in terms of High Level Concurrent Colored Petri Nets. This model can be seen as a combination of the data flow and state transition of the SASD methodology. The object model of Artifex allows independent module definition, testing and reuse.

7 Event formatting

In order to achieve scalability and modularity the DFP needs to support DAQ configurations where additional data sources (detectors) can be included in the system and

the event stream parallelised on a sub-event basis in parallel before merging into a single event for further processing. While the event input (read-out) is performed a structure is built for each single event using the *event format library*. To provide the kind of modularity we need this event format has to be defined recursively, in such a way that the structure of data is *identical* at each level of sub-event collection or event building. Navigating the structure from the top, an event consists of an event header (EH) followed by detector data. The detector data themselves may then be structured in one or more EH followed by data, where these latter EH are related to a substructure of the event. This process of sub-structuring may continue for an indefinite number of levels, in a tree-like structure. The leaves of this tree contain the original detector data.

FIGURE 6 Event Structure implemented by the Event Format Library



8 Event building

An Event Building System (EB) is the part of the data acquisition system (DAQ) where the data coming from different streams of the data flow (usually different sub-detectors) is merged together to make full events that can be sent further downstream the DAQ to either the next level of filtering or directly to a mass storage system. Thus the main tasks of an Event Building System are to transfer and to merge data.

Event Building Systems usually consist of special hardware for the purpose of transferring and merging the data, and software to control the transfer and the merging, to allow user communication and error detection as well as some monitoring. Apart from the generic studies it is necessary to evaluate emerging hardware techniques which are likely to be available at the time the LHC detectors will be built.

As the EB deals with data it is important to give a model of that data. Every data (which are the data associated with a physical event and containing the data coming from a sub-detector) have three layers:

- Actual Data:

  It is an accumulation of bits and bytes somewhere in some memory. Throughout the paper a static view is chosen, i.e. the event data exists or does not but it is never being built. The amount of data is fixed and well known.

- Description of Data:

  For every piece of data there exists a description. This mainly is a key to access and interpret the actual data stored in the memory. The description contains information on the memory location, the number of bytes, the format of the data stored and so on. The format of the data can either be hard-coded or dynamic (i.e. retrieved from the event library, database etc.)

  The description of the data itself can be part of the actual data, but in any case it has to be stored some where in the memory as well.

- Synchronization/ Timing:

  Every piece of data has a timing aspect: it can be available or not and it can be discarded from the memory. All these events in the life of any piece of data can be seen as signals to the EB (e.g. NEW_EVENT)

  Usually this aspect of the data is not an integral part of it but it logically belongs to it, signals are used by processes to notify the availability of new data or its loss. The timing is thus only defined in the context of the processes in which the data is being handled.

Furthermore every piece of data has an identity which might consist of several fields and usually is part of the actual data (it is in the description where it can be extracted from the actual data). It can also be part of the data description and the signals themselves The three layered model of data is shown in the following diagram:

FIGURE 7 Event builder: The three aspects of data

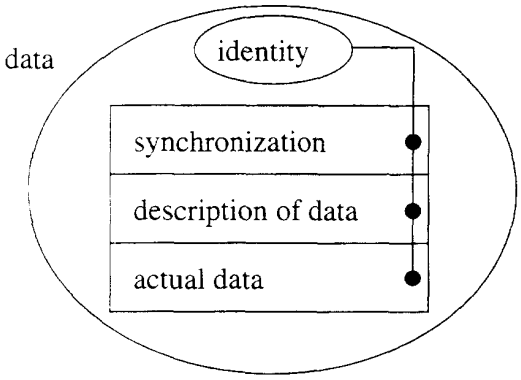
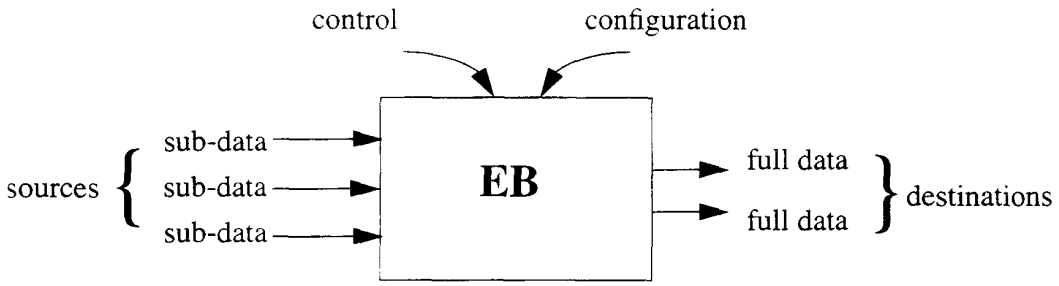


FIGURE 8 Event builder: an abstract model



8.1 Functionality

The two main functionalities of an EB are:

1. Transfer of data

   Signals are sent, descriptors are sent and the actual data has to be transferred from one location in memory to another. All these function are carried out by different ways of transferring the different aspects of the data from one location to another. This functionality must be reflected in software as well as in hardware (a physical connection for the transfer).

2. Merging of sub-data

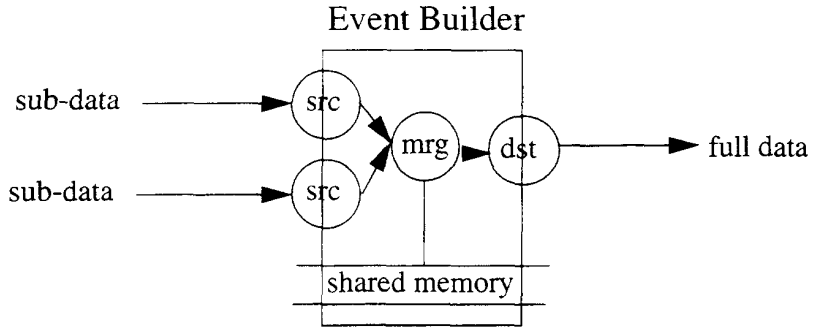
   Reacting on the signals, the descriptors need to be interpreted and merged. The actual data can be moved to a memory location for full data or the merging is done just in the description without moving any part of the actual data. But in any case the data coming from an EB will be made of one or several portion of data before the EB.

Synchronization with the upstream sources of data and downstream destinations is crucial to the efficient running of the event builder. It is necessary to add validation checks on the data (e.g. have all the sub-detectors delivered their data?)

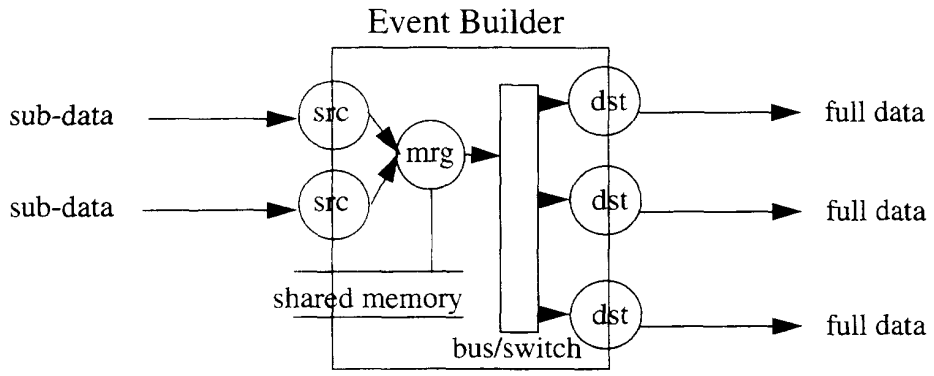
Internally the three layers of data can be handled in different ways. There might even be different hardware for each of the layers (control - interrupt on a bus system, description - ethernet, actual data - HIPPI). Two approaches can be taken to organizing the transfer and merging of data. In the "push" approach a new event arriving at the source will drive an action; in the "pull" approach the destinations are the pulling the data through the EB.

8.1.1 Event building in software

This example shows 2 sources and 1 destination. Of the three different layers the sources only receive the signals (from some process upstream) and the data descriptors, the actual data is written into a shared memory. In the EB a merging takes place (maybe without moving any data) and the destination can send signals, data descriptors and finally the actual data further downstream.



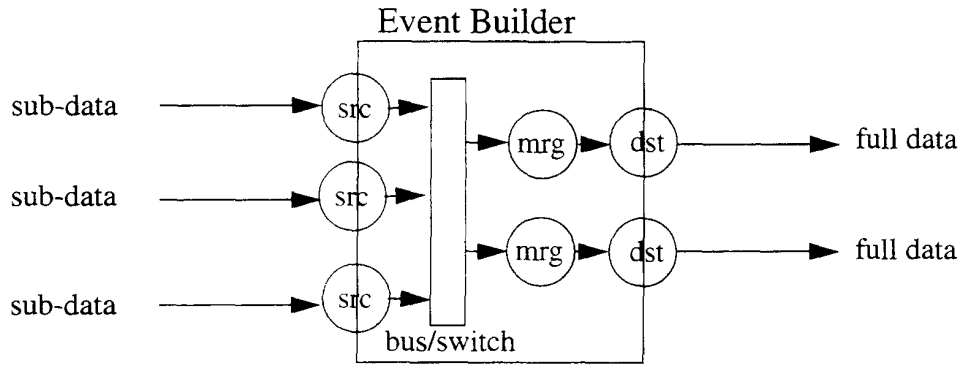
8.1.2 Event building with event distribution:



This example is an extension of the previous one: following the step of Event Building from the 2 Sources there is an additional step of distributing the full data to several Destinations. The hardware part to achieve this can either be a bus system or a high speed switch.

8.1.3 Parallel event building:

This example shows an Event Building System for N Sources and M Destinations: the sub-data is sent over a bus or high speed switch to merging processes running in parallel. The full data can be sent to several Destination at the same time.



In all three examples above it has not been defined how the data should be transferred, how the actual merging is done and what hardware devices can be used. The elements src, dst and mrg just represent the functionality of receiving the data from upstream, sending them downstream and actually merge the sub-data. They are transparent.The control part managing the sources and destination as well as the resources of the EB is not shown.

9 Monitoring the data

Monitoring of the data acquired by the system is needed to control the quality of the data. Online monitoring tasks are programs that receive a sample of the event from the DAQ while it is running and check their contents, produce histograms or perform initial analysis.

9.1 Requirements

The monitoring scheme should fulfill the following requirements:

- Allow programs to attach to any node in the DAQ-Unit to spy events flowing.
- Sample events on request.
- Provide means to tune the effect of monitoring on the main data acquisition, by limiting the amount of sampling requests.

9.2 Monitoring skeleton

All the monitoring tasks are based on a template, or skeleton which hides the details of the underlying DAQ but provides rudimentary access to the data. The programming interface is a skeleton, the same as that for other data flow modules, and a set of functions, known

as Data Flow Monitoring functions (DFM), called upon the occurrence of precise events in the data acquisition:

- ***DFM_Init***()

  called by the skeleton at program start-up.

- ***DFM_Process***()

  called by the data flow package each time an event is received by the data flow module. This routine may modify the incoming event (e.g reformat the data) and tell the data flow package what to do with the event (i.e. reject or forward the event)

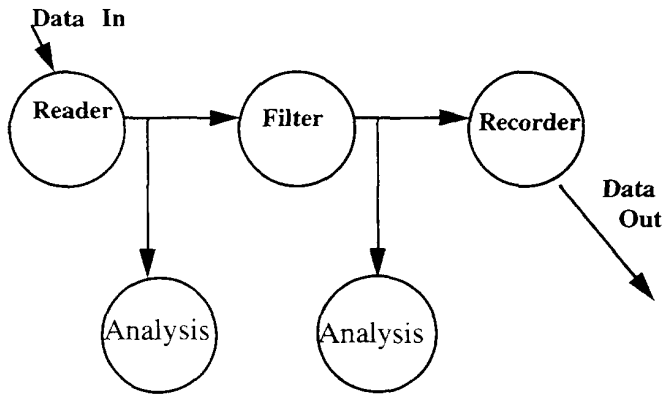
- ***DFM_Exit***()

  called by the skeleton at program termination so that the task may perform any necessary clean up.

- ***DFM_Communicate()***

  called when a communication request has been received by the dfp. It is then up to the programmer to take the necessary actions, such as popping up windows on the workstation screen to interact with the user.

FIGURE 9 A simple DAQ-Unit with attached monitoring tasks



9.3 Adding a graphical user interface to monitoring tasks

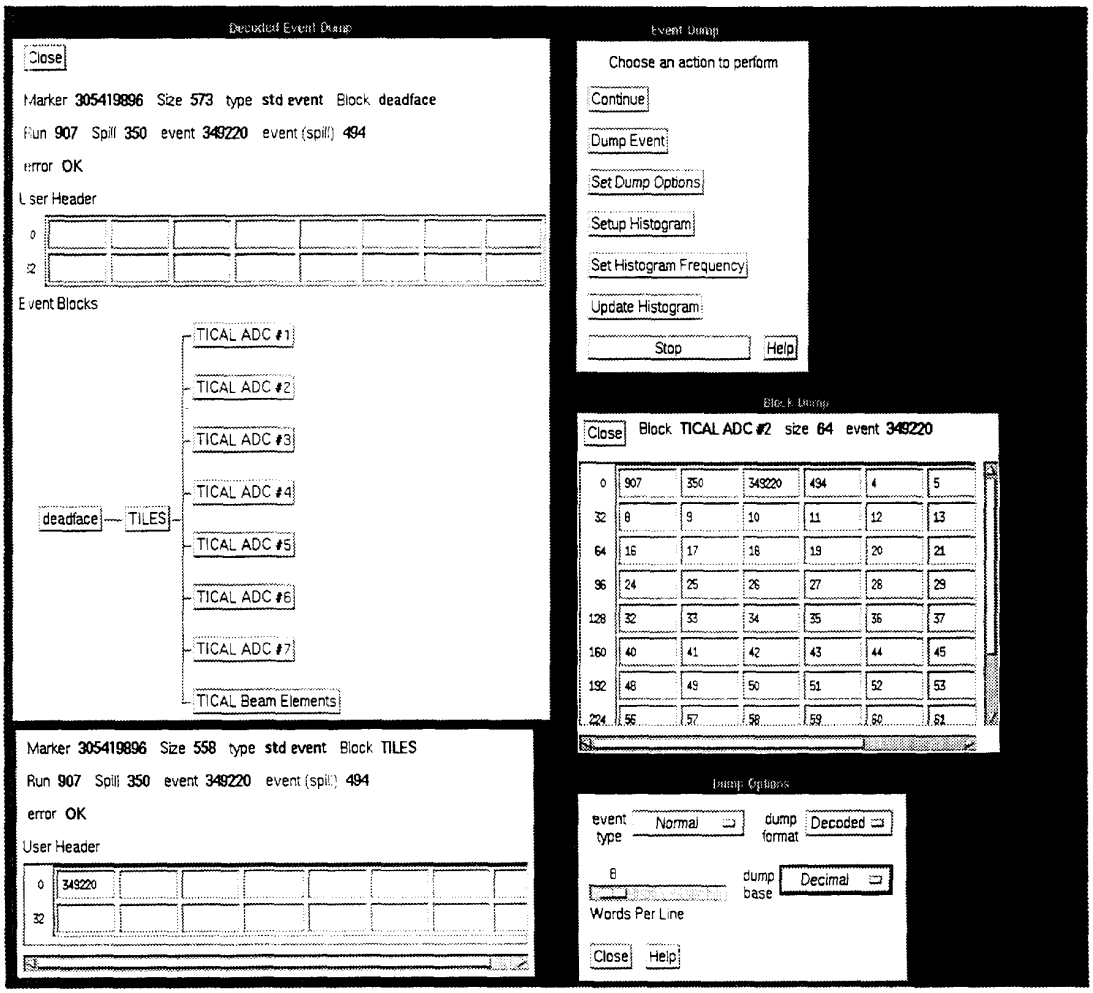
Monitoring tasks are built on top of the skeleton in order to receive events from the acquisition system. The skeleton handles "physics" event that come from the detectors in an asynchronous manner. Most modern windowing packages (e.g X window) are also event driven and require their "graphical" events to be treated by the application. The difficult part is making the "physics" event handling coexist with the "graphics" event handling inside the same application. This requires the use of some signal handling package that understands both type of events. In the RD13 DAQ, the monitoring skeleton is built on top of such a package, VOS (Virtual Operating System). All that was necessary

to incorporate both types of event was to declare the "graphics" events to VOS so that the appropriate routines could be fired when either of the two types of events are received.

The second issue is how to enforce some form of synchronization between the monitoring skeleton and user input. For example, imagine the monitoring task displays a window and should respond when the user presses a button contained within it. In this case we want the "graphics" event to be handled but the "physics" event to be blocked until the button is pressed. Again we can use the same signal handling package (i.e. VOS) for this purpose.

An example of a monitoring task with a graphical user interface is the event dump. The purpose of an event dump is to display the contents of an event to the user so that they can check its contents. It is a useful debugging tool which allows the physicists to discover if the detector read-out is producing data in the correct format without having to wait for a tape to be written and analyzed.

FIGURE 10 Event dump showing event decomposition and data block contents



10 DAQ databases

A data acquisition system needs a large number of parameters to describe its hardware and software components. In short, four distinct databases can be envisaged for a DAQ to store such information: Hardware configuration (describes the layout of the hardware in terms of crates, modules, processors and interconnects); Software configuration (describes the layout of the software in terms of processes, services provided, connections and host machines); Run parameters (e.g. run number, recording device, Level 2 trigger state etc.); Detector parameters (information pertaining to the detector and defined by the detector group themselves within a fixed framework).

10.1 Architecture

The database system must provide access to the data at run time at an acceptable speed. Typically we do not want to do any, or we want to minimize as much as possible, disk access at run time when the database is referenced. The database contents must visible to all applications in the system which need access to it - both in the frontend processors and in the backend workstations. The access must be shared. In a simple case this may be done with multiple concurrent readers (shared read transaction) and a notification mechanism to signal (rare) database modification.

In general we may identify a backend and a frontend component in the online database, the backend one deals with the permanent storage of data and all the operations related to a modification of the schema, the frontend one deals with the actual access at run time. Depending on the database technology and the product chosen, the two (back and front) parts may or may not be provided by the product itself.

10.2 Data Access Library (DAL)

The database is accessed by DAQ programs via a data access library. A DAL provides an interface to the queries and updates needed by the applications, that is the DAL implements only those database operations needed (and allowed) on the data at run time. The DAL hides the actual implementation of the database to the application as well as the actual implementation of the data model. This means that the underlying database system can be replaced without affecting the application programs.

10.3 User interface

A common set of tools must be provided to implement an easy means to browse and update the database according to the facilities provided by the DAL. That is the interactive programs must be capable of performing all the operations allowed at run time. Generic browsing and updates, assumed to be rare and performed by experts only, are made using the commercial DBMS query language and facilities.

10.4 First implementation: StP/ORACLE

The first implementation of the database framework was made using "traditional" (i.e. for which extensive expertise exists at CERN) technology: typically the entity-relation (E-R) data model and relational database technology (as implemented by the ORACLE dbms). The E-R data model has already been used in several experiments and has shown to be adequate for the purpose. A CASE tool was used to design and develop the data model for the DAQ's software configuration database.

10.4.1 Entity-Relation data model CASE tool

Software Through Pictures (StP) is a CASE tool implementing several software engineering methods via diagram editors. In particular it has an E-R editor with the possibility of producing SQL code to create ORACLE (and other relational DBMS) tables and C data structures defining entities and relationships. The tool was used in the following steps:

- StP is used to draw the E-R diagram implementing the database schema.
- StP is used to generate code from the E-R diagram: SQL statements to create Oracle tables, C type definitions for the table rows, C arrays implementing the tables in memory and C functions to load/dump a database from memory from/to disk files or Oracle.
- The DAL is hand coded in C as a set of functions working directly on the arrays in memory.
- Each application reads from disk (or Oracle) a copy of the database, then operates on it. There is no provision for distributed access to the data (which, in this case, is not needed because the contents of the database are updated off-line when a new DAQ configuration is defined).
- Versioning (i.e. the maintenance of different configurations) is handled off-line by having different files.

10.4.2 Real Time facilities and distributed access

ORACLE is not suited for real time distributed access to the data, nor are ORACLE interface available for the frontend processors. In order to use ORACLE, an architecture needs to be devised that avoids this limitation.

For real time access we adopted a two level scheme. A backend level where the database is created and maintained in ORACLE (creation, insertion, updates, etc.); and a frontend level where, at run time, the contents of the database are extracted from ORACLE and read into an application's memory (ORACLE tables are mapped onto arrays of data structures representing database rows). The DAL navigates the in-memory tables when required to access the data. Since the initial loading of the database may be relatively slow,

and in any case we also need to access the data from the frontend processors, we envisaged an intermediate level where the ORACLE database is read into a workstation disk file first (NFS being available throughout the system) and then all the applications just read the disk file to load their in-memory tables.

The major drawback of this scheme comes from the need to hand code the data access library directly in C, in particular for what concerns the navigation of the database.

10.5 Second implementation: QUID

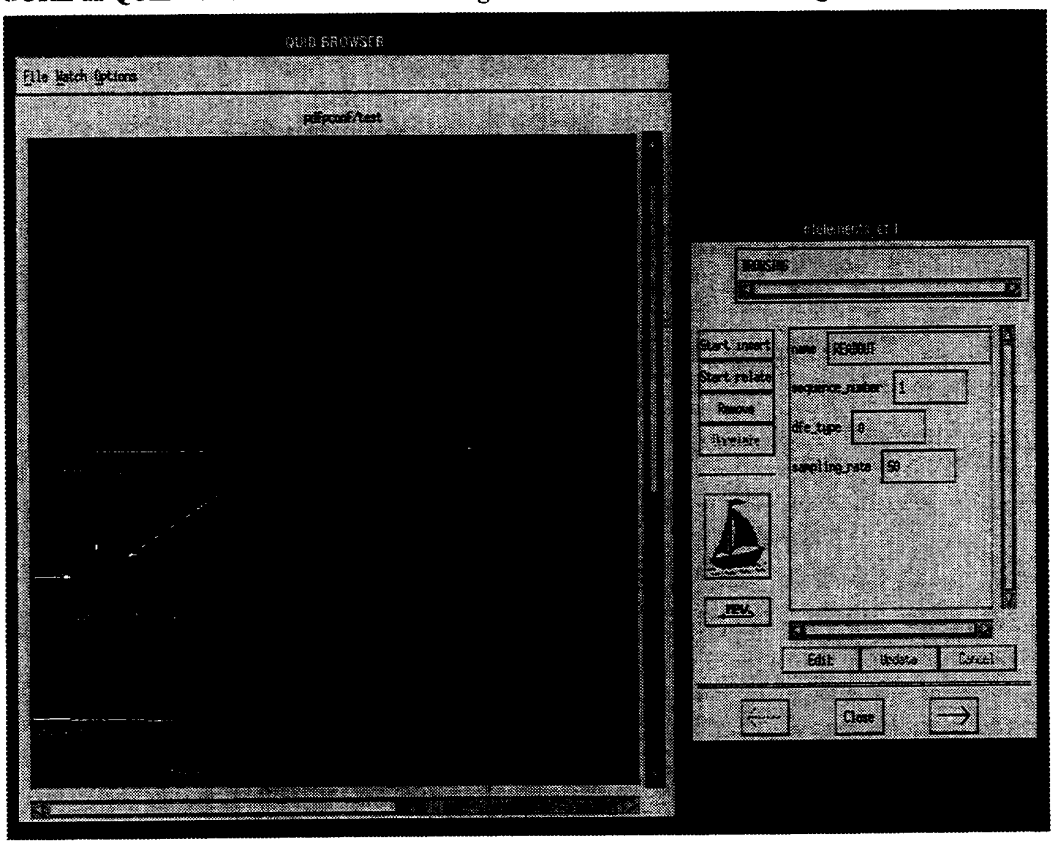
An alternative to commercial relational dbms are in-memory bank systems. They do not have all the functionality of the relational dbms but also do not have so many overheads in terms of performance and demands on the underlying operating system. One such commercial in-memory system is QUID from Artis srl.

QUID is a database development environment targeted to real time applications (i.e. where performance is needed and the full functionality of a DBMS is not) consisting of the following components:

- A E-R diagram graphical editor, implementing an extended E-R model.
- A query language: it extends the C language by providing statements to manipulate entities and relations and to navigate the database. Actions, expressed by C code, to be performed on the data can be attached to the QUID statements.
- A code generator: this takes as input both a database definition, to generate all the code needed to manage the database, and a QUID program, to generate a C module operating on the database.
- User interface (modifying and browsing a database): a tool which, given a database description as produced by the QUID editor, can produce an application to navigate and modify the contents of a database. This is an important point, since it provides a way to interactively access the data without the need for DAQ developers of build dedicated graphical frontends.

The database is kept in the application memory, its contents can be saved/loaded to/from a disk file. Multiple versions of a database, e.g. different system configurations, may be maintained by saving/loading to/from different files. There is no provision for distributed transactions, nor for the distribution of data among many processes. For read only databases this restriction is easily overcome by having all participating processes to access the disk files via NFS. When writing is also needed, a special scheme has to be devised.

FIGURE 11 QUID editor and browser showing one view of the software configuration database



10.6 Limitations of QUID

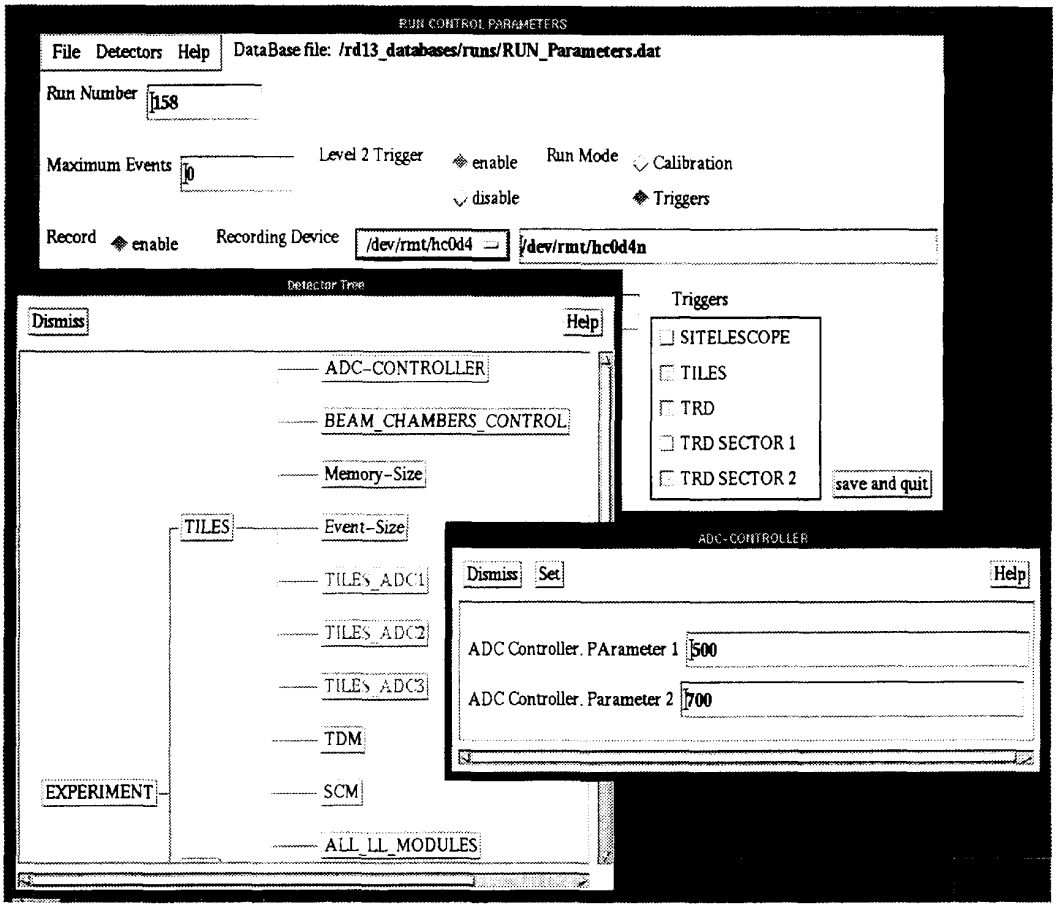
The QUID database system is currently used to store this data. QUID allows the modelling, storing and handling of the data but it is not a full database management system since it relies on the host's file system for storing data and does not provide a multiuser environment. It has some other important deficiencies:

- all the database is in the memory of the application which sets strict limitations to its size
- no referential integrity or concurrency control is provided
- no schema evolution facilities are provided. If a schema change takes place then the data that correspond to the old schema are no longer valid.

QUID has been successfully used to implement all the four DAQ databases mentioned earlier. This situation is satisfactory for the present but with the arrival of OODBMS, it may be possible to extend the functionality of the databases. HEP groups are now just starting to investigate the use of OODBMS systems for use in applications such as event recording. We have also started to make an implementation of the DAQ databases using

such a system and, from our initial impressions, it appears that OODBMS have a lot to offer the online environment.

FIGURE 12 User interface to run and detector parameters database



11 Run control

During the various phases of operation of the DAQ, a large number of operations need to be performed on the many hardware and software components. These operations must be performed in the correct order and synchronized in relation to each other and external constraints. The organization of such steps is the responsibility of the run control system. The run control system must be omnipresent, with a hand on all the components of the DAQ to guide them through these steps during the run cycle and react to changing conditions and required configurations.

It is easier to envisage the run control system as two levels: a basic facility that allows communication between components and understands the principle of "state"; and an

upper layer that represents the knowledge of how the steps to be performed are inter-dependent.

11.1 Basic run control facility requirements

- Provide a means of exchanging run-control commands between DAQ components
- Integrate with the multi-tasking environment of programs
- Provide equivalent support on both the workstations and frontend processors
- Provide a user-interface from which the status of components can be monitored and commands sent.
- Provide a means of implementing a run-control program in which the knowledge of the various steps can be represented
- Integrate with the error reporting facility
- Integrate with the database access mechanism
- Allow the operator to select from a number of DAQ configurations interactively without having to perform any programming tasks.
- Be independent of the DAQ configurations and capable of growing to incorporate more sophistication as the DAQ evolves (for example, control multiple DAQ units).
- Recover gracefully from unexpected events such as:

  time-outs on communication links

  crashes or mal-functions of components

  unsolicited changes of the state of some components

  On such occurrences, the run-control should continue to function and maintain as much as possible of the DAQ in a running state. For example, if the mal-function only affects one DAQ unit then other DAQ units should continue to function normally.

11.2 Definition of Finite State Machines

As a means of implementing the run control program, a facility for expressing the behavior of the various components in terms of finite state machines (or some derivative) can be provided. In short, a finite state machine represents all the behaviour and functionality of a process in a limited number of "states". While in a state, a process performs some predefined function. A process can swap its state by making a transition to a new state when some "event" occurs. For example, a tape recorder can leave the state "recording" when the user presses the "stop" button or when it reaches the end of the tape. An extension of this idea is to put "conditions" on the state transitions so that a state change is made when an event occurs and only if the conditions are met. Also, we can foresee a number of actions that must be performed during the state transition. For example, when the tape recorder goes from "recording" to "stopped" we must stop the motor that turns the tape.

The components, as well as the run-control program itself, should be capable of executing such finite state machines (FSM). The definition of the FSMs should be stored in a database and retrieved at run-time by each component. This permits the selection of appropriate FSMs according to the DAQ configuration being used. An FSM contains the following attributes:

- A list of states
- A list of commands that can be accepted while in each state
- For each command, a list of actions to be performed
- Actions can be

  send a command to another component (asynchronously and synchronously)

  execute a piece of code (user routine)

  set the state of the FSM
- A list of conditions for each state that, when true, perform a list of actions

  Conditions remain active while in the associated state and are re-evaluated whenever a component changes state, starts or stops

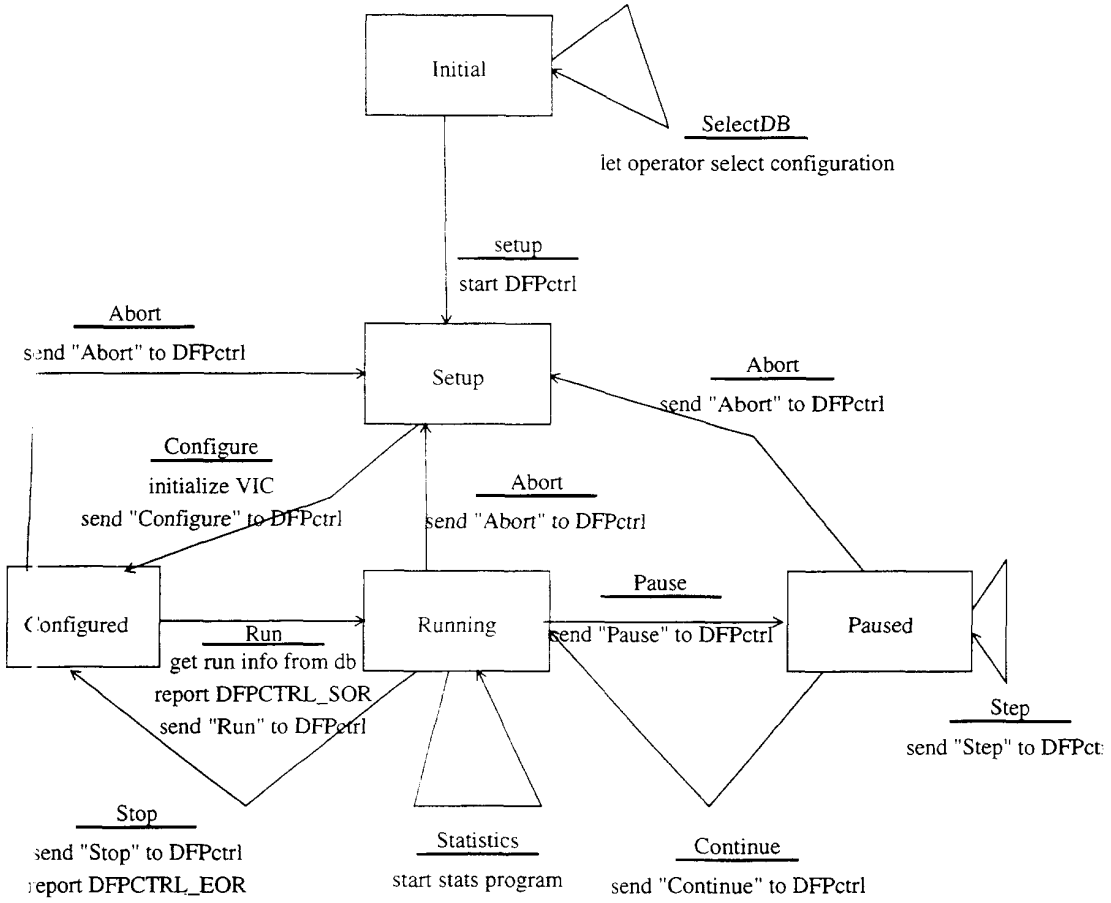
Once a FSM has been loaded from the database, it is put in the initial state and the system must keep track of the current state. The implementation of the FSM integrates with the multi-tasking environment of components.

11.3 Hierarchy of FSMs

The components are arranged in a hierarchy with the Run Controller at the top and controllers for individual DAQ units at the bottom. In between there may be intermediate levels of controllers though the hierarchy is not expected to be very deep. Commands flow downwards and replies ripple back up the hierarchy. A lower level component should not send a command to its parent controller. The only way a child component affects the behavior of its parent is by changing its own state which may cause a condition in the parent's FSM to become true and hence perform some action.

In principle, the facilities described should be used by every process that runs in the experiment and needs to communicate with the run-control system, or other processes. However, using the run control facility does imply an overhead in processing and requires that the process give control of the execution to a tasking sub-system which passes control to the application code when a message is received. In the case of a DAQ unit it may be better to create a separate process which uses the run control facility and controls the actual DAQ unit processes itself.

FIGURE 13 Run Control Program Finite State Machine



11.4 Requirements on the environment

Modern data acquisition systems can no longer be implemented using a single program, they involve many programs running on a network of computers. Such distributed systems demand specialized software in order to control all the aspects of the data acquisition system. Processes need to cooperate to perform processing functions, they need to share data and information. It must be possible to synchronize processes and monitor their progress. Operating systems, such as UNIX, do not provide all the services and facilities required to implement such features and so we have to look else where to find tools that can provide the missing features. One such tool is Isis, a toolkit for distributed programming. Isis started life as a research project at Cornell University and has since become a commercial product distributed by Isis Distributed Systems Inc.

11.5 Overview of ISIS

Applications that use Isis are organized into process groups. The membership of a process group can change dynamically, as new processes join the group or as existing members leave, either out of choice or because of a failure of some part of the system. A process can be a member of many process groups. Messages can be sent, or broadcast, from a process to a process group so that all the members receive a copy, without explicitly addressing the current membership. A process broadcasting a message can indicate that it wants to wait for replies from the recipients of the message. Process groups provide a convenient way of giving an abstract name to the service implemented by the membership of a group.

11.5.1 Fault-tolerant process groups

Isis provides location transparent communication with a process group and among its members. When a member of a group crashes, the Isis failure detection mechanism first completes all the pending broadcasts and then informs the group members of the failure.

11.5.2 Group broadcasts

Isis supports a set of broadcast primitives for sending messages to all members of a group. Depending on consistency requirements, the programmer may choose between atomic broadcast primitives that guarantee a global order on all messages and the more efficient casual broadcast protocol that guarantees only the delivery order of messages is consistent with causality.

11.5.3 Coordinator-cohort

Some actions need to be performed by a single process with the guarantee that the action is completed even if that process fails. For such computations, one member of a process group (the coordinator) is chosen to execute the action, while the other members are prepared to take over for the coordinator should it crash. This tool can be generalized so that a subset of the process group executes the action.

11.5.4 Tasks

A task mechanism is required for a process to be capable of controlling several sources of input and output simultaneously. A task mechanism allows several threads of control to exist within a single process. Isis provides a task system so, for example, if a task has broadcast a message and is waiting for the replies when a new message arrives, another task may be started to handle the new message even though the first task has not yet terminated. Non-Isis sources of input and output can be incorporated into the task mechanism.

11.5.5 Monitors

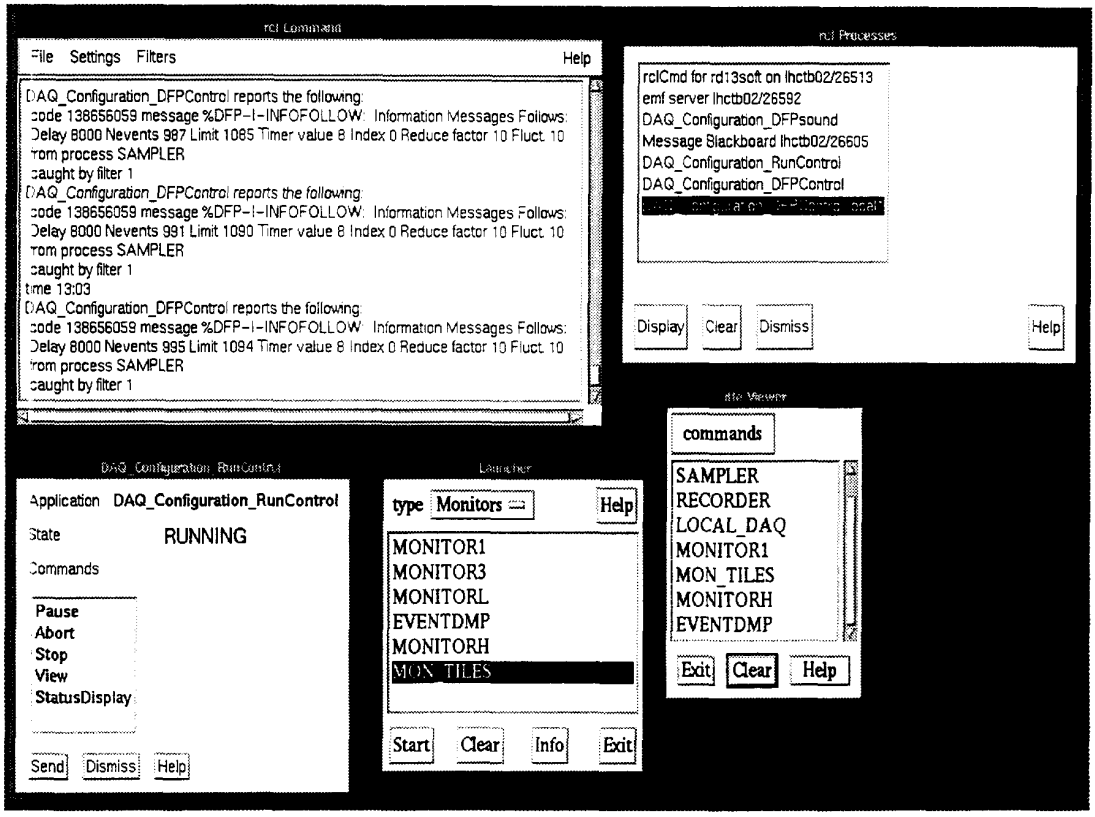
A process can instruct Isis to notify it when certain types of events occur. For example, a process can be notified if a new process joins a process group or if Isis fails on a particular machine.

11.6 Implementing the basic run control facility with ISIS

We have used Isis to implement the basic run control system. Isis is run on the network of workstations and frontend processors. The components of the data acquisition system are modelled as finite state automata which are controlled by a run-control program. The run-control program sends commands to the components in order to change their states. Starting from this model, we defined a set of process groups and message formats to support the manipulation of finite state automata. A library of routines, called *rcl*, was implemented to support the model and provide a framework in which component-specific code could be inserted. The library is linked with the data acquisition components causing them to join the predefined process groups and establish tasks that handle the messages. The library also provides a programming interface by which the run-control program can send commands to components and be informed of the result and their current state.

The run-control program is itself modelled as a finite state machine. To interact with the run-control program, we have developed a Motif based graphical user interface from which the user can send commands and interrogate any process using the library.

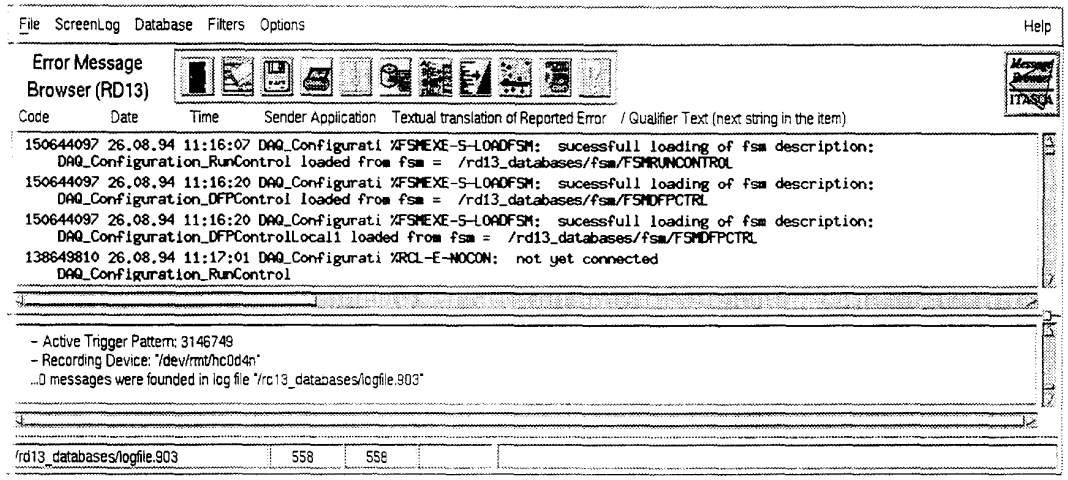
FIGURE 14 Run Control graphical user interface



11.7 Error message facility

We have also used Isis as the basis for an error message reporting facility. Data acquisition components report error conditions by calling a routine in the rcl library that broadcasts the condition as a message to a dedicated process group. The members of the process group are processes that wish to capture error reports. A filter mechanism, based on UNIX regular expressions, allows members of the group to capture subsets of all the errors reported. The graphical user interface is also a member of this process group and displays all reported error message to a window on the screen. Another process writes error reports with a time stamp and the identification of the sender to a log file. Filters can be down-loaded to data acquisition components in order to suppress error reporting at the source.

FIGURE 15 Message logging user interface showing messages sorted by date



12 User interfaces and information diffusion

Modern data acquisition systems are large and complex distributed systems that require sophisticated user interfaces to monitor and control them. An important choice to be made when designing graphical user interfaces (GUI) is that of the GUI toolkit to be used. GUI toolkits control output to the screen, accept input from the mouse and keyboard and provide a programming interface for application programmers. Traditionally programmers have coded the user-interfaces by hand but recently a new type of tool, called a GUI builder, has appeared on the market which allows the programmer to interactively develop user-interfaces.

12.1 Graphical interface toolkit

We considered toolkits based on the X11 protocol, the de-facto standard for distributed window systems. This choice was driven by a need to run client programs on remote machines and display the windows on a local workstation. A number of graphical interface toolkits exist but Motif (from Open Software Foundation) has, over the last couple of years, become the most popular GUI toolkit for UNIX workstations. The choice of a toolkit depends heavily on its programming interfaces. It must be possible to code an interface in a language which can coexist in the same environment as the other facets of an application, even if the code is to be generated by an GUI builder tool. Motif offers a set of general purpose graphical objects as well as a few specialized ones for performing tasks like file selection.

12.2 Parts not covered by Motif

During our work we have found a number of areas of interface design and implementation which we believe are not adequately addressed by the Motif toolkit. In all cases, it is possible for the developer to work around the problems by either implementing the

required functionality using the toolkit or by acquiring it from other sources, be it commercial or shareware.

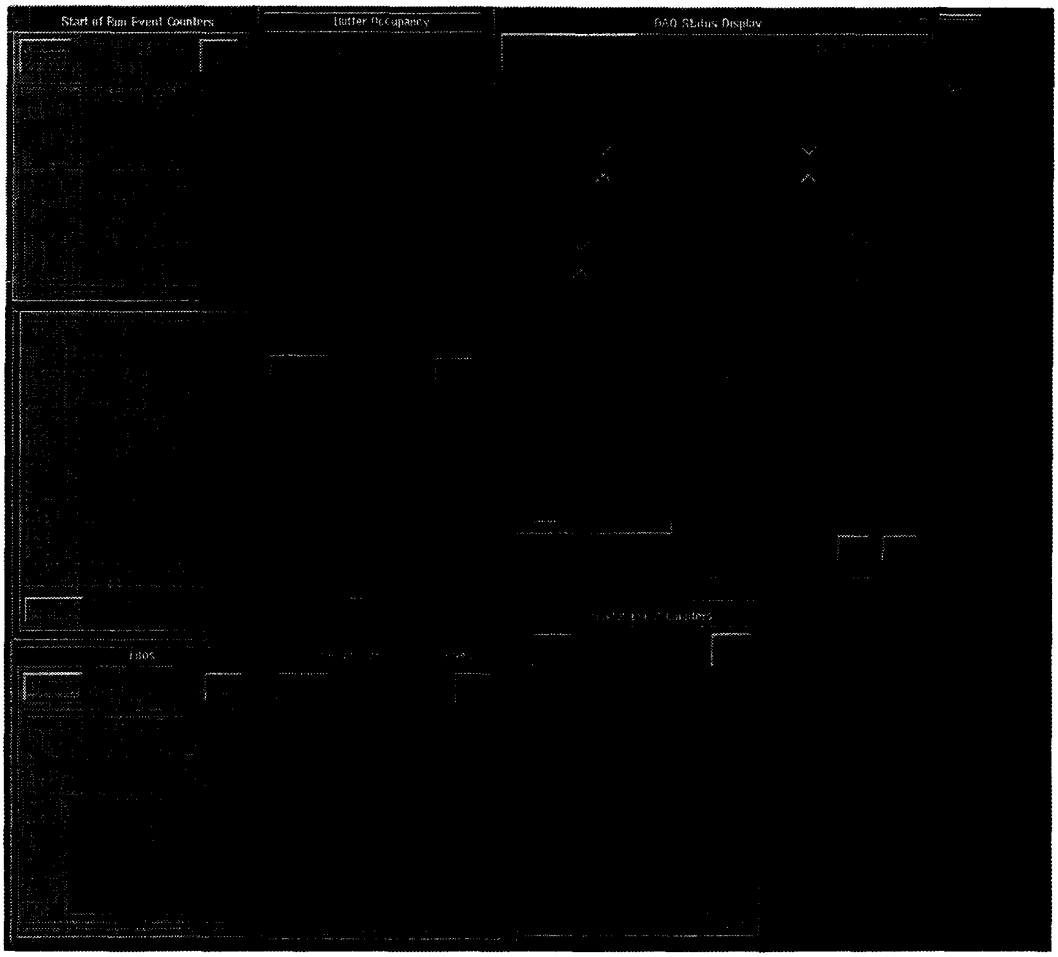
12.2.1 Visual representation of data

The Motif widget set includes many general purpose widgets, such as labels and buttons. It also contains a number of widgets, such as the file selection dialog, which address a specific function that is required by most applications. An area that also requires such specialized widgets is data visualization. Some applications need from additional widgets to present data in a table, as a pie chart or as a histogram. There are no specialized widgets in the Motif toolkit to meet these needs, so developers either have to build their own or obtain them from other sources. Many graphics packages offer such functionality and for DAQ interfaces that require more powerful graphics capabilities we used the DataViews commercial widget set (V.I. Corp). This widget set includes pie charts, histograms and various other 3 dimensional charts.

12.2.2 Updating information

Another limitation of the Motif widgets is the frequency at which the information they display can be updated. This can pose problems when a GUI is to present rapidly changing information. For example, the DAQ status display shows a counter of events treated by each data flow module, typically of the order of 1000 events per second. If we try to show the latest value of the counter every second with a Motif label widget it "flickers" so much that the figure becomes unreadable. Fortunately the DataViews widgets are capable of updating the information they display frequently (i.e. more than once per second) without such flickering.

FIGURE 16 DAQ status display built using XDesigner with Motif and DataViews widgets



12.2.3 On-line help

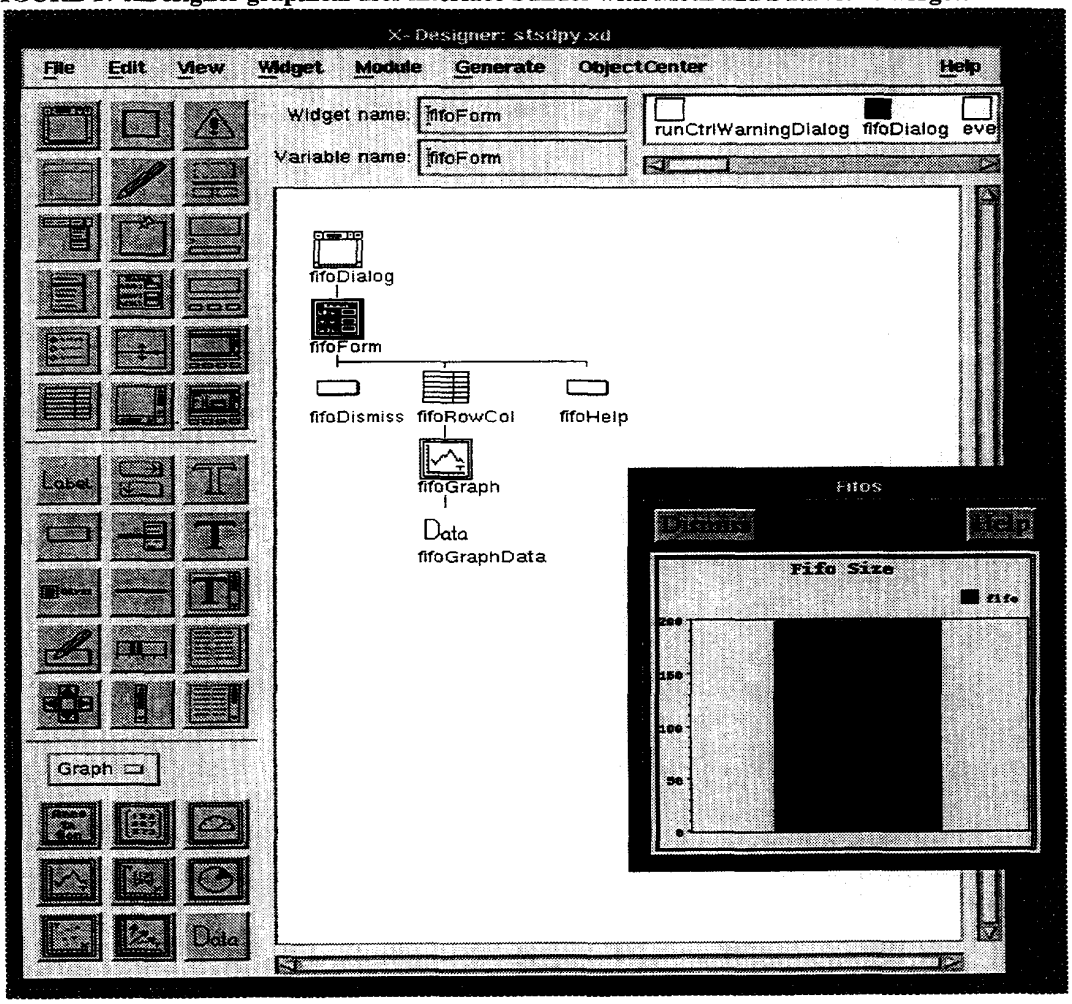
The Motif toolkit provides help buttons and help callbacks for most widgets. These mechanisms allow the interface developer to indicate help is available to the user and provide a way of signalling that such help has been requested. But the toolkit does not include a suitable means for displaying the help information itself. The developer is left to his own devices to implement a scheme using the existing widgets. It is better to provide information to the user as a combination of text and diagrams and so easy integration of text and graphics and the ability to navigate between documents would be useful. Hypertext systems offer the ability to cross-reference documents and navigate such references. For these reasons, we have looked beyond the Motif toolkit to find tools to help fulfil our goals.

12.3 Graphical user interface builder

Graphical User Interface builders allow the developer to interactively build interfaces. Using an interface builder the developer can implement an interface far more quickly than by hand coding calls to the toolkit. Interfaces can be built in a more abstract manner and hence the developer need not have such a deep understanding of the toolkit itself. Several builders, such as XVT[7], allow interfaces to be built which can run on top of more than one toolkit.

For the development of the DAQ we chose a commercial graphical user interface builder called XDesigner (from Imperial Software Technology) which has proved invaluable in aiding the production of user interfaces. X-Designer offers a means of incorporating 3rd party widgets (including the DataViews widgets) and allowing them to be used along side the standard Motif widget set.

FIGURE 17 XDesigner graphical user interface builder with Motif and DataViews widgets



12.4 Information diffusion

It is important to have a well organized documentation system for the DAQ from both a user and developer point of view, especially since the personnel that will work with an LHC detector will be renewed several times over the lifetime of the detector. We chose the commercial FrameMaker (Frame Corp, USA) interactive document preparation system as our primary authoring tool. FrameMaker is a Motif based WYSIWYG document editor available on many platforms including MacIntosh, IBM compatible PCs, NexT, VAX/ VMS and most UNIX machines. Documents can be transferred between different architectures by using a special FrameMaker Interchange Format (MIF). It offers support for input from other popular documentation systems and comprehensive graphics, table and equation features. To date we have used FrameMaker to produce:

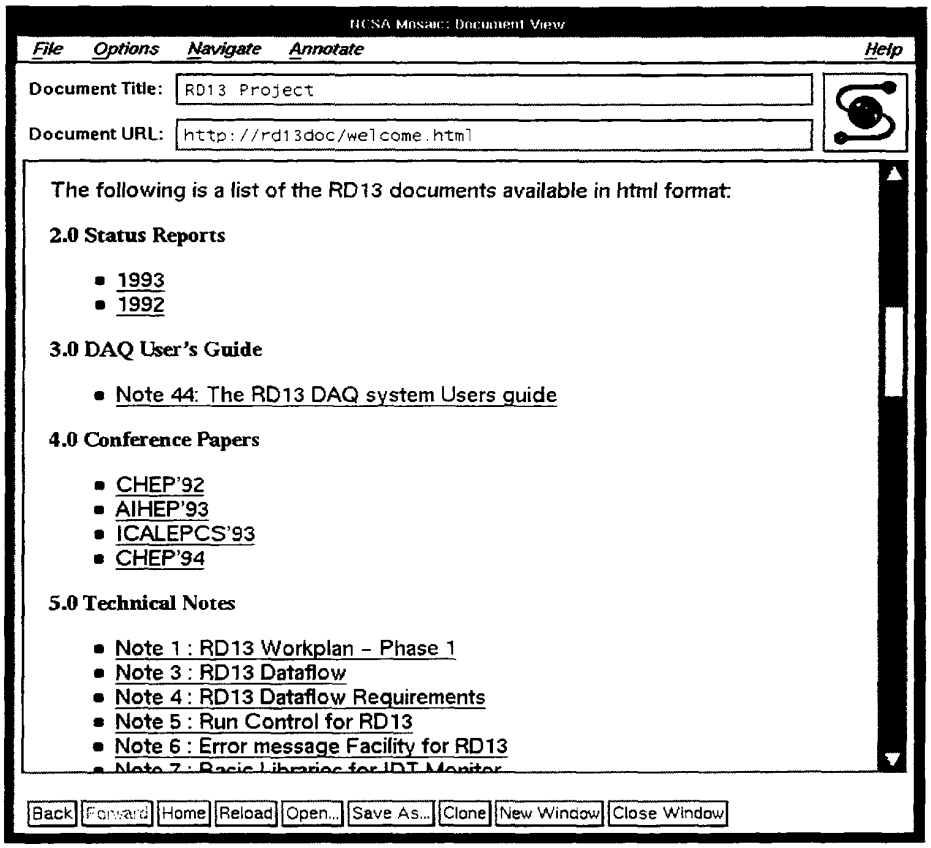
- Technical notes covering all aspects of the DAQ development
- Reports and Conference papers (including this document)
- Letters, faxes and slides

12 4.1 Making documentation available via WWW

In order to make these documents available to people outside the project we have setup a WorldWideWeb server for RD13 documentation. This required the conversion of the documents from FrameMaker's own format to HTML format suitable for use with WWW. Rather than performing the conversion by hand we have used a publicly available package that performs the conversion automatically.

We have not chosen to write our documents in HTML directly because, as can be seen from the list above, not all of the documents are destined for online viewing. Presentation on paper is very important, especially for Conference papers, and we don't know of any HTML interpreters which can give a good and totally controllable output on paper in the same manner as FrameMaker or LaTex. A LaTex to HTML converter exists but we prefer FrameMaker because it is a WYSIWYG system - with LaTex one has to print to postscript then view the output on paper or use a postscript previewer in order to see the results. Also a separate drawing tool (such as xfig) is needed to make figures while FrameMaker incorporates such a tool and other facilities (e.g. table and equation handler, spelling checker) that make writing documents simpler. Finally, it is easier and more efficient for people to only learn one tool rather than many and we have found FrameMaker addresses the needs of all the document types listed above.

FIGURE 18 RD13 WWW server welcome page



13 Software organization and distribution

In a software development of any size, especially when more than one developer is involved, it is important to have a disciplined approach to the organization of the various components and modules.

13.1 Directory tree

A directory tree to keep the code sources, include files and binaries for a **version** of the DAQ software provides a basis for a logical subdivision of the components. A version of the DAQ includes: DAQ components (e.g. the run control, the data flow modules), libraries (e.g. database access library, vme bus access library) and facilities (e.g. error messages). That is to say all what is needed to run the current version of the system and to develop software needing other system modules. The generic software components will be referred to, in the following, as a product. We distinguish between a

- **production** version: this is a tested, stable version of the system, including all software products. It is intended for "production" use. An update of the production version is done rarely (e.g. once every few months) and corresponds to a coordinated upgrade of the whole DAQ system.
- **validation** version: this is a version under test; unstable software components are brought together to provide the environment for the integration tests. Updates to the validation version are done on a product basis and relatively often (e.g. once every few weeks), in response to bug fixes or the need of new features by other products under development. The requirements for stability are here less stringent than in the production case.

A third kind of version, the **development** version, is related to individual products (the version of a product the developer is currently working on).

The scheme should cover both the backend workstation cluster and the frontend processors with access to the filesystem being guaranteed by NFS. The rationale for a common approach to storing sources and binaries is that in such a way there is a unique place where a version of application programs, libraries and include files can be found without (or with the minimal) interference with the developer's work.

A standard account should also contain the sources for the current production version of the software and a makefile to rebuild, from those sources, the product (even in the absence of the author). The account must be managed by one person, to whom a new version of a software product (for production or validation) will be given. A single access point to the modification of the contents of the standard directory tree guarantees proper notification of new versions and avoids the temptation of doing a "quick fix" to a module. The basic elements to appear in the directory structure are:

- sources

  all the source (code and includes) files necessary to build (and test, if available) a product. In addition to the sources a makefile, provided by the author, to rebuild the product from the sources. A sub-directory (below the sources directory) for each product can exist if the product is itself decomposed into component parts.

- bin

  the executable files associated to a product. Since the DAQ is to support multiple platforms a copy of the binary for each platform must be maintained. To do this end, the bin directory contains sub-directories - one for each platform - in which the corresponding binaries are kept.

- objects

  the object files (output of compilations of the source code), like the binaries must be kept for multiple platforms.

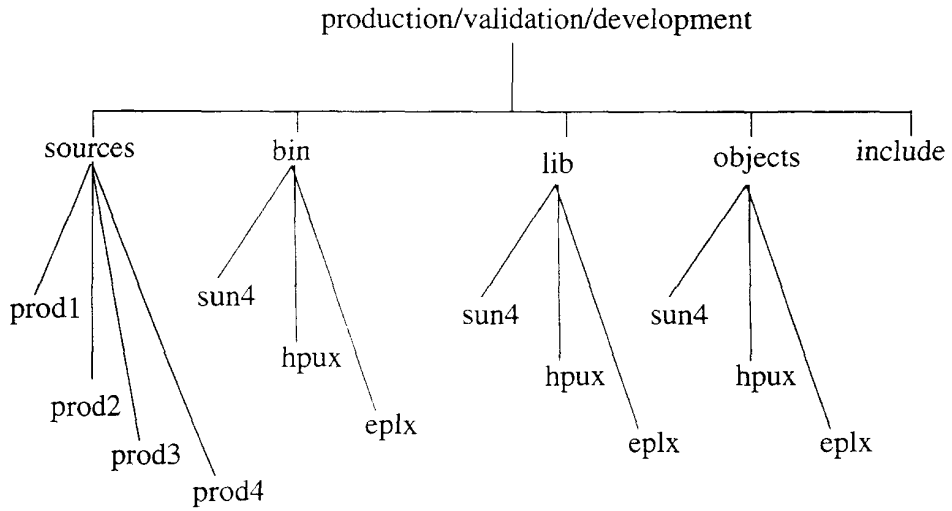
- lib

non executable, binary files associated to a product: archive libraries, object files, relocatable libraries or shared libraries. As for the binaries above, there will be copies for each platform.

- include

  the include files provided by the product to the users (e.g. error definitions, symbols, function prototypes, etc.), but not those needed only to build the product. Any platform dependent code is handled by conditional compilation flags.

FIGURE 19 DAQ software directory tree



13.2 Code repository

The management of source code, organized in the above directory tree, to track software modifications, releases and configurations is an important activity in a software project of any size. Originally the RD13 DAQ developers used the SCCS tool, which is normally available on any UNIX system, on a personal basis. The size of the developed software, as well as the interaction between different developers, is such that it was necessary to define a project-wide scheme for the maintenance of the source code.

While SCCS is probably satisfactory for a single user, a more sophisticated system was needed for the management of a complex software system made of different products and developed by several people. One such package is the Concurrent Version System, CVS. The source code of the different products is now organized into a repository managed by CVS. Each product is maintained in several releases and the full data acquisition system releases are superimposed on to the structure and are maintained by CVS.

CVS lies on top of RCS (similar to SCCS) which serializes file modifications by a strict locking mechanism and extends the RCS concept of directory or group of revisions to the much powerful concept of source repository. The latter consists of a hierarchical collection of directories or groups of revisions and related administrative files.

CVS offers a multi-developer open editing environment using basic conflict resolution algorithms. The most important features of CVS are:

- concurrent access and conflict-resolution algorithms to guarantee that source changes are not lost.
- support for tracking third-party vendor source distributions while maintaining the local modifications made to those sources.
- a flexible module database that provides a symbolic mapping of names to components of a larger software distribution.
- configurable logging support.
- a software release can be symbolically tagged and checkout out at any time based on that tag.
- a patch format file can be produced between two software releases.

13.3 DAQ builder

The DAQ builder is a shell script and a makefile developed for the management of the DAQ software components. Both the need of the integrator, who is the responsible of building a whole DAQ system, and of the individual developer, who is the responsible of the individual products, are taken into account. The tool is based on the directory organization and code repository described above.

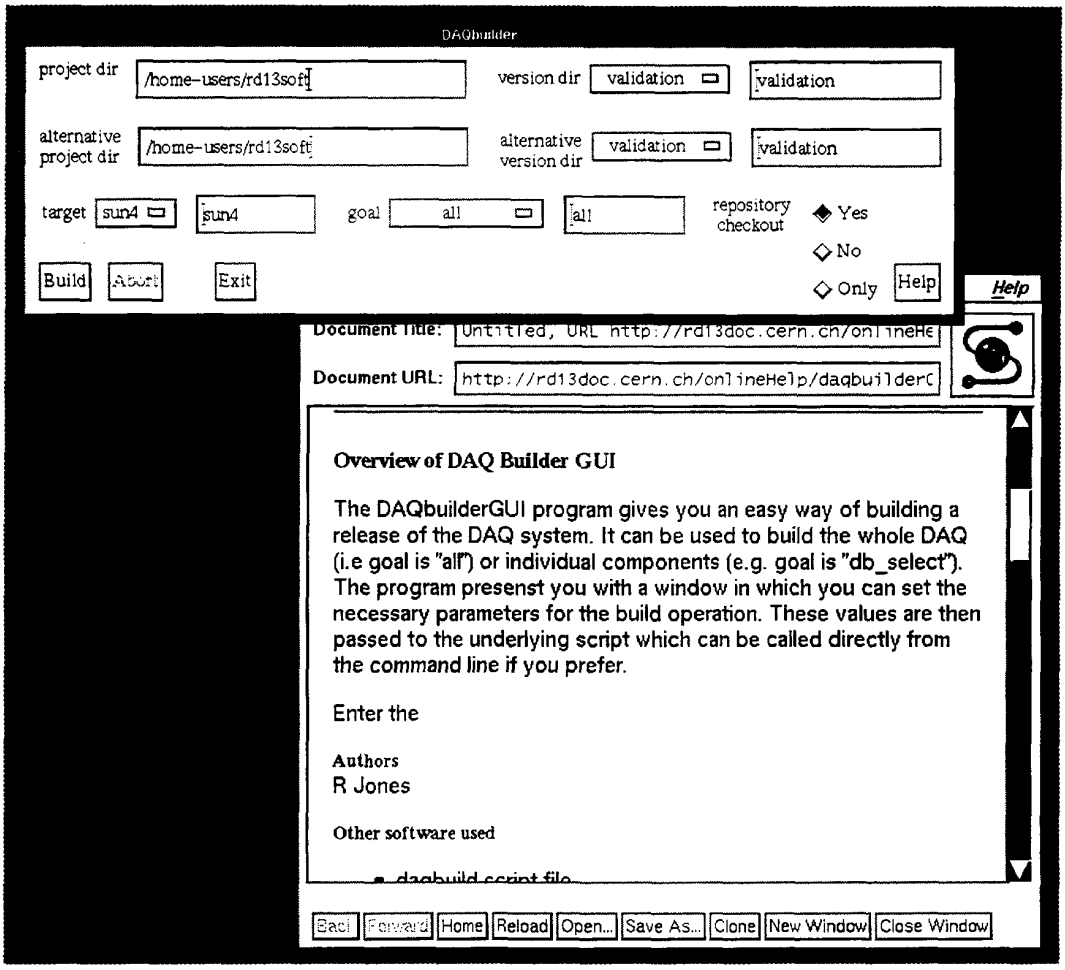
The tool is composed of three different elements (*daqbuilder, makefile* and *makerules)* which are maintained as separate revision files in the repository.

The *daqbuilder* is a shell script which defines and exports the environment variables globally needed by the software organization. It requires as input a root directory into which it will build, the name of the product to be built and the target platform. Having initialized the shell environment it then starts the make facility.

The *makefile* is a global makefile used by the *daqbuilder* shell script to invoke local makefiles which build any DAQ software component under the responsibility of the product developer. It is organized as a set of goals, one for each product. In addition the goal *checkout* extracts any software component from the common repository and the goal *dirs* creates the standard directory structure if doesn't exist.

The *makerules* file supplies global macros for compilers and their options and defines standard macros for common libraries.

FIGURE 20 DAQbuilder user interface with online help



14 Software quality

In order to continue to improve the quality of our software we must be concerned with the following issues:

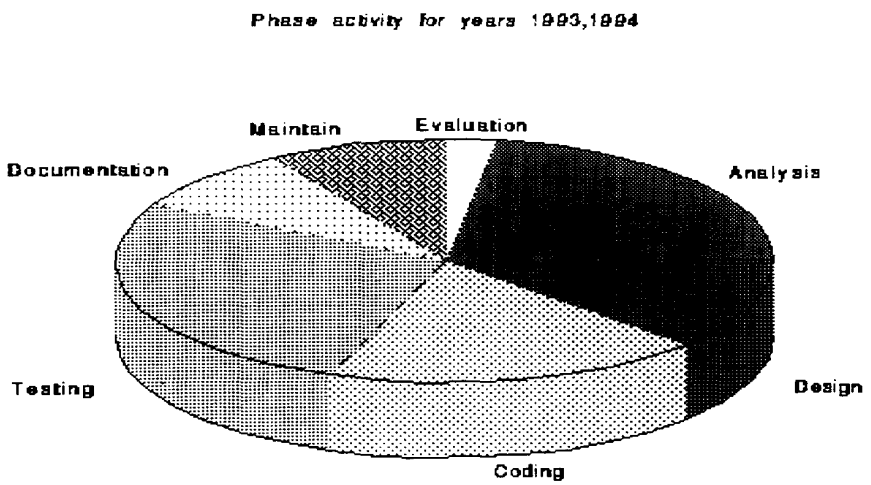
- Minimizing the number of defects in software.
- Creating mechanisms for controlling software development and maintenance so that schedules and costs are not impaired.
- Making certain that the software can be easily used and fulfils the necessary functions.
- Improving the quality of future releases.

We have adopted a number of steps that have a minimal impact on our day-to-day working but which help us to produce high quality software by addressing the above issues.

14.1 Measure the amount of effort used in software development

In order to determine the effects of any change to the way we produce software (e.g. use a new CASE tool or apply a new methodology) we need to know how much effort is put in to its production. Developers record their activity on a half-day basis (mornings and afternoons) using a calendar tool installed on the workstations. A script file, periodically run in batch, collects all the activity entries and totals the time used for each phase of each project. This information is then recorded in a spreadsheet (Microsoft Excel on a Macintosh) to perform some statistical analysis and output in graphical form.

FIGURE 21 Excel chart showing activity by phase



14.2 Record and track defects

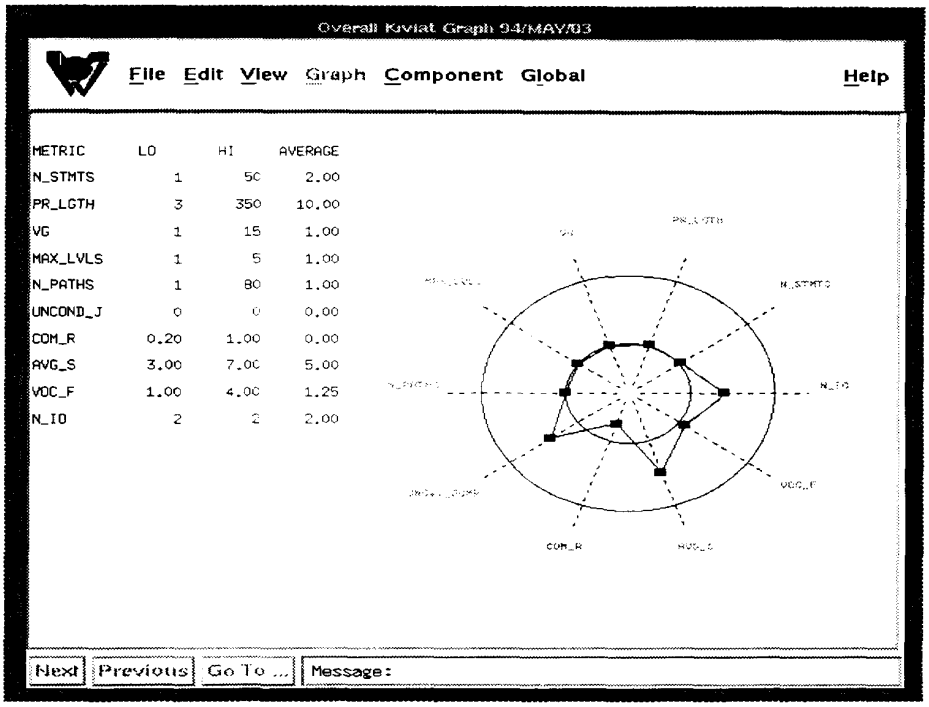
An important aspect of improving software quality is reducing the defects is the software produced. Before we can reduce the number of defects we must measure how many defects are detected. We have developed a simple database application, based on the QUID commercial database and a graphical user interface developed with X-Designer, which records defects. Details such as date found, project name (e.g. run-control), phase (e.g. design), defect category (e.g. serious or moderate) are recorded as well as the cause of the defect when determined. This information allows us to estimate the number of defects attributed to each activity of the development and indicates where we should concentrate our efforts so as to reap greater quality improvements.

14.3 Collect and record metrics

To measure the complexity of the software we have applied the McCabe metric. This metric gives a numerical value, on a per-routine basis, of source code. We have installed a public domain package that implements the McCabe metric and written some script files that give a size, in source code lines, of the software measured. As a second step we have

started to use the Logiscope commercial tool (Verilog, France) to provide more software metrics and an overall quality assessment in a graphical format.

FIGURE 22 Kiviat graph of software metrics applied to a C routine with Logiscope



14.3.1 Use metrics as an indication of the software quality

The above metrics can be applied to a version of the DAQ and the results used as input to the software production cycle. If software reviews are made then it is possible to use the results from the metrics as input to these meetings. That is to say that the review panel examines the metric values and only pass the software if these measurements fall within accepted levels. As the amount of data collected from these metrics increases, the developers will be in a position to cross check them against the defects recorded in the problem-database as a means of estimating the accuracy of the metrics themselves.

14.4 Testing software

As mentioned earlier, developers perform their own unit tests, then integration tests are performed when the various DAQ products are brought together. A number of tools and packages are available to the developer to make these test more rigorous.

- lint

This is the first tool to be used by the developer when testing the source code. It attempts to detect features of C program files that are likely to be bugs, to be non-portable, or to be wasteful. It also performs stricter type checking than does the C compiler. Use of this tool removes the simplest and most of common programming faults.

- Purify

  Purify is commercial tool for detecting run-time memory corruption errors and memory leaks. Purify intercepts every memory access by inserting instructions into the user's executable program before each load and store. These instructions make it possible for Purify to detect memory corruption just before it happens, even in code where source is not available. Purify also finds memory leaks, reading or writing beyond the bounds of an array; reading or writing freed memory, freeing memory multiple times, reading and using uninitialized memory, reading or writing through null pointers; overflowing the stack by recursive function calls, reading or writing to/from the first page of memory and so on. Such errors are often the cause of unexplained core dumps during the execution of a program and the use of Purify makes the detection of such problems far easier.

- Logiscope (metrics and code coverage)

  Logiscope has a dynamic facility that can be used to perform code coverage analysis. The DAQ source code is modified (instrumented) by Logiscope so that the paths taken through the code (e.g. routine entries and exits), branches and loops are recorded in a log file during execution. The tool then analyses the results and displays them in a graphical format to indicate which parts of the code have been exercised during the execution. Based on this information, the developer can produce new tests that exercise the sections of the code not previously covered. This facility is very useful when combined with the CVS repository since the source code can be checked-out, parsed to calculate the metrics values, instrumented, executed and the results analyzed from a batch job that can be run over-night so that the information is available to the developer the following morning.

- XRunner

  Another area of testing we intend to investigate is the exercising of GUI interfaces. Normally, the developer is required to test such interfaces by using the mouse and keyboard to select buttons and enter text whenever a new version is made. Commercial tools such as XRunner can record this activity and then play-back the mouse and keyboard strokes without human intervention and compare the results (i.e. contents of the screen) against a reference copy and signal any differences. This simulation of a testing sequence is appealing since it can be used to create a complete suite of tests for applications that can be used time and again.

15 Future trends

Having looked in detail at the various components of the DAQ system, we now turn to the future and try to indicate what is likely to change the most in the lead-up to LHC.

15.1 OODBMS

As mentioned in the databases section, Object-Oriented database management systems (OODBMS) are now becoming generally available. Not only can they be used to implement existing database functionality but also OODBMS promise to be better placed to address the issues of unstructured data, space requirements and performance which had imposed the development of ad-hoc solutions. Many OODBMS are capable of handling large unstructured data (e.g. images), provide transparency in case the database is physically on more than one device and support object caching and clustering

15.2 Object-Oriented methodology and CASE tool

Our experience has been very positive and we believe that there is a great future in software engineering and CASE technology for HEP, especially for large software production. The benefits in communications between the developer and the user, with provisions for project management and support for the full life-cycle including code reliability and maintainability are tremendous. The quality and quantity of commercial CASE tools is increasing rapidly and are sure to offer the most cost-effective and organized manner for HEP projects to develop their software. The software engineering field, and the CASE market in particular, are in rapid evolution and, although industry is investing effort in this direction, standardization is still far away.

The third generation of CASE tool, Object Management Workbench from Intellicorp, to be used in the RD13 project is just starting to show its worth. The tool supports the OO analysis and design method by J. Martin and M. Odell and provides a number of tools to fully support the software life cycle from analysis through to code generation and testing. OMW is built on top of the Kappa programming environment , an ANSI C-based visual environment for developing and delivering distributed applications in open systems. Kappa is based on a core engine with a complete ANSI C programmer's interface (API). Above the core sits a layer of graphical development tools, including a GUI builder and C interpreter. The next layer is class libraries, including classes for specific window systems (Motif or Microsoft Windows), and those from third parties. The developer's application classes and method code make up the final layer. Kappa applications can be distributed between UNIX workstations and PCs running Windows by using the Kappa CommManager which provides transparent communication among distributed objects running over TCP/IP networks and complies to the CORBA protocols defined by the Object Management Group.

CASE tools are currently used to implement individual parts (e.g user interfaces, database access etc.) of applications but it is hoped that, with such *Integrated*-CASE tools as OMW, it may be possible to develop completed applications.

15.3 A question of attitude

The above technologies will no doubt help in the drive to provide software for the LHC experiments. But the biggest advances can be made by changing our attitude towards software.

When the SSC experiment groups came to CERN to talk about possible participation in LHC experiments an important difference in costing was noted - European labs do not take into account the man-power costs when designing and building an experiment - this was very surprising to our American colleagues. This includes software development and makes Europe lean to a grow-your-own attitude to software since any man-power used to develop applications, packages and tools is essentially "free" whereas commercial software is visible on an experiment's budget-sheet.

An impressive level of professionalism is used by physicists in designing and building detectors. Panels of experts discuss and prioritize the requirements, alternative designs are made that take into account budget costing, civil engineering issues, energy consumption, integration problems, upgrade paths, radiation damage and so on (temperature effects, alignment corrections, toolings for mass production...). Sophisticated software packages simulate the expected performance of the various designs and the results are again discussed and compared. Only when all of these issues with their myriad of technical points have been reviewed several times and all the relevant parties (including the end-users) are in agreement, is the decision made on what detector will be developed.

It is a pity that such energy, interest and professionalism is not always carried through into the design and development of the associated software. It is often approached more as an after thought, almost as if it were a necessary but uninteresting last step akin to painting the outside of a new house before finally moving-in. All of the various techniques and steps used in the design and development of detectors are applicable to software as well, this is what the field of software engineering tries to define. This is the way LHC software will have to be developed if we want it work.

16 Acknowledgments

A presentation such as this obviously covers the work of many people and so I would like to thank all those who have participated to the RD13 DAQ and contributed to the incredible set of technical notes from which this paper is drawn. In particular, I would like to thank Giuseppe Mornacchi for the information on real-time UNIXs, the data flow protocol and database framework; Arash Khodabandeh for the Artifex and StP CASE tool usage; Pierre-Yves Duval for the extended run control system; Ralf Spiwoks for the simulation and event builder studies; Giovanna Ambrosini for the event formatting and monitoring skeleton; Giorgio Fumagalli for making sure it all glues together (i.e. the daqbuilder software production system and sacred code repository) and Livio Mapelli for keeping the project on a straight path.