Proposal to update the Milagro online system

Dec. 15, 1998
Miguel F. Morales

I have been concerned about our current Milagrito based online system and whether it can handle the complex online reconstructions needed for Milagro. Based on discussions with Gus Sinnis and David Williams I have formulated a proposal to change the current online system in order to attain the science goals for Milagro.

Many of us envision an online reconstruction system for Milagro that is both powerful and flexible. It must have the power to perform complex reconstructions that are tailored to the specific event, but flexible enough to quickly incorporate advances in our reconstruction code. Our current online system is simple and robust, but does not have the power or flexibility to perform complex event based reconstructions. We need to adopt a new way of organizing the online system so that it can become the sophisticated online system we need.

Let me use some diagrams to illustrate both the problem and the proposed solution. Figure 1 depicts the current online system. The current system takes every trigger and runs the event through the same set of reconstruction routines. The reconstruction is static - every event is treated in exactly the same way. Though adequate for Milagrito, for Milagro we want to perform the best possible reconstruction, not a reconstruction that is optimized for an average event. This necessitates having a system that can tailor the reconstruction to the event, making run time decisions on how to process each event. Figure 2 depicts the behavior of a sophisticated run time system. There are many different paths that can be chosen – each path is optimized for a different kind of event. In Figure 2 there are eleven different reconstructions that can be performed (count the "leaves" on the tree). This proposed system separates out different kinds of showers, and makes run time decisions on how an event should be processed. This type of run time system has the sophistication to deal with the complex data set presented by Milagro.

I am proposing that we adopt the conceptual model of the decision tree linked with the advantages of fully modular decision making. Note that Figure 2 is nothing more than a decision tree. This in itself is useful, but can be a nightmare to implement. In order to have the system be both easy to implement and flexible enough to incorporate advances in the reconstruction routines, we must also make the system fully modular. While discussed at length later, a fully modular system can make all run time decisions based on local information. Local decision making means that a new branching point or path can be added without affecting the existing code. This feature can greatly simplify updating the online system.

Using the combination of a decision tree model with fully modular decision making we can easily develop a very sophisticated and flexible run time system. In addition, because we already do the real reconstuction work in subroutines (angle fitter, core finder, etc.), implementing this proposal only requires minor revisions to the Milagrito Workers. I am proposing to change how the reconstruction routines are implemented by the Workers, not changing the reconstruction routines themselves.

In the next three sections I will discuss my proposal in detail. First is an overview of the decision tree structure and how to implement it so the system is fully modular. In the next section I discuss the details of how the system is actually implemented in the C programming language, and in the final section how the entire system should be managed.

Proposal Details

The main points of the proposal are as follows:

- Decision tree structure (with no recombination)
- Decision tree implemented in the Worker, not in the reconstruction routines
- Decisions are made by a new type of routine called a Branch Arbitrator, called by the workers

The key to this proposal is the decision tree structure. I have adopted this as the conceptual model to organize how we view the reconstruction process. The decision tree structure depicts each type of reconstruction as a “leaf”, or end point of the tree. For example, the online system depicted in Figure 2 depicts eleven ways of reconstructing an incoming event, depicted by the eleven leaves. The relationships between the eleven separate reconstructions are depicted by the branching structure of the decision tree.

The decision tree should not include explicit recombination so as to avoid ambiguity and to ease coding. A true decision tree can perform any reconstruction depicted by recombinatorial tree by simply duplicating portions of the decision tree. Notice that in Figure 2 large portions of the decision tree are in fact repeated. Though recombinations can make a decision tree more compact, they can also introduce fatal ambiguities. These ambiguities arise because how a shower should be processed often depends on its previous reconstruction history. Using recombinations erases this history information (was the ChiSq or MaxLikelihood fitter used?), forcing this history information to be stored in additional data structures and greatly complicating the code.

A true decision tree without recombination is much easier to implement in C (see Code Details) and avoids unwanted ambiguities.

Also central to this proposal is that all run control functions should be handled in the Worker, not in the individual reconstruction routines. This keeps all of the run control in one section of the code, and allows reconstruction routines to be easily reused. If the run control is handled in the reconstruction routines, minor changes to the run control could involve major code changes involving many of the subroutines. Keeping the run control in one place makes our code more modular and flexible.

I suggest that a new type of routine be added to the online code. All of the routines currently used in the Worker can be classified as Reconstruction Routines – the code that actually does the reconstruction. We need a new type of routine that does not do any reconstruction but instead decides how an event should be processed. I have discriminated between these two kinds of routines in Figure 2 by using a red circle for the Reconstruction Routines and a green box for the decision routines. I suggest that we call these new routines Branch Arbitrators because they determine which branch of the decision tree will be used for a specific event. The Branch Arbitrators will be subroutines that can be called by the Workers to make decisions on how an event should be processed. Please see below for a full description of these routines.

Code Details

In order to efficiently implement the decision tree architecture into the Workers we need two pieces: a way of converting the decision tree into C code, and a new type of subroutine that will decide which branch of the tree will process an event.

The easiest way to implement the decision tree architecture is through a series of nested switch statements. Every time the decision tree branches we need a switch statement with a case (or default) for each of the possible paths. The Branch Arbitrator then decides which branch of the switch statement will be used to process the event. A Branch Arbitrator is thus defined as a routine that can take any argument, but will return an integer value to the switch statement – the integer returned indicates which branch of the switch statement to use.

As an example I will write sample code that would implement the first branching of the decision tree depicted in Figure 2. First I need to write the Branch Arbitrator that will decide what kind of trigger accompanies the event, I will assume for demonstration purposes that this information is held in a integer called triggerCondition.

```
int TriggerBA(int *)                    /*definition of Branch Arbitrator TriggerBA( )*/

int TriggerBA( *triggerCondition)       /*Code for Branch Arbitrator TriggerBA( )*/
{
        if( *triggerCondition == CALIBRATION_TRIGGER)
            return(1);
        else if( *triggerCondition == 40TUBE_AS_TRIGGER)
            return(2);
        else if( *triggerCondition == SINGLE_HADRON_TRIGGER)
            return(3);
        else            /*Catches bad triggers*/
            return(0);
}
```

Now I will use the Branch Arbitrator that I have written to make a decision in the Worker. The branch structure is represented by the switch statement, with three possible paths as shown in Figure 2.

```
/*Code in Worker for the Trigger Branching point*/

switch( TriggerBA( &triggerCondition) )         /*infomation handed to appropriate Branch Arbitrator, */
        {                                       /*Branch Arbitrator returns an integer to indicate which */
                                                /*branch should be taken */
        case 1:
            CalibrationEvent( &rawData);        /*Passes event infomation to a routine that will handle */
                                                /* a calibration trigger (may call subroutines itself) */

        case 2:                         /*Starts processing event, can be explit as here or use a subroutine*/
            CalibrateRaw( &eventData, &rawData);        /*Calibrate event*/
            MuonSearch( &eventData, &muonData);         /*Searches for Muons*/
            switch( MuonBA( &muonData) )                /*Next Branch point,  note this switch is nested and*/
                    {                                   /*calls the next branch arbitrator  MuonBA() */
                    case 1:
                            .
                    case 2:
                            .                           /*Full code for shower recon is in these skipped steps*/
                            .
                            .
                    }

        case 3:                                         /*In original switch statement, case for Single Hadron*/
            SingleHadronEvent( &rawData);       /*Passes event to a single Hadron routine*/
        default:                                /*Catches bad triggers and other errors and calls a function */
            DiscardEvent();             /*that will release the FIFO memory and clear the event memory*/
        }
```

There are two important code specifics: use of switch statements to implement the decision tree, and creation of integer subroutines called Branch Arbitrators to make the decisions. For a large decision tree the switch architecture leads to a lot of code. However, this structure is very closely related to the decision tree and is easy to implement. In addition, because the decisions are made by the Branch Arbitrators and are inherently local, the switch statements can be hidden in subroutines. Notice that I called the subroutine CalibrateEvent() which includes further branchings as diagramed in Figure 2. This allows me to create subroutines that hold subsets of the decision tree to help organize the code. This ability to group the code in pieces is allowed because the switch/Branch Arbitrator structure is completely modular. Which brings us to flexibility.

One of the problems with Milagro will be code management. The run control structure will have to be modified frequently to take advantage of developments in the Milagro reconstruction code. With the switch/Branch Arbitrator structure it is easy to implement a change. To add a new branching point simply insert a switch statement and a new Branch Arbitrator in the appropriate position. Because the code is modular, all changes are local. No changes are required to either the Reconstruction Routines, or any of the existing switch/Branch Arbitrator pairs. Adding a new branching point takes a matter of minutes, not hours of rethinking if-then statements. Adding a new path to an existing branching point is even easier: just modify the Branch Arbitrator and add a case.

This ability to easily adapt the run control is the strongest argument for this proposal. Most of us involved in the code envision that the reconstruction should evolve significantly over the first couple of years of Milagro. The run control will have to change regularly to take advantage of developments in the Reconstruction Routines and deal efficiently with new science questions. Too often developments can be delayed or unimplemented because it is hard to work them into the online system.

This proposed system combines both the power of advanced run control and the flexibility of local decision making. Using this system we can develop the sophisticated reconstruction system that Milagro needs and easily adapt it as the system matures over the next couple of years.

People and Proposal Management

One of the difficulties with managing a complex system such as the online code has nothing to do with code, but rather with managing who gets to update the code and when. The trick is to find a balance between central control ("one person has all the power and work") and distributed control ("no one knows how the system works"). This was a big issue with Milagrito and will be even worse with this system because of its flexibility. We need to have a way of controlling how the system evolves without overworking some poor postdoc.

To address these issues I would like to propose that any changes to the online code must conform to a general set of guidelines.

- A memo outlining proposed changes and documenting their advantages must be submitted to the full collaboration
- The proposal must include a full decision tree with changes to the current code highlighted
- The proposal must also include the full code for all of the necessary Branch Arbitrators

Furthermore, we need some safeguards to prevent the online code from fragmenting. We do not want the added flexibility to inspire three competing versions of the online, each being updated by different people.

- A "Worker subgroup" of 2-3 people will be responsible for taking an approved proposal and updating the Workers

These guidelines serve three purposes. First, they allow anyone in the collaboration to understand the online system by reading the decision tree and accompanying memo. Instead of having the important details hidden in C code, this allows collaborators who do not program in C to understand how the online system works. The added input of the full collaboration should improve the quality of the online system.

Second, responsibility for most of the coding is given to the person proposing a change. All the important run control decisions will be made in the Branch Arbitrators, which are included in the proposal. This helps keep the people who administer the system day-to-day from being overwhelmed by coding jobs.

Third, it creates a Worker subgroup to oversee any updates and prevent new bugs from being added to the system. There should be a select group of people who actually implement changes to the online code. However, because the Branch Administrators are already written this will normally consist of adding a few switch statements and linking in the new Branch Administrators. This gives us oversight of the code updates without overburdening a few people with tons of code development.

We want to use our strength, which is the large number of bright people with individual interests. We need to involve everyone in developing the online reconstruction and analysis system in order to meet the physics goals of Milagro. However, we also need to manage our growth so that the system does not fragment or overload one person with too much work. These guidelines are intended to organize how the online code evolves. I think this code management structure, linked with the proposed changes to the run control system, make for a system that is powerful, flexible, and manageable.

Conclusion

I am proposing a change in the online reconstruction system that is conceptually radical but easy to implement in code. Linking the advantages of a fully modular design with a decision tree architecture will allow us to easily build a sophisticated run time system. The proposed system can make run time decisions to optimize the reconstruction for the individual event while preserving the flexibility to easily incorporate advances in the reconstruction routines.