

RTES Supercomputing 2003 Review

Author: Mark Fischler
Author: Marc Paterno
Revision: 1.8
Date: 2004-01-14

Table of Contents

- 1 Introduction
 - 1.1 Use Case 1
 - 1.2 Use Case 2
 - 1.3 Use Case 3
- 2 Major Concerns
 - 2.1 Organic Complexity
 - 2.2 Using **GME** in BTeV
 - 2.3 Using **Chameleon** in BTeV
 - 2.4 Logging Changes in the Trigger System
- 3 Physical Organization
- 4 Build and Release Environment
- 5 The Messaging System
- 6 Software Engineering Practices
 - 6.1 Circular Dependencies
 - 6.2 Idiosyncratic use of *do while*
 - 6.3 Non-use of the C++ Standard Library
 - 6.4 Non-use of Standard Protocols
 - 6.5 Use of *chm_assert*
 - 6.6 Multithreading Issues
 - 6.7 Release Procedures and Testing
- 7 Overall Comments
 - 7.1 Performance of the Demonstration System
 - 7.2 Communication Performance
 - 7.3 Documentation
 - 7.4 VLAs
- 8 Analysis of Code
 - 8.1 Forcing the User to Provide Boilerplate Code
 - 8.2 Concern About Thread Safety
 - 8.3 *ckpt_ct*
 - 8.4 *RTES message*

- 8.5 *element_ct*
- 8.6 The *Generator* Package
- 8.7 The *PA* Package
- 8.8 The Physicist's View of the VLA

9 GME

- 9.1 Who Won't use **GME** in BTeV
- 9.2 Who Might Use **GME** in **RTES** or BTeV
- 9.3 General Comments
 - 9.3.1 Support for Rapid Development
 - 9.3.2 Scalability
 - 9.3.3 Physical Coupling
 - 9.3.4 Transitory Nature of Generated Code
 - 9.3.5 User Interface to Resource Constraints
- 9.4 Versioning
- 9.5 Communication APIs
- 9.6 Configuration Maintenance
- 9.7 Comments on Existing Diagrams
- 9.8 State Machine

10 Conclusion

1 Introduction

We have reviewed two of the major software products from **RTES** used in the **SC2003** prototype project, with the goal of evaluating the applicability of these products for the use of BTeV. The two products we have reviewed are **Chameleon** and **GME**.

In this document, we comment on those issues we believe to be of greatest importance to BTeV. While we propose solutions for some problems, we have *not* attempted to provide solutions to all the problems we have identified.

In order to orient ourselves for this review, we considered a few of the roles in which BTeV authors would be acting. The most obvious were the following:

Element producer A BTeV author who makes a new *element*. Why would I produce a new element? I have data posted by my physics algorithm, for which I need to collection statistics. An *element* would be written to capture this information, to deal with it, and to put the output somewhere useful. An *element* could also perform changes to the system to alleviate problems.

Elements produced by BTeV users would typically have one of the following purposes:

- Accumulating statistics.
- Watching for "bad things" in the statistics.
- Making changes in the system.

VLA producer *VLA* means *very lightweight agent*. Like an element, but runs in the embedded world. Relatively few BTeV experts are likely to be writing VLAs.

VLA user Someone who has written a PA that needs to communicate with a VLA. Many more BTeV authors will use VLAs than write new VLAs.

These user categories aren't very intuitive for BTeV. So we chose to consider a few scenarios (use cases) describing concrete jobs to be done, and what they entail.

1.1 Use Case 1

I need to collect statistics for every event, for example the total number of hits in a system. I need to see this for every event that comes *into* level 1, not just those that survive the trigger process. What do I have to write?

- A physics algorithm that looks at event data, and calculates the number for that event, and which puts it into my local histogram. This software uses a VLA, because it communicates the histogram to the collection system by using a VLA.
- Somewhere, the 2500 histograms (one from each DSP) have to be collated. I would write an element to do this, to be run in a local ARMOR. The locals collate perhaps 200 different parts.
- Somewhere, the final collation has to be done, and the results get put into ROOT format. This would be done in another element, which is run in the regional/global ARMOR. This element posts the result to the database.

1.2 Use Case 2

I have to write the code that deals with a farm board temperature going out of range. What do I write?

- An element to go into the local ARMORs that watches the temperature. It reads the temperature once every n seconds – n is set by the configuration. Different board types may need different temperature thresholds, which also need to be set individually. The element can alert the operators via the control system.
- I write another element, run at the global level, which looks for a “temperature it too high message”, and does the right thing to gracefully shut down the board. This thing could also send a message to the operators.

1.3 Use Case 3

I have invented a new physics algorithm. Do I have to integrate with anything here? The end result is I don't care about the ARMOR system for this – the configuration is done outside of the configuration for the elements, etc.

It seems inconvenient for users to have to know about more than one configuration system – users will want there to be the One True Way, and this way should be chosen by BTeV, not by **RTES**.

2 Major Concerns

2.1 Organic Complexity

The system architecture is very rich and complex. Part of this complexity is inherent in what BTeV needs done, and perhaps part of it could be excised as implementing flexibility that BTeV would never under any circumstances make use of. Be that as it may, there are several interacting sorts of complexity:

- The multitude of elements and ARMORs.
- The layering of messages coming from one element, through routing elements, to others.
- The parent-offspring relationships.
- The notion that messages are shopped around and responded to by (frequently multiple) subscribers.
- The pervasive use of threads.
- The use of locks, mutexes, and related notions.
- The fact that all these activities are going to happen on a multitude of different processors.

A key point is that this is supposed to be a fault-tolerant architecture – that is the whole purpose of the project. That means that all this organic complexity is going to have to work smoothly in the presence of:

- Hardware which is assumed to occasionally crash and/or produce incorrect intermediate results.
- The assumption that some software modules may not be coded perfectly, and may either crash or produce misleading messages and/or data.

In our experience, any system of this level of complexity that functions gracefully in such a hostile environment does so for three reasons:

1. There is some overweening set of principles, scrupulously followed, which enables the designers to present logical proof that most or all of the potential unacceptable failure modes cannot occur.
2. The overall system and various intermediate levels of combinations of parts of the system have been subjected to vicious integration, mid-level, and unit testing, to demonstrate that the “logical proof” mentioned above really is effective. This testing requires careful and clever (diabolical would be better) planning and design. It is distinct from (and ideally totally separate from) the sort of diagnostic testing that allows you to shake the bugs out of the early code. We can call this set of tests the core system validation.
3. The coding of the core of the system is under a lot of pressure in that when the tests in (2) are performed, if the system fails in an unacceptable way, there is *no longer any convincing way to demonstrate* that the proofs going into (1) were valid. This means that the no-failure-mode concepts had better be valid, and that the coding and early testing of a key core has to be done as carefully as if a single failure in the core system validation would require abandoning the system and having it re-written by a different development team. (Obviously one would not really throw the whole thing out;’ the point is that to succeed, you have to design and code and pre-test as carefully as if you were going to risk that.)

A *necessary* part of the design of this system is the design and implementation of the sort of core system validation testing just described.

We are concerned that we were unable to find the sort of logical proof that most or all of the potential unacceptable failure modes cannot occur, which would allow some confidence that the complexity of the system won’t contribute to overall hard-to-explain, impossible-to-anticipate, unacceptable failure modes.

2.2 Using GME in BTeV

Use of **GME** is intended to be an integral part of the software development cycle. The reality of software development for BTeV is that many different groups will be developing (and need to test and release) separate modules.

We think that in order to be useful in the context of a large collaboration, there is a need to break up the model into sub-models. This implies some sort of way to define external references and dependencies and qualities that allow coupling, which is richer than just saying “module A depends on modules B and C.”

Furthermore, **GME** faces a serious sociological hurdle: many BTeV users do not use Windows, most do not prefer IDEs, and few currently use modeling tools. Some have had previous experience with other modeling tools, and most of these previous experiences have been failures.

2.3 Using Chameleon in BTeV

BTeV authors will need to invent, implement, and install new elements quickly during a crisis. The ability to use an alternative language such as [Python](#) would help reduce the learning curve and effort required to make and test a new element.

2.4 Logging Changes in the Trigger System

It is critical to deal with logging any changes in the trigger system in great detail, so that physics information isn’t lost. The example of changing prescales (while fictional) show this very well – if a prescale were changed, the system must record what the prescale was at every time, and when was it changed. Even if modification of prescales were not possible, *other* changes made by the system must be logged. There must be a standardized and easy way to do this, and to make the data available wherever it is needed – not just at the graphical UI.

3 Physical Organization

BTeV has a build/release system that their members will expect to use. **Chameleon** needs to be presented as a *product to be used*:

- headers to include
- libraries to link to
- executable tools to invoke

It must not be something that BTeV users are expected to add to. Thus the place that BTeV-written *element* source code will be put must conform to BTeV’s build system, and *not* be a subdirectory of some

Chameleon subdirectory. **Chameleon** must not require BTeV authors to modify or add to the **Chameleon** product (including header files and source files). It must not be necessary to recompile any part of **Chameleon** just because a BTeV user wants to create a new *element* or *message*.

Chameleon currently uses IMAKE as a build system. This is not the system with which BTeV authors will be familiar, but that does not matter as long as **Chameleon** is delivered as a product ready for use. The BTeV build system will place no more requirements on the physical design of **Chameleon** than those listed above.

We recommend that even internally **Chameleon** should distinguish between core infrastructure and utility elements. It should be possible to add new utility elements to the **Chameleon** product without needing to modify infrastructure headers.

4 Build and Release Environment

It is necessary to be able to perform *unit tests* on BTeV-written ARMOR elements. One needs to have a scaffolding to allow users to plug their elements into, and which will allow testing, *without* requiring the full system to do so.

The BTeV release system will require tagged versions of infrastructure code. Although **Chameleon** will not be under the control of the BTeV release system, it would be advantageous for BTeV (and for **RTES**) to have a more controlled release system. Code versions in CVS need to be tagged with meaningful tag names, so that one can refer to a specific version of **Chameleon** with assurance of the meaning of that version, and that it works properly with other **RTES** products.

BTeV's assurance that a new version of **Chameleon** meets their needs will be significantly enhanced by the presence of sufficient testing in **Chameleon** itself. Absent such testing, BTeV will be required to do so itself. This will at least lead to long delays in adoption of new versions of the code, and perhaps discourage BTeV from the use of **Chameleon**.

5 The Messaging System

The current **SC2003** design has a simplistic message handling system. This may be true of the core communication elements as well. It seems to be assumed that all the parts of a fully-deployed BTeV trigger system will communicate in a binary compatible manner. It is very likely that this will not be the case. The BTeV trigger system is likely to involve more than one variety of embedded processor; it is unlikely that BTeV would be willing to be restrained to use only those processors which are binary-compatible. In addition, it is likely that more than one variety of general-purpose processor will be used in the higher-level trigger systems. It is not guaranteed that these processors will be binary-compatible, and even less likely that they will be compatible with the embedded processors of Level 1.

It is crucial for the system to have a well-defined messaging protocol which deals with the problem of translating between binary formats where necessary. It is also crucial that this system not impose any burden in the embedded processor portion of the system, where the additional computational burden of translating messages between the processor's native format and some exchange format will be unacceptable.

It is also important that the details of dealing with different message formats (from different wire protocols, for example) not be placed in the laps of the BTeV collaborators who will be writing *element* subclasses. These details must be handled by the **RTES** infrastructure.

In the current design, the messages all share a single C++ type, *mc_message_ct*. Their *informational* type is determined by an integral parameter passed along with the message. The implementation of this design has a flaw that makes it unsuitable for use by BTeV. BTeV will need to invent new (informational) types fairly often. In the current design, this means inventing new defined constants, and recompiling **Chameleon**. It is unacceptable for BTeV to need to rebuild an external package every time a BTeV author invents a new message type. Furthermore, these numbers (in the current design), must be globally unique – it would be disastrous for the integer 5 to mean a different thing to a Level 1 node than it does to a Level 2 node. Thus splitting up the headers to allow different systems to have different defined constants does not solve the problem. BTeV will require a global message identification system, akin to the UUID of CORBA.

We also note that some sort of data verification (like a fast xor checksum) should be done on messages, even in the embedded processors, to help assure that bit errors are not propagated silently.

In the current design, the class *mc_message_ct* is what the BTeV user would see; he would create an instance of *mc_message_ct*, fill in its data, and send it along to the system. This class contains a vast amount of minutia that should be hidden from the user. We expect that the typical BTeV author will be overwhelmed by the complexity of the existing class. In addition, the part of the message that the user's element cares about, the payload, is dealt with entirely by user code. The degree of complexity left in the hands of the BTeV authors is too large. The interface presented to the BTeV user must be considerably simpler than *mc_message_ct*. For

example, BTeV will not be interested in using the key/value pair portion of the interface; if it is possible to remove them to simplify the interface, it would help BTeV.

It is not clear how nameservice works in ARMORs. Jim indicated that it may be through files in an NFS file system. This will need to be extended to be an actual robust nameserver.

Who assigns the *severity* of an error message? It shouldn't be the generator of the message – he doesn't have enough information to decide. How does *Category* and *Type* and *Priority* factor in this? Who fills in *Priority*? It should not be the sender. Can it be the receiver, or a middle-level ARMOR, which adds the information when it is available? There needs to be some statement of policy about error messages.

There is additional related discussion in Section [RTES message](#).

6 Software Engineering Practices

In this section, we have a somewhat random collection of observations about coding practices we observed in the **SC2003** prototype code.

6.1 Circular Dependencies

In order to support robust unit testing, it is important to avoid circular dependencies wherever possible. For example, the classes *element_ct*, *compound_ct*, and *armor_ct* are circularly coupled, so it is not possible to thoroughly test any one of these in isolation.

6.2 Idiosyncratic use of *do while*

In several of the C modules we looked at, we find the following construct:

```
do
{
  /* some bunch of work done here */
} while (0);
```

This loop is functionally the same as the linear flow of code:

```
/* some bunch of work done here */
```

However, when it spans over several screens, the *do while* construct misleads the reader to expect that some looping is actually to be done. We can see no performance advantage to this construct, and significant disadvantage to the ease of understanding.

6.3 Non-use of the C++ Standard Library

There is a widespread *lack* of use of the C++ Standard Library. C-style null terminated character arrays are used instead of *std::string*; fixed-size arrays are used instead of containers such as *std::vector*. We could find no use of the standard algorithms of the standard library. C-style IO is used in place of the less error-prone C++ IO.

Bare pointers are used everywhere, making memory management unclear and frequently unsafe (especially in regards to exception safety). Use of *std::auto_ptr*, or the appropriate smart pointers from [boost smart_ptr](#), would make the code both easier to understand and more robust.

The [boost](#) library provides other utilities which would be of benefit for **Chameleon**, and which BTeV will use in other places. The [boost thread](#) library, for example, provides mutex and threads which have been widely tested and which work on a wide variety of platforms.

6.4 Non-use of Standard Protocols

Where ever possible (certainly in the various Unix or Linux or Windows machines), communications should be handled by a robust communications library (BEEP, XDR-RPC, HTTP are examples), rather than by home-brewed solutions. The effort of **RTES** should be applied to the fault-tolerance related parts of the design, and not dissipated in reproducing tools which are already available.

6.5 Use of *chm_assert*

There is widespread use of *chm_assert*, which is surprising in a fault-tolerance system. For example, we find places where, if dynamic memory allocation fails, it seems that an assertion will be triggered. If this really results in runtime failure of the system, this is not acceptable for the BTeV trigger system.

6.6 Multithreading Issues

We did not have time to sufficiently investigate the locking system to evaluate the degree of thread safety in the system. The handling in *compound_ct* is rather complex. We note, for example, that RAII (resource acquisition is initialization) is *not* used to handle lock objects, and so it is hard to tell whether all obtained locks are released properly under all circumstances.

We are informed there were problems with deadlocks observed before the **SC2003** demonstration, and are therefore concerned with thread safety. This concern is heightened because the BTeV authors who will be writing elements will certainly *not* be experts in the writing of thread-safe code.

It is critical that the **Chameleon** system be reliably thread-safe, and that it be *very easy* for the BTeV authors to write element classes which do not damage the thread safety of the system.

6.7 Release Procedures and Testing

During the demonstration we ran into trouble because modifications had been made (to the ARMORs?), and the display software could not deal with this. Is it possible to go back to an old version? What testing is in place to allow a user to tell that a new component works with the system? Can we test without the graphical front end? Can the front end be tested without the full set of ARMORs behind it? What is the mantra for checking something in, to make sure broken components don't get into a release?

We did not see adequate unit tests for any of the software. We strongly believe that these are necessary, not only to get the initial system working but also to give confidence that the system still works after changes due to enhancements and refactoring.

7 Overall Comments

7.1 Performance of the Demonstration System

We were concerned that during the demonstration of the **SC2003** demo system, in what was supposed to be "nominal" operating configuration, the system was losing about 20% of the events handled by the system.

7.2 Communication Performance

If stats messages are to appear at a high rate, then communication paths must be established and the connection be long lived. Furthermore, the number of connections going from node to node must be minimized. An algorithm that creates and destroys connections for every transaction will perform poorly.

7.3 Documentation

Much of the software has few comments. Especially important is something in each header to describe the purpose of each class. BTeV users will need such documentation, so they understand the purpose of the classes.

7.4 VLAs

In the current system, the idea of "a VLA" is very amorphous. We recommend turning this into a clearly defined concept. The following questions need to be answered:

- What is the category of tasks to be handled by *VLAs*?
- What actions can be taken by a *VLA*?
- What actions are prohibited to a *VLA*?
- What are the constrained resources that may be used by a *VLA*? For each *VLA*, the definition of that *VLA* should address how much of each resource may be used.
- Should there be a "standard interface" for *VLAs*? Should there be points of similarity in some other sense? If not, there should be a convincing reason why not. If so, details should be presented.

8 Analysis of Code

8.1 Forcing the User to Provide Boilerplate Code

Code like the following is commonly repeated; similar code appears in most functions for most ARMOR elements:

```
dword dsp_monitor_ct::get_state (char *pachBuffer, dword *pcbBuffer)
{
    chm_assert (pachBuffer != NULL);
    chm_assert (pcbBuffer != NULL);

    // Checkpoint state in base element_ct class

    dword cb = 0;
    dword rc = element_ct::get_state (pachBuffer, &cb);
    chm_assert (rc == 0);
    chm_assert (cb > 0);

    // Checkpoint information specific to dsp_monitor_ct element

    char *pach = pachBuffer + cb;
    chm_assert (pach != NULL);

    *pcbBuffer = pach - pachBuffer;

    return 0;
}
```

In this example, two lines are really doing the work of the function, and the rest is "boilerplate". The burden on the user could be lessened by application of the **template method** pattern (from the Gang of Four). There are also other places in the code where this pattern would also make sense. The companion document show one of these other places.

8.2 Concern About Thread Safety

Throughout most of the code we have reviewed, we find non-thread-safe code that appears intended for use in a multithreaded environment. It might be useful in the modeling tools to indicate what functions are re-entrant and what functions are not. In the section on *Generator.c*, we note one such instance of trouble; more can be found elsewhere.

8.3 *ckpt_ct*

This class has public data. Why is pointer-to-array-of-char used, rather than *std::vector<char>*, for example, or anything else that is automatically managed?

One problem is that, by design, *ckpt_ct* will be copied incorrectly, and so anything that inherits from it will also not be copied correctly. If the intent is to make elements non-copyable, this should be done explicitly. If it is not (and we see no reason that elements must be non-copyable), then all the base classes of *element_ct* must be copyable. Furthermore, if copying is allowed through the base class, then cloning (polymorphic copying) must be supported.

What is the intent of the member functions of *ckpt_ct*? For example, we find both *get_raw_state* and *get_state*; they take arguments of different types. The default implementation ignores the arguments and returns zero. Presumably, for an element with no state, this is the correct behavior. (If not, then these functions should be pure virtual.) For an element with state, what is the difference in intent?

8.4 *RTES message*

It does not seem that the information in the **RTES** message class is sufficient for BTeV's use. We propose the logical content of an **RTES** message be the following:

- an integer ID indicating the C++ data type of the data payload. This number must be a unique identifier, automatically generated. There is probably an ID somewhere in **Chameleon** to which this corresponds, but we are not sure which it is.
- an integer (NQ) telling the number of following *qualities*.

- a sequence (of length NQ), containing integer IDs for various qualities assigned to the payload. These quality IDs need to be centrally registered.
- an integer (ND) telling the number of bytes in the data payload.
- a sequence of bytes (of length ND), the data payload itself.

We have found this to be necessary in other cases, because we have found the need for things like *elements* that respond to every "error" type message, and other things that respond to every "silicon" type message. The categories are not mutually exclusive.

Of course, the sequence of integers does not have to be part of the physical message; it can be encoded in a single integer. The choice of implementation is a matter of implementation efficiency.

8.5 *element_ct*

What is the purpose of the public data members, some of which also have accessor methods? Public data members are generally a mistake, unless the class is a clear value-type, e.g. *std::pair<X, Y>*.

What is the meaning of *pcmpParent*, which seems only to be used to get an *id*, if *pcmpParent* is non-null? Who needs to get at the pointed-to *compound_ct*? Similarly for *parmor* and the pointed-to *armor_ct*? Why are some variables named with the (unattractive) Hungarian notation (e.g. *pcmpParent*) while other are not (e.g. *id*, *et*)?

We note that each *element* in the system contains a factory function, and that these are all of very similar nature. It would be convenient for BTeV if they were provided with a macro which would write this function. In addition to making the writing of an *element* easier, this would also make for easier maintenance. Without such a facility, any change in the design of the factory function would require modification of every *element* class in existence.

The factory mechanism has another significant omission. Users will want to be able to query the system for a list of all *elements* which are available to a program. The existing factory mechanism does not allow this. The common solution is to have a global registry of all *elements* known to the system, automatically populated at program startup by the construction of objects of static lifetime. Each *element* would be required to have a constructor with a given signature, and would also invoke a library-provided macro which deals with the issues of registration. It is useful for this macro to also automatically handle tracking of version numbers. Several examples of systems using this sort of solution can be provided on request.

The suggested look for each *element* is:

```
class some_element_ct {
public:
    // Every element class must take a ConfigParams object.
    explicit some_element_ct(const ConfigParams& c);

    // ... whatever virtual functions of element need to be
    // overridden,
    // ... whatever additional functions are needed by
    // some_element_ct
};

// CVS will replace $Name with the CVS tag for this
// release.
REGISTRY_MACRO(some_element_ct, "$Name");
```

8.6 The *Generator* Package

We considered this package as an exemplar of all the packages developed for the embedded processors, with the expectation that a critique of this one package will identify practices followed elsewhere. This package consists of the files *Generator.c*, *boxmuller.c*, *mt19337-1.c* and *poisson.c*.

We find the presentation of interfaces for this module chaotic. Some of the function prototypes are found in *generator.h* (note the difference in capitalization), but not all the function interfaces are there (for example, *poisson* is missing). Some of the function prototypes are repeated in *Generator.c*, and the header *generator.h* is also included. Other source code modules (for example, *poisson.c*) do not include the modules header. This needs to be reorganized to meet with standard coding guidelines.

The file *Generator.c* contains some magic numbers:

```
// defines of input and output port numbers
// THESE NEED TO CHANGED IF THE MODEL CHANGES!
#define SW_OUT 0
#define PARAMS_PORT 0
```

The note indicates that these values are not kept in agreement with the model by any automatic mechanism. This will surely lead to lack of agreement. We find no tests that indicate the values have been verified. The magic numbers must either come from a central repository, or from automatically generated code.

There is widespread use of global variables; *Generator.c* contains two blocks of them. Since they are not *static*, they are visible to the entire program during linking. They are likely to lead to multiply defined symbols. Even worse, these are global variables introduced in a *thread function*. If multiple threads are running this same thread function, we have little faith that complicated modules programmed in this manner will function correctly. Even something as simple as the Box-Muller implementation is not thread safe.

The file *Generator.c* also contains conditional compilation, hiding the fact that *Generator* is logically two different functions:

```
#ifndef USE_DSP_BIOS
    while(1)
    {
#endif

    /* a large inlined function body */

#ifdef USE_DSP_BIOS
    TSK_sleep(sleep_time);
}
#endif
```

This organization hides the fact that there are two different programming models supported in this one function. What happens if one needs to use the function in *both* models? What if there is a third programming model to be added (for example, Windows or Linux for testing, vxWorks or OSE kernel for running in the embedded processors)? How is the function to be tested?

We would prefer to see such code written in a layered approach:

- an explicit function, doing the equivalent of the "inner function"
- a separate function that can be called as the "thread function," perhaps a different one for each supported system.

We comment elsewhere on the idiosyncratic "do {...} while(0)" loop.

The cut-and-paste method of programming in *Generator.c* has led to code with is both a maintenance burden (it is harder to read than necessary) and inefficient (the same function, *get_input_buffer*, is called repeatedly). The inefficiency is especially important when we consider the time-critical nature of the code which runs in the embedded processors.

The function *MakeOneEvent* contains a mixture of the implementation of the function itself, interleaved (via conditional compilation) with what seems to be testing code.

8.7 The *PA* Package

We understand that the organization of *pa.c* was influenced by the fact it was a throw-away development, for use in the **SC2003** prototype. However, we see nothing in *pa.c* that indicates the design for a more suitable implementation is at hand. BTeV will need a framework that hides most of the complexity of the non-physics-related code, so that they can concentrate on the physics tasks. They will expect to see a much higher-level API, rather than the low-level details of interactions with VLAs, timers, etc. Jim tells us that David Berg's Level 1 framework document would be a reasonable starting point.

There is a small amount of such a framework in place; we note that *start_vla_timer* and *reset_vla_timer* are called from the function *PA*. However, this is much too low-level an interface, even for a framework. We would prefer to see a framework which uses functions like *start_process_event* and *end_process_event*, where the *start_process_event* does whatever is necessary – starting a VLA timer, and whatever other actions are needed. We have found this to be a useful abstraction in many other contexts; the sequences of actions to be taken often need to be configurable.

8.8 The Physicist's View of the VLA

Above, we noted that the interface chosen for physicist's interaction with the VLA is not at the right level of abstraction. In the current design, physics applications are littered with the details of state handling. We would prefer for this level of detail not to be in the face of the physicist programmer.

The proposed interface for the physics application to follow imposes insufficient structure. The developer must be told in a clear manner:

- What situations may at one time or another be declared (by the system) for this module to react to (e.g. `process_an_event`, you timed out, things were reconfigured, initialize).
- It must be made obvious that the module is required to provide a function to react to each such possible declaration. Some implementations may be "do nothing."

This is very similar in concept to the creation of a C++ class which implements a specific abstract interface.

This structure should be enforced either by the model/development tool, or by the runtime system. And, of course, unit tests should verify that all necessary functions have been supplied.

The implementation of `start_vla_timer` and `reset_vla_timer` both contain hard-coded constants, giving a timeout value. Such a value *must* be a configuration parameter, to be set at runtime.

9 GME

9.1 Who Won't use GME in BTeV

Some BTeV developers simply are not target users of **GME**. Features that would make **GME** unsuitable for these users are not "fair game" for this review to critique as a flaw; features which would help only these (non-)customers should not be considered as desired capabilities. Here are some non-target user communities:

GME is a Windows-only tool. For much of BTeV, this makes it unattractive. Most BTeV users will not be interested in using anything other than their "normal" Linux systems. **GME** will not be used by BTeV physicists doing analysis; it does not fit the "business model." It will also not be used to configure BTeV Monte Carlo jobs which may well run on the trigger farm during pauses in data collection, again because it does not fit the expectations of the BTeV community.

We expect that the Level 2 and Level 3 trigger executables will use the offline framework. This framework already has a configuration mechanism. If **GME** is to be used to configure these executables, it must be able to use the "standard" BTeV configuration mechanism as a lower-level tool.

Developers of physics algorithms for the BTeV trigger will want to work in their normal environment, and we do not want to put additional stumbling blocks in their way.

9.2 Who Might Use GME in RTES or BTeV

We can identify the following categories of potential users of **GME**:

- The BTeV developers and maintainers of **Chameleon** elements, VLAs, and any other source code that contributes to state machine behaviors in BTeV. These people develop code to run the trigger and detect faults, as opposed to code to *do physics*.
- BTeV members assembling and building the executables for the trigger system. There are several types of executables. These include ARMOR executables, trigger and filter/reconstruction executables, VLAs and Level 1 algorithms.
- BTeV members designing runtime configurations for existing executables. We assume that *using* the runtime configurations is done outside the context of **GME**. Some of these configurations may include: physics parameters used to configure the wide variety of physics algorithms; communication links between various executables; locations at which executables will run; and choice of fault-tolerance policies.

Considerations which make **GME** less appropriate for these types of use are "fair game," though in the end it may be decided that **GME** will not be used for one or more of these purposes.

9.3 General Comments

9.3.1 Support for Rapid Development

It must be possible for BTeV shifters in the control room to be able to invent and quickly deploy new diagnostic software in emergency situations. They must be able to do so with minimal impact on the running system. It

must not be necessary to stop the running system to deploy new diagnostics. It is unclear to us how **GME** helps in this situation. **RTES** should provide several use cases for BTeV that show how such situations can be handled using **GME**. The tool must not cause an entire rebuild of a release due to one or a few files being changed.

9.3.2 Scalability

In looking at the overall software aspect in the model "system," we find the diagram includes two buffer managers. In the real system, there will be several hundred buffer managers. The existing diagram is not scalable to handle this many elements. An diagram element that represents a collection of similar elements, and their individual connectivities, is needed. The code generation must automatically handle configuration of multiple elements – the user should *not* need to separately configure hundreds of nearly identical elements.

When a small change is made in the model (changing a single configuration parameter, for example), does this require regeneration of all interpreter output, or does this produce only the output directly related to the modified input? Generation of the entire output would be unacceptable.

If a model parameter that is a runtime value (such as a physics algorithm threshold) is changed, is any new code generation done? Must any code be recompiled? Must any binary be rebuilt? Any of these results would be unacceptable for BTeV.

9.3.3 Physical Coupling

GME must not generate monolithic files that create a huge amount of physical coupling. For this reason, we do not think modeling the messages with **GME** would be necessary. Certainly assembling all the IDs into a single header file would be inappropriate.

Code generated by the tool should meet the same code quality requirements placed on real developers. For example, it must not generate any unnecessary physical coupling.

9.3.4 Transitory Nature of Generated Code

We do not like the placement of generated code into the repository. We believe that generated code is a temporary artifact of the build system. The compiled results should be used to build libraries and executables, but the generated code itself is disposable.

During the kickoff meeting for this review, it was mentioned that some of the generated code required hand modification. What feature of **GME** made it inconvenient to produce the code directly using the tool? It would be unacceptable for BTeV users to need to do the same.

9.3.5 User Interface to Resource Constraints

In the **GME** IDE, we found that many of the attributes associated with some of the elements conveyed little meaning (and did not seem to be used). For example, in the detail box for the software primitive *Generator.c*, we find "object code size," "dynamic memory size," and "execution time." While these are interesting quantities, and important in an environment with significant resource constraints (e.g., limited size of dynamic memory and time budget in a low level trigger), we observe that in the current system they are not being filled. Indeed, we cannot see, in the current system, any way to fill them with meaningful data. The system should provide ways to deal with some of these automatically, and tools which can be incorporated to deal with others.

There should be some mechanism to calculate and summarize the usage of constrained resources, and to report on their relations *vis a vis* global total constraints. Items that are constraints in this category should be clearly labeled, and routinely checked by the model when the interpreter step is done. Items that are calculable automatically should be calculated.

9.4 Versioning

Many versions of the model could correspond to a single production version of the BTeV software. Different developers will need to work with different versions of the model without colliding with each other or with the production release. BTeV will need to have several production versions in use at one time, and also an arbitrarily large number of development versions. We have more to say in Section [State Machine](#) about the versioning of code related to the state machine.

The tool must distinguish between a "certified release" and "certified runtime parameters" and allow them to be manipulated separately.

9.5 Communication APIs

Part of the purpose of a code generator (interpreter) for **GME** is to impose a specific protocol for the coupling of software modules. The current code generator model (the task model) is a simple, low-level protocol, unsuitable for use at BTeV.

It is critical for the interpreters to be easy to write and extend, because the needs of BTeV will not be clear early on in the development cycle. It is nearly certain that several communications APIs will be devised, each with its own realm of applicability.

9.6 Configuration Maintenance

In what format is the data saved from **GME**? Is there any way to have it automatically processed, outside the graphical tool? Is it stored in some database? Can it be queried?

If **GME** is to be used to configure all the software for the running trigger system, it will be necessary to be able to capture a configuration state for use outside of the tool, and for the configuration data to be stored in the standard BTeV production databases.

Is the configuration *really* all in code? If one wants to change a some numerical value (the number of nodes above a threshold temperature before declaring a fire emergency), does new code have to be generated? This would not be acceptable for the trigger application.

Does *reconfiguring the system* always entail *recompiling* code? Is this to happen in the control room, when the users (people running a shift) just want to change a trigger configuration? What if one wants to start a new run, which is allowed to use more processors than a previous run? What if one wants to add new data output streams? What if one wants to change the level of informational output, to further study the reason for a misbehavior? What if one is done with a study, and wants to make the system quiet again?

9.7 Comments on Existing Diagrams

In existing diagrams, we find that most of the labels of diagram elements are not descriptive. This makes it very difficult to understand the diagrams. While we understand that the labels on the diagram must be terse, there must be a way to associate a longer, more descriptive name with each diagram element. In the example diagrams, this feature should be used.

We find that the existing diagrams do not reflect a consistent decomposition of systems. Some things are aggregated that are really distinct elements (e.g. "generator/switch"), and some things are not aggregated that should be (e.g. "buffer manager," "worker," "farmlet," which really form a farmlet).

We are concerned that the meta-modeling is so hard to do that even the experts have a hard time in producing a natural modeling language. The modeling language developed for the prototype system was so difficult to use properly that the developers themselves failed to devise a natural decomposition of the example system. It appears that a first pass at decomposition was put into the diagram, and later when it became apparent that the decomposition was not quite right, the effort that would have been needed to perturb it was too great and so it was left alone. Experimenters at the lab are even less like to correct initial flaws if the system makes changes awkward. If the natural use isn't easy, BTeV developers are likely to produce very confusing models.

9.8 State Machine

How does one deal with supporting several different architectures? It is possible that more than one type of embedded processor would be in the running system. Since the user of **GME** is writing snippets of C code in the context of running **GME**, does this mean that it is possible that different versions of the snippets have to be written for different processors?

The state machine as currently modeled relies exclusively on *guards*, and makes no use of *event types*. This is bad for a variety of reasons:

1. One of the advantages of deterministic finite state machines is the ease of proving them correct. The lack of distinct event types makes these state machines into something other than DFSMs, and thus this advantage is lost.
2. If two guards may be passed, the system could end up in an indeterminate state – or rather, one determined by the order of the code rather than by the guards themselves. The actions for more than one state may be taken. The state *actually* reached is determined by the last snippet of code to be run. This seems error prone.
3. It is inefficient, because each output state must execute its guard to decide if it is the result of the transition. This seems to be the case even after the correct "transition" has

been handled. The scaling behavior is linear in the number of states to which a given state may transition, which may lead to poor performance in a time-critical system.

The code implementing these guards is strewn with casts, magic variables, and magic numbers. The existing examples are hard to read, and would not be acceptable by common coding standards.

The code implementing these guards is not under any apparent code versioning control. This code has critical effects on the behavior of the system, and so it must be under version control.

The user of **GME** has to type fragments of complex C code into a small editing box, with poor editing facilities. This code is then stitched together into the final code. This is a hard way to write code.

We would prefer to see the user deal with a higher-level, abstract view of a DFSM, and to have the generated code hidden from the user. We would like the tool to automatically verify the transition table. This view of the state transition table should be saved in CVS. The action code should be in named functions, which are themselves stored in CVS. Of course, this code should be accompanied by unit tests. This would allow version tracking of both the transition table and the actions, and thus of the functioning of the entire DFSM. The issue of what *version* of a named function is associated with a specific version of the model needs to be solved.

10 Conclusion

We congratulate the authors on the successful deployment of working system to the **SC2003** workshop.

In this document, we have raised the most significant concerns which we had while reviewing the **SC2003** product. We invite the authors to contact us for clarification if any of the issues we have raised in this document are unclear.