# EMMA: a new paradigm in configurable software

# EMMA: a new paradigm in configurable software

**J M Nogiec and K Trombly-Freytag**

Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

E-mail: nogiec@fnal.gov

**Abstract**. EMMA is a framework designed to create a family of configurable software systems, with emphasis on extensibility and flexibility. It is based on a loosely coupled, event driven architecture. The EMMA framework has been built upon the premise of composing software systems from independent components. It opens up opportunities for reuse of components and their functionality and composing them together in many different ways. It provides the developer of test and measurement applications with a lightweight alternative to microservices, while sharing their various advantages, including composability, loose coupling, encapsulation, and reuse.

## 1. Introduction
The main objective of the EMMA framework was to define an architecture and provide common functionality for a family of measurement systems, thereby ensuring unification of design and a high level of software reuse. This approach results in standardization of many aspects of the developed systems, including internal communication, configuration, error handling, logging, data archiving, and user interface (UI) look and feel.

Measurement systems are developed in an iterative process, whereby algorithms are improved, and various settings and parameters are adjusted and tuned until the system performs satisfactorily. This process requires a software system that allows for quick and robust changes of hardware and the accompanying software and also allows for experimentation with the data processing algorithms. A mature measurement system, on the other hand, requires a high degree of automation to remove the human factor from impacting the repeatability of measurements and ensuring the uniformity of the measurement process.

Although EMMA is a universal framework, it has been created with test and measurement applications in mind. The resultant system is extensible, allowing for new measurements, analyses, and new DAQ instruments to be added, and configurable, allowing for quick customization of components. The automation of measurements is accomplished by scripting, which provides the necessary flexibility in changing measurement sequences.

## 2. Architecture
EMMA is a component-based system, where measurement applications are constructed by assembling sets of components. Thus, many different applications can be built using the EMMA framework by supplying new components and reusing the existing system and domain components.

EMMA's architectural model requires a flexible communication mechanism that supports sending messages using unicast, multicast and broadcast communication patterns. This message-oriented architecture is implemented using a publish-subscribe software bus (figure 1), which provides both local and remote communication.

In the publish-subscribe software bus architectural pattern, components subscribe to specific topics (types of data messages exchanged on the bus) and publish messages with specified topics. When a message with a given topic is published on the bus, all the components that have subscribed to that topic (subscribers) are notified. A publish-subscribe bus is similar to its hardware equivalent and, in addition to inter-component communication, allows for installation, configuration, and removal of components.

This publish-subscribe inter-component communication model results in a system of loosely coupled components, unaware of the presence of other components and their specifics. The inter-component dependencies are thereby reduced to the connascence of event names and the data structures exchanged as the associated message payloads.

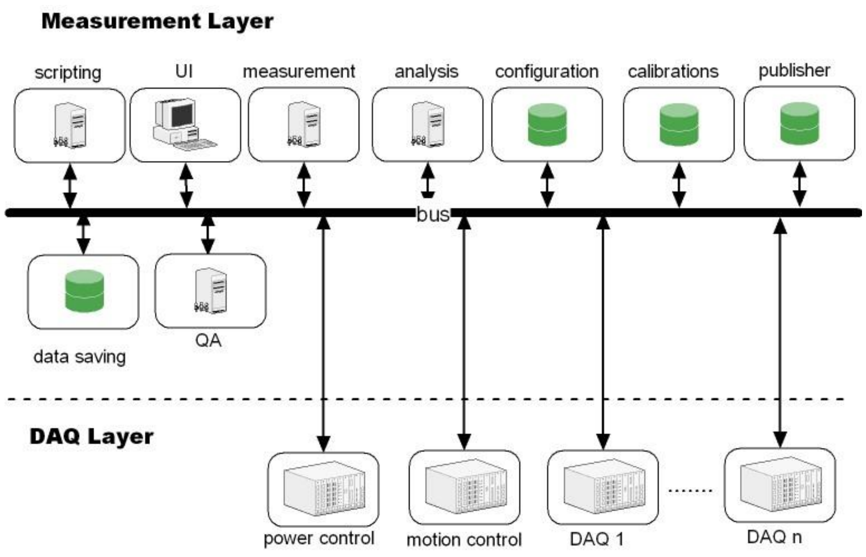


**Figure 1.** Software bus with remote and local components.

Local components (components located in the same node) communicate via message queues, and remote components (components residing in different nodes) communicate with the bus via sockets (TCP/IP).

The bus implementation in EMMA is, unlike in an Enterprise Service Bus, relatively simple with no routing or message translation functionality. A message is identified by a <topic, event> pair, which also determines the format of the message's data payload. Message topics are specified using the dot notation, and form a hierarchical name space with four root names: control, data, exception and property, denoting the following message categories:

- Control messages, which request specific actions or get statuses of hardware or software and may also manipulate the state of the component (initialize, end, suspend and run component).
- Data messages, which send results of data acquisition, analysis, or comments entered by the user.
- Property messages, which tailor components and modify their behaviour.
- Exception messages, which report problems or significant events (errors, warnings, or significant action/state changes).

The EMMA framework provides real-time monitoring of messages published on the bus, and also allows inspection of the topics a component is subscribed to.

## 3. Components

A component encapsulates the implementation of some functionality needed in an application. Component developers are encouraged to design components to serve a single purpose - to do only one thing, but do it right. This rule simplifies creation of the components that may be functionally equivalent but implemented differently, or that are based on different hardware.

The generic EMMA component is designed following the classical object model, where objects are separate entities with states and specified behavior and communicate via messages. Consequently, each EMMA component has a defined communication interface consisting of input and output messages, a set of properties that parametrize and tailor its functionality, and specific actions that define its behavior in response to software or hardware events.

### *3.1. Component structure*

EMMA components are built based on a template, which guarantees their similarity. Each standard component contains:

- Universal Connector, a module that transparently interfaces the component to the software bus and its dispatcher, using either the local or remote connection mode.
- The internal queues that provide buffering between the asynchronous component and the connector**.**
- Controller, a module implementing the component's state machine.
- The externally defined control logic.
- Executive, a module performing the actual processing.

Each EMMA component runs its controller and executive modules in a single thread and can have one or more additional parallel threads for asynchronous monitoring and control.

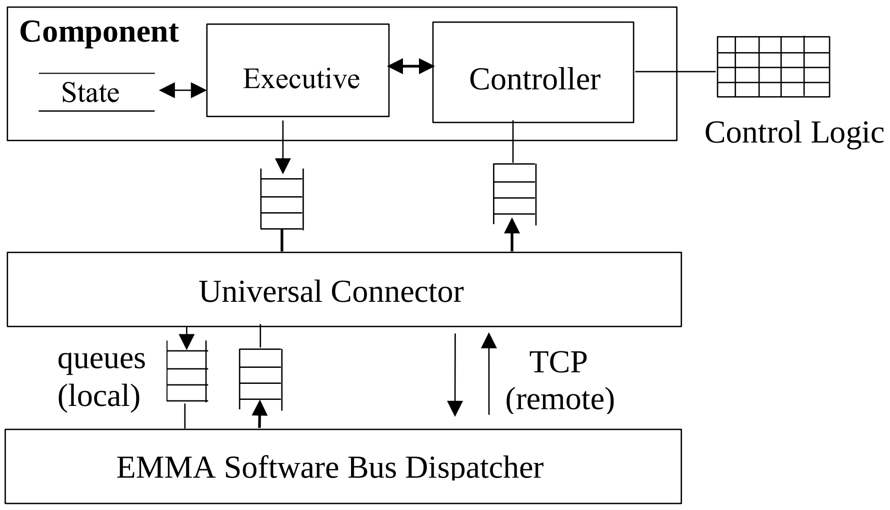


**Figure 2.** Organization of a standard EMMA component.

### *3.2. Controls and data-driven processing*

There is a common set of system control events (identified as commands or "cmd") sent to the control topics that all components respond to, which will manipulate that component's state (table 1).

In addition to the standard control events, each component may have one or more specific control commands it alone reacts to. Typically, after completing a requested command, the component sends an acknowledgment, (ACK or NACK).

Components can also receive data events. Upon receipt of a data event, the component processes the data contents (performs calculations, visualizes or persists the data) and, optionally, sends one or more data messages.

**Table 1.** Standard component control events (commands).

| Command | Action | Reply |
|---|---|---|
| **init.cmd** | Initialize the component by setting its properties to their initial values (as specified in the configuration file) and perform initialization of software and hardware in preparation for performing its core functionality. | init.ack |
| **pause.cmd** | Pause execution of the component | pause.ack |
| **run.cmd** | Resume previously paused execution | run.ack |
| **abort.cmd** | Abort current activity or function | abort.ack |
| **exit.cmd** | Clean up and terminate the component | exit.ack |

## *3.3. Component Dynamic Model*

A component's behaviour is specified by a state machine. EMMA state machine diagrams are defined externally to the component's controller as matrices, which allows for easy modification of the component's behaviour.

The state machine matrix specifies the action that will be taken, taking into consideration the received message (topic and event), the component's current state, and, optionally, a modifying condition.

The modifying conditions can be different for each component. An example of such a condition could be the name of the sender component, which might result in only events from a specified component being taken into account when processing a particular message. A standard dynamic model of a component is shown in figure 3.

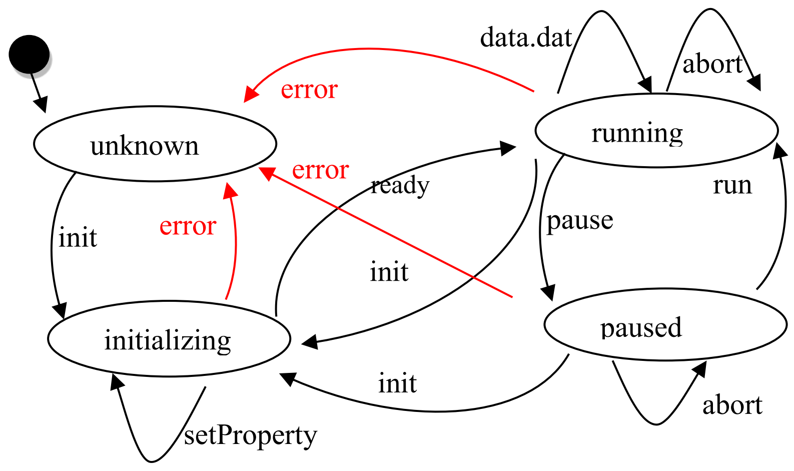


**Figure 3.** Component dynamic model (state diagram).

## *3.4. Component properties*

Each component has a number of properties that can be both modified and inspected. Properties alter or influence the execution of the component's methods by specifying parameters for data acquisition, motion, current control, etc.

The standard property events that are sent on the property topics are show in table 2.

**Table 2**. Standard EMMA property events.

| Command | Parameters | Action | Reply | Parameters |
|---|---|---|---|---|
| **set.cmd** | [property1= value]+ | Set listed properties to given values | set.ack | None |
| **get.cmd** | [property]+ | Return values of listed properties | get.dat | [property1= value]+ |
| **state.cmd** | None | Return current state of the component | state.dat | State |

[contents]+ denotes an array of one or more elements of specified contents.

## 4. Configuration
An EMMA configuration specifies the system to be executed and contains the following information:
- meta-data: information about the measurement, test type, etc.,
- the components that make up the system, both remote and local,
- component properties, which set the intended behavior of the component,
- initial control events for the components, which specifies their initial state, and
- UI components setup, which defines what to display in the viewports in the UI shell and what as top-level windows. See User Interface section for details.

EMMA configurations are stored in INI files, which are simple text files with a basic structure composed of sections that contain related parameters with their values.

## 5. System Components
The EMMA framework includes system components that provide the common and necessary functionality used by all EMMA systems. Currently, they include the following components:
- Configuration, a component that allows selection of a configuration, a script and its parameter set, and loads and prepares the configured components. It also allows inputting of measurement metadata for data documentation.
- Script, a bridge component between the Python script and the bus. It both controls the script's execution and communicates with it.
- Log, a component that receives log messages and writes them to the log files.
- Starter, a component that starts the local and remote components once the configuration is activated by the user via the configuration component.
- Persistent System Memory, a component that stores data, such as calibrations or partial results, which can be accessed in subsequent runs of the system. Since any data structure can be persisted and the components control which data to store, it creates a very powerful and flexible mechanism to build sequences of measurements with shared data.

## 6. Component Coordination
EMMA allows for coordination of components via both orchestration (a script component or a mediator component coordinates by sending control events) and via dataflow-based choreography (implementing a functionality based on the data moving through the system of components). A typical organization of the components for a measurement combines both the orchestration and dataflow–based choreography as shown in the diagram below (figure 4).

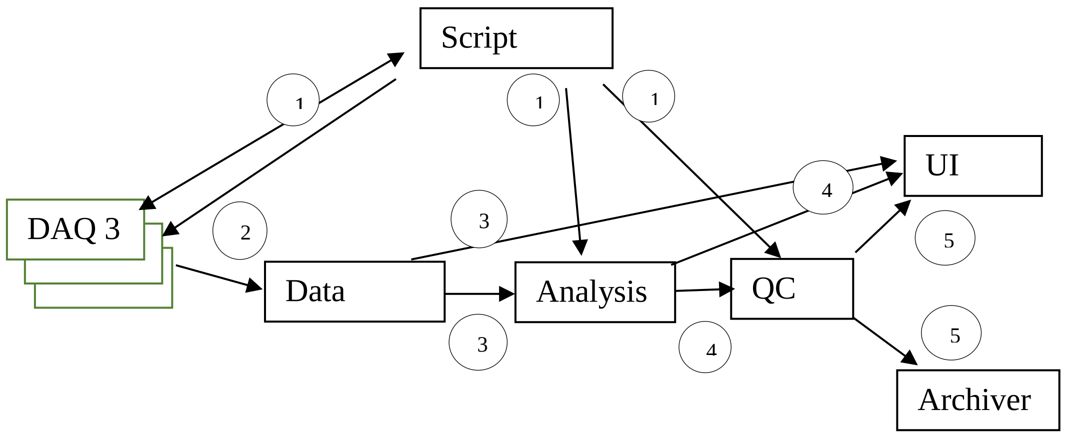


**Figure 4.** Simplified measurement collaboration diagram.

1. Script sets up components for the measurement by adjusting hardware settings on DAQ components, selecting algorithms to be used for analysis and quality criteria.
2. Script triggers data producers (DAQ components)
3. The data produced by DAQ are aggregated inside the Data component, and then received and processed by the Analysis and UI components
4. The data produced by Analysis are received by the UI and QC (quality control) components.
5. The QC component sends data to the UI and Archiver components.

## 7. Automation and scripting

### *7.1. Scripting Language*

Python was selected as the scripting language for EMMA because of its high-level language constructs (action sequencing, iteration, exception handling), its relatively high expressiveness, and its high suitability for scripting. The development of a domain specific scripting language was dismissed due to its implementation and maintenance costs.

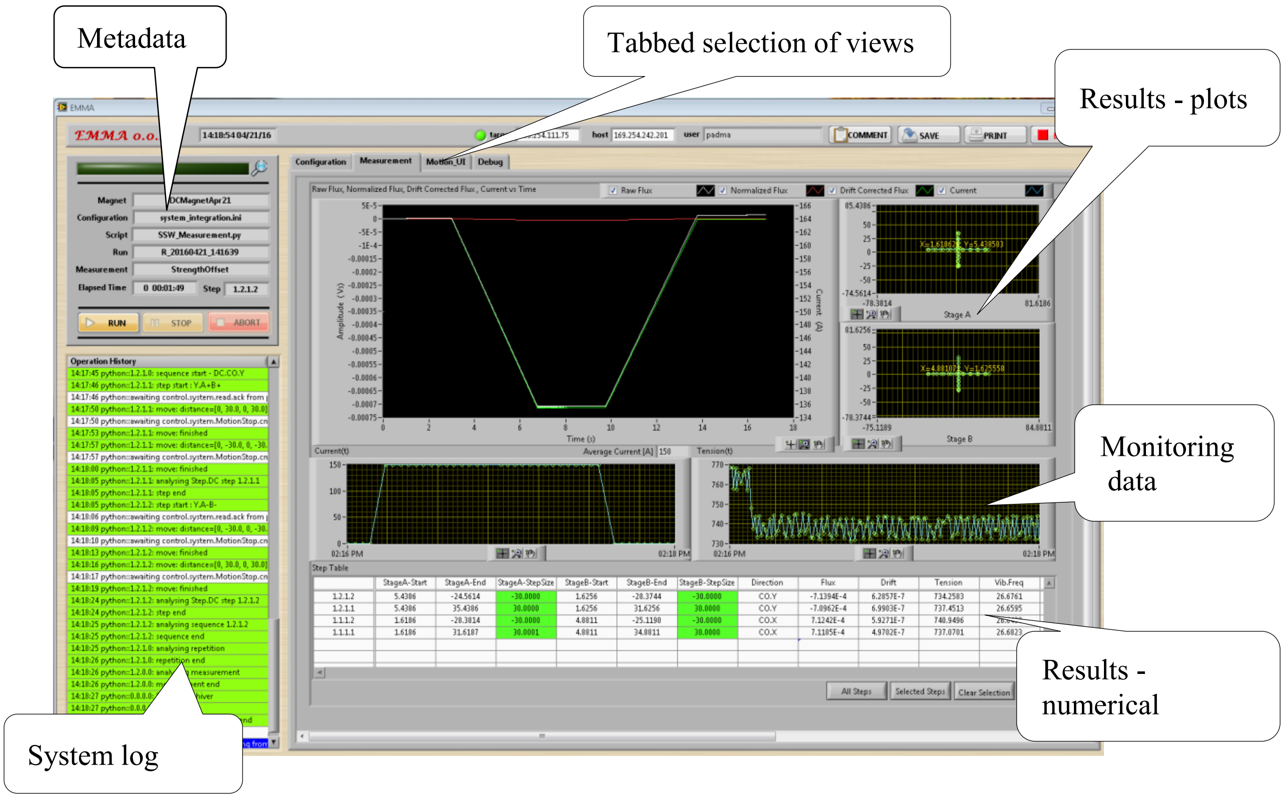
Python measurement scripts are developed using the EMMA API, which provides the necessary message-based communication primitives to communicate with the EMMA system. The API has a layered design with primitives from each layer using the lower layer primitives for their implementation. The layers of the scripting API are, in the order of increasing abstraction levels: the message layer, the event layer, and the application layer.

**Table 3**. EMMA scripting API

| Calling sequence | Description |
|---|---|
| Message Layer | |
| msgString = **recvMsg** (timeout) | Receive a message string or timeout. |
| **sendMsg** (msgString) | Send a message string |
| Event Layer | |
| **sendEvent** (format, topic, event, data) | Send event of specified format, topic, event, and data. |
| (f, t, e, c, d) = **recvEvent** (timeout) | Receive event or timeout. The event contains format, topic, event, sender component and data attributes. |
| (f, t, e, c, d) = **awaitEvent** (topic, event, sender, timeout) | Await a specified event or timeout. An asterisk in a field denotes that any value would be acceptable. |

| Application Layer | |
|---|---|
| **sendCmd** (topic, event) | Send command ( a simple event) |
| **setProperty** (topic, event, property, value) | Set property by sending an event with property name and value to an intended recipient defined by topic and event. |
| **log** (text) | Write a given entry (text) to the system log. |
| (f, t, e, c, d) = **rpc** (topic, event, replyTopic, replyEvent, sender, timeout) | Send an event to a given topic and await a reply event from a specified sender component on a reply topic. Time-out if there is no reply in a specified time. |

### 7.2. *Script parameters*
An EMMA script is not intended to contain any measurement parameters; measurement parameters are read from a separate parameter file. This design avoids the creation of multiple similar scripts with different sets of values by separately specifying the measurement conditions (e.g., set of currents, tensions, etc.) within a parameter set (or sets).



| | |
|---|---|
| Magnet | DCMagnetApr21 |
| Configuration | system_integration.ini |
| Script | SSW_Measurement.py |
| Run | R_20160421_141639 |
| Measurement | StrengthOffset |
| Elapsed Time | 0 00:01:49 |
| Step | 1.2.1.2 |

Operation History
```
14:17:45 python::1.2.1.0: sequence start - DC.CO.Y
14:17:46 python::1.2.1.1: step start : Y.A+B+
14:17:46 python::awaiting control.system.read.ack from
14:17:50 python::1.2.1.1: move: distance=[0, 30.0, 0, 30.0]
14:17:50 python::awaiting control.system.MotionStop.cn
14:17:53 python::1.2.1.1: move: finished
14:17:57 python::1.2.1.1: move: distance=[0, -30.0, 0, -30.
14:17:57 python::awaiting control.system.MotionStop.cn
14:18:00 python::1.2.1.1: move: finished
14:18:05 python::1.2.1.1: analysing Step.DC step 1.2.1.1
14:18:05 python::1.2.1.1: step end
14:18:05 python::1.2.1.2: step start : Y.A-B-
14:18:06 python::awaiting control.system.read.ack from
14:18:09 python::1.2.1.2: move: distance=[0, -30.0, 0, -30.
14:18:10 python::awaiting control.system.MotionStop.cn
14:18:13 python::1.2.1.2: move: finished
14:18:16 python::1.2.1.2: move: distance=[0, 30.0, 0, 30.0]
14:18:17 python::awaiting control.system.MotionStop.cn
14:18:19 python::1.2.1.2: move: finished
14:18:24 python::1.2.1.2: analysing Step.DC step 1.2.1.2
14:18:24 python::1.2.1.2: step end
14:18:25 python::1.2.1.2: analysing sequence 1.2.1.2
14:18:25 python::1.2.1.2: sequence end
14:18:25 python::1.2.1.0: analysing repetition
14:18:26 python::1.2.1.0: repetition end
14:18:26 python::1.2.0.0: anal      measurement
14:18:26 python::1.2.0.0: m       ent end
14:18:27 python::0.0.0.0:        hiver
14:18:27 python::0.0.0
```

Step Table

| | StageA-Start | StageA-End | StageA-StepSize | StageB-Start | StageB-End | StageB-StepSize | Direction | Flux | Drift | Tension | Vib.Freq |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.2.1.2 | 5.4386 | -24.5614 | -30.0000 | 1.6256 | -28.3744 | -30.0000 | CO.Y | -7.1394E-4 | 6.2857E-7 | 734.2583 | 26.6761 |
| 1.2.1.1 | 5.4386 | 35.4386 | 30.0000 | 1.6256 | 31.6256 | 30.0000 | CO.Y | -7.0962E-4 | 6.9903E-7 | 737.4513 | 26.6595 |
| 1.1.1.2 | 1.6186 | -28.3814 | -30.0000 | 4.8811 | -25.1190 | -30.0000 | CO.X | 7.1242E-4 | 5.9271E-7 | 740.9496 | [illegible] |
| 1.1.1.1 | 1.6186 | 31.6187 | 30.0001 | 4.8811 | 34.8811 | 30.0000 | CO.X | 7.1185E-4 | 4.9702E-7 | 737.0701 | 26.6823 |

**Figure 5.** A measurement view inside the Shell module.

## 8. User interfaces
The Shell module is a basic user interface provided for each UI component (figure 5). It contains views for metadata and logs, and viewports, which are empty panels in which the component can place its own front panel with customized data visualizations. UI components have the option to show their output in Shell viewports or to run as top-level windows.

## 9. Summary

The EMMA framework embraces a fine-grained, component-based architecture, which produces a network of asynchronous communicating processes organized into components. Each component is independent and can evolve internally without compromising system integration as long as its functionality boundaries and external interface remain unchanged. Components are short lived, created for the duration of running a particular system. A component has a set of properties that can be initialized to system customized values and then dynamically modified to alter the behavior of the particular component at run-time.

The functionality of each application created with the EMMA framework is orchestrated by scripts. Several different scripts can execute on a given set of components providing different behaviors, with each script parameterized to provide an easy way to tailor the application's behavior.

The EMMA framework has been built upon the premise of composing test and measurement systems from independent components. This opens up opportunities for reuse of components and their functionality and composing them together in many different ways. It provides the measurement system developer with a lightweight alternative to microservices, while sharing their various advantages, including composability, loose coupling, encapsulation, and reuse.

The EMMA framework system builds on experiences gained from designing and using configurable multi-measurement systems, including those developed by the authors - the CHISOX and EMS systems [1] [2]. To date, it has been successfully used to build two measurement systems, and another system is already in the inception phase.

## 10. References

[1] Sim J et al. 1995 Software for a Database-Controlled Measurement System at the Fermilab Magnet Test Facility *16th Biennial Particle Accelerator Conf. (Dallas)*

[2] Nogiec J, Sim J., Trombly-Freytag K, and Walbridge D 2000 EMS: A Framework For Data Acquisition And Analysis *VII Int. Workshop on Advanced Computing and Analysis Techniques in Physics Research (Batavia)*

**Acknowledgments**

This work has been conducted at Fermi National Acceleratory Laboratory operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy.