A Tool for Dynamic Data Capture and Visualization in Heterogeneous Simulation Environments

by

Matthew Genovese, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2005

A Tool for Dynamic Data Capture and Visualization in Heterogeneous Simulation Environments

Approved by
Supervising Committee:

Dedication

To my parents Michael and Maria, my ever patient wife Jennifer, and children Caleb and Emily – my absolute pride and joy.

Acknowledgements

I would first like to thank Dr. Margarida Jacome for agreeing to be my faculty supervisor, and for her constant availability for discussion and guidance. I am grateful to my advisors Steven Smith and Mark McDermott, who helped me develop and cultivate the idea for this project in the first place. Thank you to Dr. Adnan Aziz for generously giving of his free time to review the specification for this project. Thank you to Anders Hedström who provided tremendous support and speedy enhancement of his open-source C++ TCP/IP Sockets library. I am grateful to the OpenDX community on the user support e-mail lists, where people whom I have never met authored detailed and immensely helpful responses to my newbie questions. Finally, thank you to Rich Lathrop, my first MAT alpha-site user, who provided me with valuable feedback and constant encouragement as he used the tool for data acquisition during simulation of his own graduate project.

December 2, 2005

Abstract

A Tool for Dynamic Data Capture and Visualization in Heterogeneous Simulation Environments

Matthew Genovese, M.S.E.

The University of Texas at Austin, 2005

Supervisor: Margarida Jacome

System performance is important to accurately validate as early as possible in the design process. Throughout the process of design refinement, engineers assemble heterogeneous simulation environments that commingle sub-models at various levels of abstraction, using assorted hardware description languages and system-level design languages. Continuous performance validation can be possible in these environments throughout the architectural exploration process, and subsequently during design implementation. In order to free engineers to measure performance for any resource within the heterogeneous system during simulation, it is necessary to capture data from any point of abstraction within the model to a single database. Consequently, by providing a graphical and configurable visualization interface into this database, the architect can easily group data and quickly assemble key metrics that enable conclusions to be made about system performance. This report is the culmination of a project undertaken to develop a tool that implements the above proposal.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Preface

ABSTRACT

System performance is important to accurately validate as early as possible in the design process. Throughout the process of design refinement, engineers assemble heterogeneous simulation environments that commingle sub-models at various levels of abstraction, using assorted hardware description languages and system-level design languages. Continuous performance validation can be possible in these environments throughout the architectural exploration process, and subsequently during design implementation. In order to free engineers to measure performance for any resource within the heterogeneous system during simulation, it is necessary to capture data from any point of abstraction within the model to a single database. Consequently, by providing a graphical and configurable visualization interface into this database, the architect can easily group data and quickly assemble key metrics that enable conclusions to be made about system performance. This report is the culmination of a project undertaken to develop a tool that implements the above proposal.

BACKGROUND

System models are progressing to yield heterogeneous simulation environments, which enable them to become more broadly used throughout the design process. Initially, an Electronic System Level language (ESL) may be employed to construct an untimed or synchronous (cycle-accurate) model at a high level of abstraction early in the design process to explore performance and validate hardware and software requirement assumptions. As the design process progresses, parts of the same model may become more well-defined and more closely represent the actual hardware. Sub-models are refined to more accurate models of computation, thus yielding a heterogeneous simulation environment with Hardware Description Languages (HDL's) engaged to implement these new abstractions. Additionally, external design and/or verification intellectual property (IP) may be integrated into the simulation environment as it becomes more solidified and ready for functional verification, leading to additional overall diversification. Orthogonal to this development process, variations of the same environment can be used by system architects, designers, verification engineers, and software application engineers – each with their own goals, and therefore preferred input interface(s) and associated languages used to stimulate the model. A beneficial characteristic of such a progressive unified environment is that the accuracy of initial performance estimates will increase as the level of design abstraction lowers towards a more accurate hardware implementation at the Register-Transfer Level (RTL), and beyond to the gate-level. Therefore, a goal should be for engineers to monitor the system and block-level performance metrics of the model as the level of abstraction proceeds toward implementation, and ensure the predicted aspects of performance converge towards the previously estimated goals.

However, this simulation environment is plagued by the same heterogeneity that enabled it to be employed throughout the design process. In order to acquire performance measurements during simulation, the architect needs to extract data from the environment, conceivably from various points scattered about the model and stimuli, each potentially modeled at different levels of abstraction. The method used for collection of the data may be as simple as employing embedded textual display statements that are post-processed by hand or script, though either method may prove to be relatively inflexible, or a non-trivial endeavor. Other methods may involve using custom in-house developed tools, or vendor tools designed for this purpose; nonetheless, either scenario requires that the tool support communication with the diversified environment.

Once collected, the raw data alone may not yield meaningful information, unless it is condensed and combined to create *performance metrics*. Performance metrics summarize vast quantities of simulation data to accurately assert meaningful indices of system performance. These metrics provide feedback in the design refinement process for relative performance improvements when compared to previous designs, and an understanding of where performance problems currently exist when the system encounters specific preconditions.

One way to start assembling relevant performance metrics is to view the captured data graphically, and visually associate related streams of data to initiate the analysis and correlation of variables. Viewing one or more sets of captured data over time, or creating scatter plots of two or more dependent axes can aid in analysis, and subsequently allow the engineer to devise metrics that are useful for the particular analysis underway. The visualization of data as multi-dimensional objects that can be freely manipulated in a graphical interface is seen by the author as positively contributing to this analysis.

MOTIVATION

Current Electronic Design Automation (EDA) tools in the architectural exploration and performance modeling arena are geared towards providing integrated high-level design environments. In surveying the field, these tools are designed with the architectural exploration, co-design, and rapid prototyping features tightly coupled to the performance measurement aspects of the tool. Within that group of tools, a small number are able to handle heterogeneous simulation environments with mixed languages and levels of abstraction. Furthermore, the performance measurement capabilities are often coupled to the proprietary models provided by the EDA company, yielding standardized and rigid measurement facilities available to the user. Thus, once the design implementation has commenced, the performance measurement facilities within these tools decrease in value, and it becomes difficult to assess and feedback the performance of the RTL versus the original estimate gathered from the high-level model. In addition, customers may have existing functional models in RTL, unique behavioral stimuli, and/or entire in-house developed simulation environments already available, such as for derivative products. In these scenarios, it is quite possible that leveraging a portion of this intellectual property or environments for reuse in a new performance model is desirable, and only the performance measurement features are required versus the overhead of an entire architectural exploration suite.

The motivation of this project is to develop a tool that enables the engineer to overcome the described obstacle with heterogeneous simulation environments by providing various methods of data capture from anywhere within the model and stimuli during simulation. With uniform support for different software, electronic system-level, and hardware description languages that may be present, the architect can capture nearly any type of data from any employed level of abstraction within the model, including continuous-time, discrete-time, synchronous, and untimed models of computation. Additionally, the tool framework provides a centralized online repository for deposition of this data, and enables access to this data via a multi-dimensional, highly configurable visualization interface for performance analyses. The engineer is then empowered to explore the data and draw conclusions to feedback into the design process, from design-space exploration through to RTL implementation.

CHAPTER 2

Introduction

RESOURCES AND PERFORMANCE MEASUREMENT

In order to realize most any measurement of performance[1], the evaluation must be performed relative to a particular resource. In the context of this report, a *resource* is simply defined as a specific item of observation within the model for which data is captured during simulation, with the goal of assessing a type of performance. The item may consist of a single gate, a logic block, a group of associated blocks, or even the entire model. However, the precise definition of the resource should always be kept mindful because it declares the portion of the model to which the derived performance metrics pertain. Equally as important, it declares the set of external resources that the performance metrics do <u>not</u> directly measure.

The instrumentation developed for this project was devised with resource performance measurement in mind. In particular, the measurement facilities provide a means for the user to ultimately derive *weighted performance metrics*. These are essentially figures of merit describing overall system performance given a relative weighting of the relevance and criticality of individual performance metrics. This is accomplished by enabling the user to capture both raw resource performance data and resource state data over time during the simulation, and subsequently combining them to accurately describe how the resource performance modulates overall system performance based upon the actual resource usage with a given system configuration and load.

[1] The use of the term *performance* is purposefully ambiguous because most any specific aspect of performance can be substituted by the reader in this context, such as timing or power performance.

Given this strategy, the three categories of measurement devised for this tool are as follows:

- **Resource Utilization** – Measurements yielding the arbitrary state of a resource during simulation.
- **Resource Performance –** Measurements yielding performance information for a resource during simulation.
- **Global Measurements –** Measurements not relative to a particular resource, but rather yielding information that pertains to the entire system model, or simulation environment.

Per these definitions, any given weighted performance metric can be calculated as the cross product of the observed resource utilization statistic and the desired resource performance measurement over the course of the simulation. Whereas individual resource performance measurements alone focus on the sole operation of the resource, a weighted performance measurement is calculated as individual performance in terms of its utilization within the system, and yields a system-relevant metric.

Resource Utilization

The goal of resource utilization measurements is to ascertain the states of a resource, and time intervals thereof, continuously over the duration of the simulation. The *resource state* is an arbitrary summarization of the condition of the resource for a period of time. For instance, almost any given functional resource can be assigned a resource state of either BUSY or IDLE, designating if the resource is active or not active, respectively. Building upon that degenerate case, more complex and meaningful resource states can be devised to amplify visibility of the resource activity during simulation. An example can be found in Figure 1 below, where the state of a resource *bus_state* is captured over time as the simulation progresses. As shown, three state values are arbitrarily applied by the user (*Idle*, *Arbitrate*, and *Transfer*), presumably signifying the temporal condition of the resource under observation.

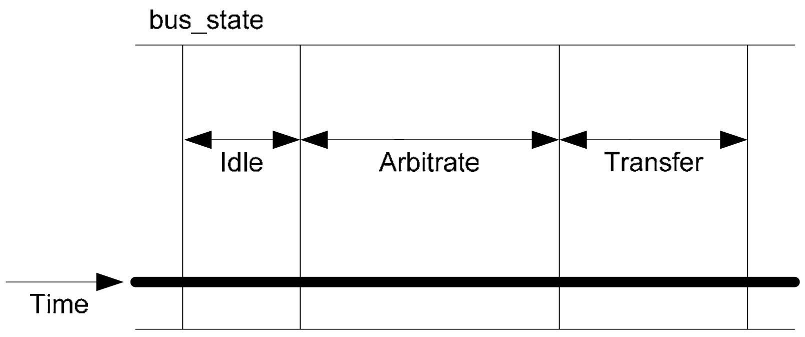

Figure 1 – Example Depiction of a Resource State Measurement

Recalling the latter part of the above definition, any observed resource state has an associated bounded duration of time. In a timed model, a bound may be any duration between the entire length of the simulation, and the smallest time duration made available by the event-based or cycle-based simulator. Even in an untimed model, a notion of time can be applied to partition functional steps towards completion of an algorithm, and can be weighted by the anticipated effort required to perform each step. Thus, every resource state measurement contains an annotated time of transition to the state, and duration of the state. Accumulation of these resource state statistics over the course of the simulation yields the resource utilization, segregated into the total amount of time spent in each state.

Resource Performance

The goal of a resource performance measurement is to obtain raw or calculated numerical values, or record milestones observed in the model during simulation that specify direct measures of instantaneous or cumulative performance. Various types of measurements fall into the first category; for instance, general numerical value measurements, quantifications of latency (delay), bandwidth or data-rate (data quantity per unit time), activity factors (percent of resource in operation), and power consumption (energy consumption per unit time.) Though most any value can be captured from the model to designate some quantification of performance, it is important to remember the measurement should not account for the state of the resource under observation; this is the responsibility of the resource utilization measurement.

Another means of measuring resource performance is to assess performance as an accumulation of functional milestones during simulation. The milestone, or *resource event*, is posted at the point in time when the observation is made, and may be recurrent as time progresses. The resource event carries an annotation of the time of occurrence, and the cumulative count for that particular event from the beginning of simulation. This allows visibility into the functionality of the resource for any arbitrary time interval during simulation, and can be used to derive performance statistics. For example, a user wishing to monitor cache subsystem performance may decide to post a resource event when a data request results in a cache-miss. The user realizes that when the cache-miss event is posted, it is an implication that overall performance may be degraded because the cache could not immediately provide the requested data. By tracking the quantity and proximity of occurrences of this event during the simulation, exploration can be done to ascertain the cause of this performance-degrading behavior, and whether it occurs enough to warrant some redesign.

A resource event only exists at a single point in time. However, it can be useful to relate separate but logically connected resource events that together formulate a meaningful span of time. The result is a *resource event span*, where resource events are posted individually, and yet related to other resource events occurring at different times.

The relation of the resource events is arbitrarily defined by the user, and yields a dynamic association of resource events regardless of the order they are posted in simulation.

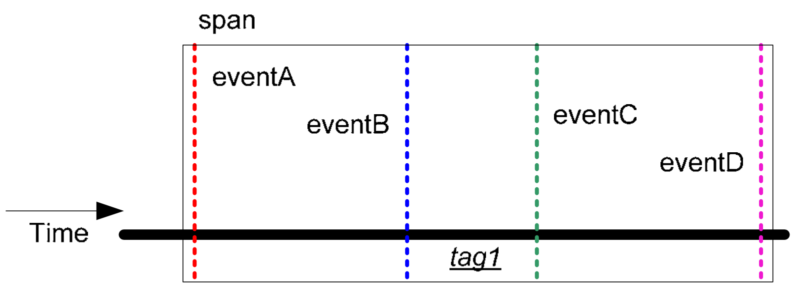

Figure 2 – Example Depiction of a Resource Span Measurement

The relation of the grouped events in the span is accomplished by use of a common *span tag*. As depicted in Figure 2, the four events posted are related by the tag *tag1*. Although each posted resource event exists at a single point in time, the tag relates the events such that the span exists over a duration of time. Each time an event is posted, the tag is used to associate it with other events posted with the same tag, and subsequently calculate the time delta since a previous event with the same tag (i.e. within the span). By assigning a common thread of data to be a span tag, the span measurement enables data traces through a resource, thus providing an understanding of temporal performance along a prescribed data path.

Global Measurements

The goal of a global measurement is to yield information about the overall model or the simulation environment, not pertaining to a particular resource in general. The measurement can be a declaration of *global state* of the model or simulation, similar to that of a resource state. One example is a global state that defines the condition of the model as being in a reset phase, or in a normal operation phase. This state can be thought to apply to the entire model, or across all functional resource boundaries in the model, and therefore global to all resources as an indication of overall device state. Thus, this is an appropriate use of a global state measurement. Similar to that of a resource state, global states have an attributed time of transition to state, and duration of state.

Another global measurement is a *global event*, which is similar to a resource event. An example use of a global event is to post an event when a new test case is applied in the simulation. The event milestone can be used to logically partition individual tests that are applied in the same simulation. Similar to that of a resource event, global events have an attributed time of event posting, and cumulative count for the particular event.

MODEL ANALYSIS TOOL OVERVIEW

The purpose of the Model Analysis Tool (hereafter abbreviated *MAT*) is to enable dynamic extraction of data from a homogeneous or heterogeneous simulation environment using the resource-centric philosophy described above, and deposit the data into a centralized data store for access and manipulation via a graphical visualization interface.

Key Features

- The MAT front-end Data Capture Interface Library commands have a unified syntax and can be used in a variety languages (Verilog, C, C++, SystemC) to capture data from continuous-time, discrete-time, synchronous, and untimed models.
- Data capture commands can be scattered about the simulation environment as virtual probes that send most any type of data to MAT as the simulation executes.

- Online storage of the captured data can reside on a separate workstation from the system running the simulation, which decreases the simulator memory overhead associated with using MAT to collect data during a simulation.
- Visualization enables data viewing as multi-dimensional plots that can be manipulated on-screen to best present metrics for performance analysis.

User Command Taxonomy

In accordance with the performance assessment categorization outlined in the preceding discussion, the user instrumentation for the Model Analysis Tool follows the same organizational structure, and supplements with additional control functionality as summarized below.

Command Category	**Description**
Time Management	Enables creation of *timesets*, which are arbitrary notions of regular or irregular time.
Data Management	Enables creation of *datasets* – the top-level organization for data, with optional association with predefined timesets.
Resource Utilization Assessment	Provides commands that enable resource measurement in terms of temporal state.
Resource Performance Assessment	Provides commands that enable resource measurement in terms of temporal behavior or function.
Global Measurement	Provides commands that measure global aspects of the model or the surrounding simulation environment.
General Control	Provides commands that manage aspects of MAT functionality.

Table 1 – User Command Taxonomy

The subsequent chapter will more thoroughly describe the commands available to the user within each of these categories in terms of syntax and usage.

Visualization

The visualization of the acquired data is accomplished through an open-source tool called OpenDX. This is a full-featured application that can be used as a stand-alone visualization environment, or integrated with other applications via the OpenDX Application Programming Interface (API). The current integration with MAT allows OpenDX to run as a stand-alone tool, and import MAT-created data for subsequent analyses. Future revisions are planned to fully integrate OpenDX with MAT to yield a unified visual interface into the data storage, and to provide predefined visualizations that can be applied by the user to the imported data.

CHAPTER 3

The Model Analysis Tool

HIGH-LEVEL ARCHITECTURE

The Model Analysis Tool is comprised of three separate entities that work together to provide a means of data capture from the model during simulation, maintain online data storage, and enable graphical visualization, as depicted in Figure 3 below. The first is the *Data Capture Interface Library* (DCIL), which is a shared library that is linked in with the model and stimuli as the simulator executable is created. The DCIL is primarily responsible for receiving data obtained from the simulation environment, and subsequently translating this data into the internal MAT data model via creation of data objects that are sent to an online storage server. Several versions of the DCIL are provided, one for each software or hardware description language supported. In addition to each shared library, a text header file is also supplied that provides the function prototypes for each DCIL command. This header file is included during the model compilation process, and is necessary for any source code files that make use of the MAT user commands, which are essentially DCIL library calls.

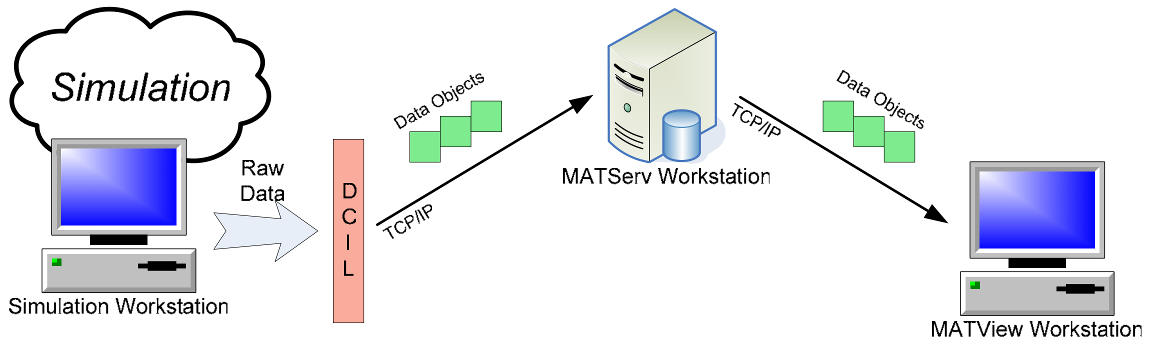

Figure 3 – Example of the Distributed MAT Architecture

The next application provided is *MATServ*, which is a data server that receives the MAT data objects from the DCIL, and stores them for subsequent access by the visualization application. This server may be running on the same workstation as the simulator executable, or may reside on a different workstation, as shown in Figure 3. Communication between the DCIL and *MATServ* is accomplished via a TCP/IP network socket connection [2]. Thus, the workstation executing the simulation model need not suffer from a decrease in available memory due to the data storage required by *MATServ*. In addition, future expansion of *MATServ* will feature lossy data compression such that large quantities of related data in storage can be algorithmically compressed to decrease the storage footprint in the workstation memory, and the size of the subsequent OpenDX visualization data model. For example, simple data compression or aggregation techniques can be employed, such as remapping the original data into moving averages to reduce the data storage required.

The final MAT application is *MATView*, which enables the data visualization. When executed on the same or a separate workstation as *MATServ*, *MATView* communicates with the data server via a TCP/IP socket connection to receive data objects as they become available from the DCIL during simulation. *MATView* can be operated asynchronously with respect to the simulation, and therefore can receive the latest data when subsequent requests are made to *MATServ*. *MATView* is responsible for acquiring the MAT data objects from *MATServ*, and translating them into the separate data model used by the OpenDX visualization tool. Currently, this OpenDX data model is exported to a text file which can be imported by the OpenDX tool for visualization.

MAT DATA MODEL

MAT maintains an internal data model for organization of inbound data from the simulation environment. Portions of the MAT data model are created explicitly via DCIL commands, while other parts are created dynamically as data arrives. It is important to understand the MAT data model structure before proceeding to an explanation of the DCIL commands.

A Notion of Time

At the heart of the MAT data model is the concept of time. From the perspective of the heterogeneous environment, the notion of time can have different meanings, depending upon the model of computation utilized for a given resource. Discrete-time model abstractions typically rely a base simulation clock, from which all signal transitions, state changes, and model-generated events have an associated simulation clock timestamp. Synchronous models of computation also rely on a simulation clock to indicate when all evaluations are instantaneously performed in each clock cycle. In both cases, the simulation clock is typically the highest frequency clock in the model. Continuous-time models of computation execute with a notion of analog time versus the discretized time that was managed with the previous models. This can be thought of as non-integer time, where the analog precision of a model measurement is only limited by the functional resources of the machine conducting the simulation. Finally, untimed models of computation are by definition without a concept of time; however, even here time can be arbitrarily applied to denote successive points of algorithmic execution.

All inbound data to the DCIL have an associated simulation timestamp appended. Ultimately, this enables the user to create the most basic two-dimensional plot of the data values versus the simulation time when they were acquired. As multiple variables are created during simulation, each with its own set of points consisting of data values and timestamps, the dependent variables can be correlated with their common simulation time to create multi-dimensional scatter plots.

Although simulation time is the most basic notion of time, timed models typically function with clocks running slower than the simulation clock. For example, a discrete-time or synchronous microprocessor model may have a clock domain for the internal core, and another domain for the system bus clock, both of which are slower than the simulation clock. These are examples of clocks that exhibit regularity, in that the steady-state clock frequencies are constant. In contrast, recurring events of interest may also exist in an timed or untimed model, such as bus transactions or keep-alive packets on a network. These recurring events can be considered a type of clock, even if they exhibit timing irregularity with respect to the frequency of occurrence. In either case, the user may wish to timestamp inbound data values according to a regular or irregular user-defined clock; MAT addresses this by providing *timesets*.

A *timeset* is an alternate notion of time, defined as monotonically increasing integer points along an axis, relative to simulation time. A timeset can track a traditional clock within the design that exhibits regularity as shown in Figure 4, or it can track irregular events that occur over varying intervals as shown in Figure 5. The creation of a timeset is performed via a call to the DCIL from within the simulation environment. Similarly, the trigger to increment a particular timeset is performed via a call to the

DCIL. MAT keeps track of the timeset time as a function of simulation time, and can apply the current timeset timestamp to any inbound data value received from the simulation environment, in addition to the simulation time. Thus, ultimately the data values can be plotted versus the simulation time, or the time of capture according to a timeset time, the latter of which may be of more useful to the user for the particular analysis being performed.

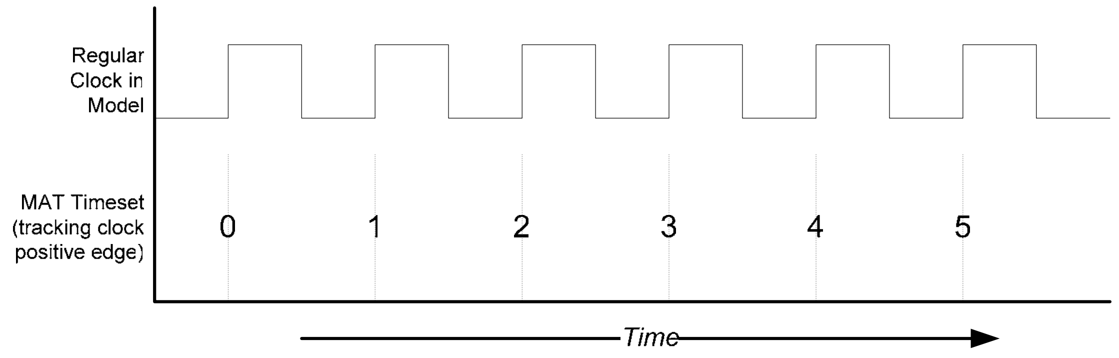

Figure 4 – Timeset Tracking of Regularity in the Model

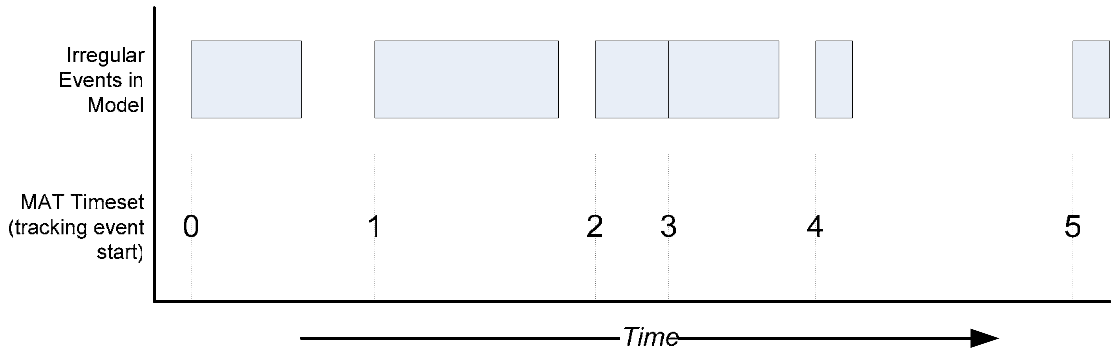

Figure 5 – Timeset Tracking of Irregularity in Model

Note that with both simulation timestamps and timeset timestamps, it is possible for multiple data values pertaining to the same variable to arrive in the same time instant, and therefore with the same timestamp. In this case, the last value received is the one ultimately stored, overwriting all previous values captured.

Data Model Organization

The highest level of data categorization within the MAT data model is a *dataset*, as depicted in Figure 6 below. A dataset is a collection of axes (variables) that can be logically related over time, or to each other. The notion of a dataset is perhaps more philosophical than concrete, as the user is left to decide which sets of data formulate a reasonable grouping. However, it is suggested that a single dataset should be relevant to a single resource within the model, and a single time domain, apart from simulation time.

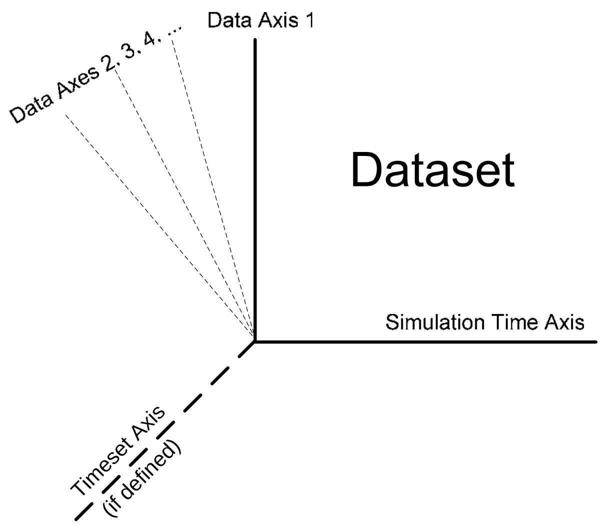

Figure 6 – Depiction of a Dataset

Datasets are created explicitly by the user via a DCIL command, and must be created before any captured data can be sent to it. The dataset creation typically occurs early in the simulation to define the dataset name, and whether there will be an association with a predefined timeset. If the dataset is not associated with a timeset, all inbound data values into the dataset will be appended with the current simulation time. Otherwise, if the dataset is explicitly associated with a timeset, all inbound data values into the dataset will be appended with the current simulation time and timeset time. MAT supports creation of datasets with at most one associated timeset.

Underneath the MAT data model hierarchy of datasets exist *axes*, each of which is likened to a one-dimensional array of data values that expands to store all data sent to it from the DCIL. Upon creation, every dataset has one or two axes created automatically: the simulation time axis, and the timeset time axis (if the dataset was associated with a timeset at creation.) Unlike datasets, new axes are automatically created as they are referenced by data capture calls to the DCIL from the simulation environment. That is, the first time in simulation that data is captured by the DCIL for a yet unreferenced dataset axis name, the axis is created by MAT and the data is stored. Subsequent references to the same dataset and axis will continue to store the data as defined by the particular DCIL command.

Conceptually, the MAT data model can support an unlimited number of datasets, and an unlimited number of axes for each dataset. However, the number is practically limited by the memory available to the *MATServ* data server. The intended application of the MAT data model is for the user to create a dataset for each resource under observation that does not require an associated timeset, or for each resource within a specific time domain that requires a MAT timeset for advanced time tracking. Axes within each dataset should be created and segregated based upon the type of information being collected. An axis collecting state data for a resource utilization measurement should exist only to collect that type of information. If deemed necessary, several aspects of the resource's state can be captured, each on its own axis. Additionally, other axes can be created to capture the desired aspects of resource performance, such as a latency, or the observed quantity of data transmitted over time, and so on. By grouping these single resource-related axes into a unique dataset, the resource performance and utilization data can immediately be combined and presented graphically over independent time axes (simulation time, and timeset time, if available). Additionally, the common simulation time axis allows multi-variable correlation to be performed between dependent axes within the dataset to determine how they track each other.

DATA CAPTURE INTERFACE LIBRARY

As previously described, the Data Capture Interface Library (DCIL) is the interface between the simulation model and MAT, providing the commands by which model data is captured and subsequently translated into the internal data model representation as MAT data objects. The DCIL provides several categories of commands to initialize portions of the MAT data model, and accomplish the specific type of data capture.

All of the DCIL measurement commands which perform a data capture have at least two arguments: the dataset name and axis name for which the data is destined. In some versions of the DCIL (e.g. for C, C++), the *simulation time* argument is also required. For many commands, this argument is needed because the supported language or employed model of computation does not intrinsically have a notion of time, so this must be supplied by the user. In other versions of the library (e.g. Verilog, SystemC), the language does have intrinsic simulation time support which is automatically captured by the DCIL command, and therefore it does not need to be explicitly specified in the command arguments. By convention, if the simulation time is required, it is always the last argument in the DCIL command.

Some data value measurement commands have another argument called a *tag*. More generalized than the *span tag* mentioned in the previous chapter, a *tag* is a means by which the user associates previously captured data with newly captured data to formulate the new data point. Tags are created dynamically as they are used, similar to how axes are created on-the-fly. A tag is employed when the captured data may not directly correspond to the data immediately stored. There are several measurement commands that employ tags, such as **mat_measure_delta()**. As will be described later, this command uses the current data value measured, and subtracts the previous data value measured with the same tag name, and stores that new value as a MAT data object, or data point. The initial case is when **mat_measure_delta()** is called with a new tag name on a particular dataset and axis; in this scenario, no data object can be created because there is not a previous value to subtract. Rather, the current data value is stored within the DCIL, and awaits a subsequent **mat_measure_delta()** command call with the same dataset, axis, and tag name. This will cause MAT to compute the difference and create the first data point.

The sections below are categorized by function, and detail each of the DCIL commands available for use in the model. In each section, the commands will be designated as having an "HDL Syntax" (for Verilog and SystemC with an intrinsic notion of time), and a "C Syntax" (for C and C++ which do not have an intrinsic notion of time.) If one syntax is provided, it is applicable for all DCIL language implementations.

> **Note**: The DCIL syntax for some commands allows for *optional* arguments, and denotes them as surrounded by square brackets "[ ]". The DCIL as implemented for C++ and SystemC allows for any unused optional argument for a command to be simply discarded from the list of arguments. However, for the DCIL implemented in Verilog and C, optional arguments are not allowed. Therefore, all arguments, including

any denoted below as optional, must be included. In the case where an optional argument would normally be discarded, the syntax should replace the argument with zero (0), and the DCIL will handle the command as if the argument was not present.

Timeset Creation

The two commands in this category handle the creation of a timeset, and the increment of a timeset time, which will be applied to all subsequent inbound data for datasets that associate with the timeset.

Timeset Commands

Syntax:	**mat_timeset_define** (<*timeset*>, <*units*>)	
Parameters:	timeset (string)	Name of the timeset.
	units (string)	Units of the timeset.
Description:	Defines a timeset name and the associated units. This timeset name is later referenced when updating a timeset time, or defining a dataset that will be associated with the timeset.	

Syntax:	**mat_timeset_update** (<*timeset*>)	
Parameters:	timeset (string)	Name of the timeset.
Description:	Given a predefined timeset, this command monotonically increments the current time value associated with this timeset.	

Table 2 – Timeset Commands

Example Usage

Below is an example Verilog code snippet showing how MAT timesets are created and updated.

```
initial
begin
  $mat_timeset_define (“pci_clk”, “PCI Clocks”);
  $mat_timeset_define (“core_clk”, “Core Clocks”);
  $mat_timeset_define (“ddr_clk”, “DDR Clocks”
end
…
always @(posedge pci_clk)
  $mat_timeset_update (“pci_clk”);
…
always @(posedge core_clk)
  $mat_timeset_update (“core_clk”);
…
always @(ddr_clk)
  $mat_timeset_update (“ddr_clk”);
```

In this example, three timesets are created when the simulation begins. Subsequently, Verilog *always* blocks are used to ensure each timeset is updated when the positive edge of each respective clock is observed.

Dataset Creation

This category is responsible for the creation of datasets.

Dataset Creation Command

The command below creates a dataset, with or without association with a predefined timeset. Note that the timeset argument is optional; if not applied, the dataset will not have a timeset association.

Syntax:	**mat_init_dataset** (<*dataset*>, [<*timeset*>])	
Parameters:	dataset (string)	Name of the dataset to be created.
	timeset (string)	Name of timeset associated with this dataset. If not specified, no timeset association is performed.

Table 3 – Command Syntax of mat_init_dataset()

Example Usage

The below example Verilog code shows how datasets are created without timeset association, and with timeset association.

```
initial
begin
  // Create dataset bus_arbiter without a timeset association.
  $mat_init_dataset (“bus_arbiter”);
  // Create dataset pci_perf with association with timeset pci_clk.
  $mat_init_dataset (“pci_perf”, “pci_clk”);
end
```

The above code demonstrates the creation of two datasets when the simulation begins. The dataset *bus_arbiter* is created without timeset association, and the dataset *pci_perf* is created to have association with the *pci_clk* timeset, which is assumed to have been previously defined via a **mat_timeset_define()** command call.

Resource Utilization Assessment

This category handles the extraction of resource utilization data from the simulation environment.

Resource State Measurement

This command captures the current state of a resource, and stores it on the specified axis in a predefined dataset, along with the current simulation time and timeset time (if associated with the dataset.)

HDL Syntax:	**mat_set_state** (<*dataset*>, <*axis*>, <*state*>)	
C Syntax:	**mat_set_state** (<*dataset*>, <*axis*>, <*state*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the state data will be stored.
	state (string)	Name of the state to apply to the axis.
	sim time (floating point)	Simulation time of data capture.

Table 4 – Command Syntax of mat_set_state()

Example Usage

The below example Verilog code shows how the arbitrary state of a resource is set as the state of the resource changes over time.

```
always @(posedge clk)
begin
  case (current_state)
    `IDLE :
     begin
       $mat_set_state (“pci_perf”, “tx_state”, “Idle”);
     …
     end
    `REQUEST_BUS :
     begin
       $mat_set_state (“pci_perf”, “tx_state”, “Arbitrate”);
     …
     end
    `START_TRANSFER :
     begin
       $mat_set_state (“pci_perf”, “tx_state”, “Data Transfer”);
     …
     end
  …
  endcase
end
```

The above code fragment depicts a finite state machine implementation. Each of the three states listed within the Verilog **case** structure set the state of the *tx_state* axis to reflect the state of the *current_state* variable within the model. Because the state is captured along with the simulation time (and timeset time, if applicable), the amount of time spent in each state during simulation is automatically calculated.

Resource Performance Assessment

The commands in this category relate to the extraction of resource performance data from the model. There are a variety of commands provided to enable the data extraction in a manner that is most intuitive for the user. These commands support the capture of simple and composite numerical data, and user-defined event data. All data points generated will have the simulation time appended, as well as the timeset time of the dataset (if associated with the specified dataset.)

Simple Value Measurement

This is the most basic numerical measurement command, which receives a passed in floating point value, and becomes the created data point in the MAT data model.

HDL Syntax:	**mat_measure** (<*dataset*>, <*axis*>, <*value*>)	
C Syntax:	**mat_measure** (<*dataset*>, <*axis*>, <*value*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the data will be stored.
	value (floating point)	Numerical value.
	sim time (floating point)	Simulation time of data capture.

Table 5 – Command Syntax of mat_measure()

As shown in Figure 7 below, the floating point value passed in via **mat_measure()** is directly translated into the data point created, without any calculation or manipulation. Also note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

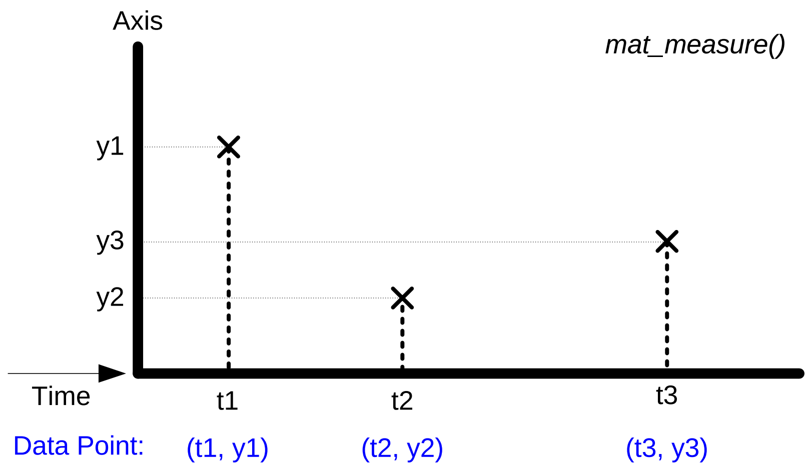

Figure 7 – Depiction of Simple Value Measurements

Example Usage

The below example Verilog code shows how direct value measurements are made.

```
always @(posedge capture_clk)
begin
   …
   // Capture the current ADC output from the signal adc_out.
   $mat_measure (“adc”, “adc_output_value”, adc_out);
end
```

Incremental Value Measurement

This is a slightly advanced numerical measurement command. The floating point value passed in is added to the value of the previous data point, and the sum becomes the new data point. Thus, the value passed in creates an incrementally modified value with respect to the previous data point.

HDL Syntax:	**mat_measure_sum** (<*dataset*>, <*axis*>, <*value*>)	
C Syntax:	**mat_measure_sum** (<*dataset*>, <*axis*>, <*value*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the data will be stored.
	value (floating point)	Numerical value.
	sim time (floating point)	Simulation time of data capture.

Table 6 – Command Syntax of mat_measure_sum()

As shown in Figure 8 below, the floating point value passed in via **mat_measure_sum()** is translated into a data point via use of the previously captured data point. Also note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

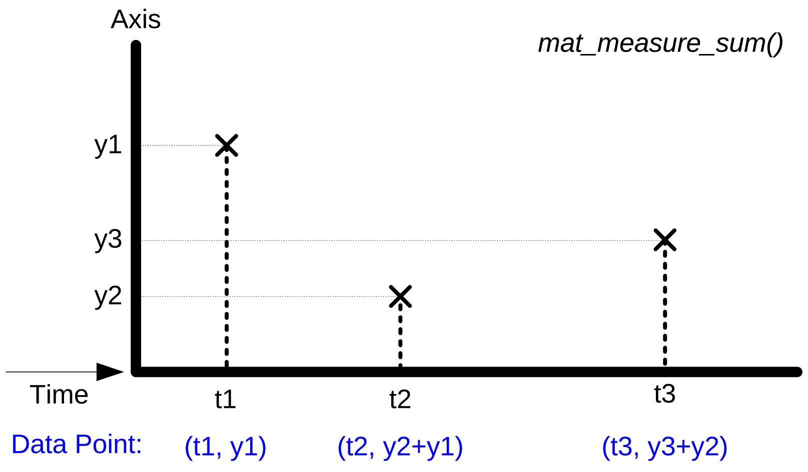

Figure 8 – Depiction of Incremental Value Measurements

Example Usage

The below example Verilog code shows how accumulated value measurements are made.

```
always @(posedge capture_clk)
begin
   …
   // Capture and accumulate the error signal output.
   $mat_measure_sum (“filter”, “feedback_err”, err_out);
end
```

The advantage to using **mat_measure_sum()** over **mat_measure()** in this case is that the summation is maintained by the DCIL, and therefore an extra variable within the model is not required to store the accumulated data. If **mat_measure()** was employed in this scenario, the extra variable to maintain the accumulation would be required in the model.

Tagged Delta Measurement

This is in the category of advanced numerical measurement commands. The creation of a data point is dependent on whether a previous delta value measurement command was performed with the same tag name on the dataset axis.

(a) If no previous delta value measurement was performed with the same tag name, the passed in value is saved in the DCIL, and no data point is generated.

(b) If a previous delta value measurement was performed with the same tag name, that previous value is subtracted from the passed in value, and the result becomes the new data point.

HDL Syntax:	**mat_measure_delta** (<*dataset*>, <*axis*>, <*tag*>, <*value*>)	
C Syntax:	**mat_measure_delta** (<*dataset*>, <*axis*>, <*tag*>, <*value*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the delta value data will be stored.
	tag (string)	Tag name used for association of data on the axis.
	value (floating point)	Numerical value.
	sim time (floating point)	Simulation time of data capture.

Table 7 – Command Syntax of mat_measure_delta()

As shown in Figure 9 below, the floating point value passed in via **mat_measure_delta()** is translated into a data point via use of the previously captured data point with the same tag. Also note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

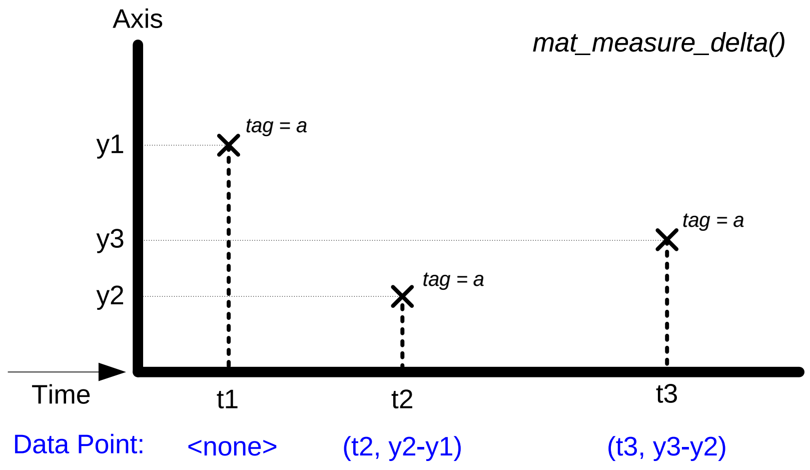

Figure 9 – Depiction of Tagged Delta Measurements

Example Usage

The below example Verilog code shows how delta measurements are made.

```
always @(posedge capture_clk)
begin
   …
   // Measure the difference between each successive maximum s_out
   // value over time.
   if (peak)
      $mat_measure_delta (“adc”, “sine_out”, “delta_max”, s_out);
end
```

In the above example, the goal is to measure the difference between each successive maximum value of a signal within the model. Since maximum values of *s_out* are attained over time, **mat_measure_delta()** uses the tag *delta_max* to retain the previous maximum value of *s_out* used in the current calculation. Thus, after the first capture of the *s_out* value, each successive capture will use the previous value to calculate the new data point in the *sine_out* axis. The *peak* model variable is assumed to be true when the maximum value is driven on *s_out*, based on the design.

If **mat_measure()** was employed to implement this same measurement, an extra variable within the model would be required to store the previous maximum value, and the delta calculation would be performed explicitly before submitting the result as the new data point.

Tagged Rate Measurement

This is in the category of advanced numerical measurement commands. The creation of a data point is dependent upon whether a previous rate value measurement command was performed with the same tag name on the dataset axis.

(a) If no previous rate value measurement was performed with the same tag name, the passed in value is saved in the DCIL, and no data point is generated.

(b) If a previous rate value measurement was performed with the same tag name, the passed in value is divided by the difference in time between the current time, and the previous data point's time. The result of that calculation becomes the new data point.

HDL Syntax:	**mat_measure_rate** (<*dataset*>, <*axis*>, <*tag*>, <*value*>)	
C Syntax:	**mat_measure_rate** (<*dataset*>, <*axis*>, <*tag*>, <*value*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the rate data will be stored.
	tag (string)	Tag name used for association of data on the axis.
	value (floating point)	Numerical value.
	sim time (floating point)	Simulation time of data capture.

Table 8 – Command Syntax of mat_measure_rate()

As shown in Figure 10 below, the floating point value passed in via **mat_measure_rate()** is translated into a data point via use of the previously captured data point with the same tag. Also note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

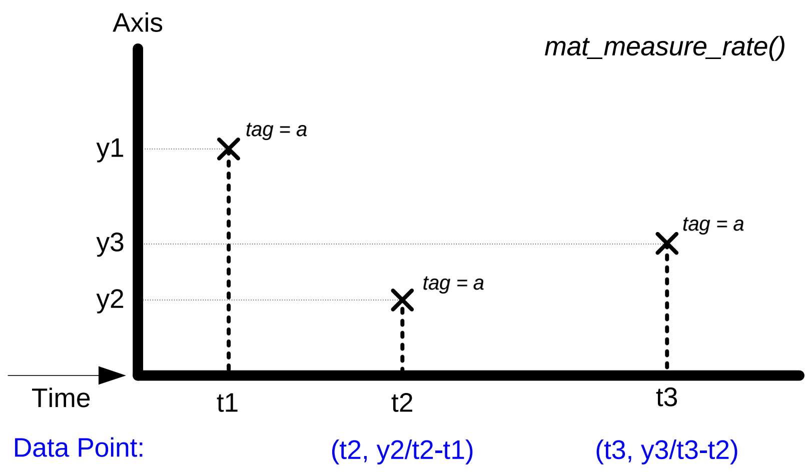

Figure 10 – Depiction of Tagged Rate Measurements

Example Usage

The below example Verilog code shows an example of rate value measurements are utilized to capture bandwidth measurements during simulation.

```
always @(posedge pci_clk)
begin
  case (transaction_state)
    `START_TX :
    begin
      …
      // Clear the tag (tx_bw) for the trans_bw axis.  This ensures
      // the first mat_measure_rate() call will establish a new set of
      // measurements.
      $mat_clear_tag (“pci_perf”, “trans_bw”, “tx_bw”);
      // Make the first measurement.  Note that the very first axis
      // value measurement for a new tag is a dummy value, since it is
      // not used in the subsequent rate calculation.  So, we use 0.
      $mat_measure_rate (“pci_perf”, “trans_bw”, “tx_bw”, 0);
    end
    …
    `END_TX :
    begin
      …
      // Measure how many bytes were transferred during this
      // transaction.  byte_count is assumed to be a register holding
      // the number of bytes transferred.  This is a measurement of
      // transaction bandwidth, or the amount of data
      // transferred over the duration of the transaction.
      $mat_measure_rate (“pci_perf”, “trans_bw”, “tx_bw”, byte_count);

      // Here, we’re measuring along a different axis, and the tag
      // (bw) is never cleared.  Therefore, the delta in time is
      // from transaction end to transaction end.
      // So this a measurement for the bytes transferred over the time
      // elapsed since the last transaction, which is an overall bus
      // bandwidth measurement.
      $mat_measure_rate (“pci_perf”, “bus_bw”, “bw”, byte_count);
      …
    end
    …
  endcase
end
```

This example depicts how **mat_measure_rate()** can be used to measure bandwidth (data rate). The axis *trans_bw* captures measurements that assess the transaction bandwidth: amount of data transferred divided by the time spent in the transaction. This is accomplished because the tag *tx_bw* on the *trans_bw* axis is cleared every time the transaction begins (see description of **mat_clear_tag()** below). Conversely, the axis *bus_bw* captures measurements that assess the overall bus bandwidth: amount of data transferred divided by the time since the last transaction ended. This is accomplished because the tag *bw* on the *bus_bw* axis is never cleared, and therefore the time delta measurement in the rate calculation includes any idle time on the bus between transactions.

Tagged Derivative Measurement

This is in the category of advanced numerical measurement commands. The creation of a data point is dependent upon whether a previous derivative value measurement command was performed with the same tag name on the dataset axis.

(a) If no previous derivative value measurement was performed with the same tag name, the passed in value is saved in the DCIL, and no data point is generated.

(b) If a previous derivative value measurement was performed with the same tag name, the previous data point's value is subtracted from the passed in value, and that quantity is divided by the difference in time between the current time and the previous data point's time. The result of that calculation becomes the new data point.

HDL Syntax:	**mat_measure_deriv** (<*dataset*>, <*axis*>, <*tag*>, <*value*>)	
C Syntax:	**mat_measure_deriv** (<*dataset*>, <*axis*>, <*tag*>, <*value*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the derivative data will be stored.
	tag (string)	Tag name used for association of data on the axis.
	value (floating point)	Numerical value.
	sim time (floating point)	Simulation time of data capture.

Table 9 – Command Syntax of mat_measure_deriv()

As shown in Figure 11 below, the floating point value passed in via **mat_measure_deriv()** is translated into a data point via use of the previously captured data point with the same tag. Also note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

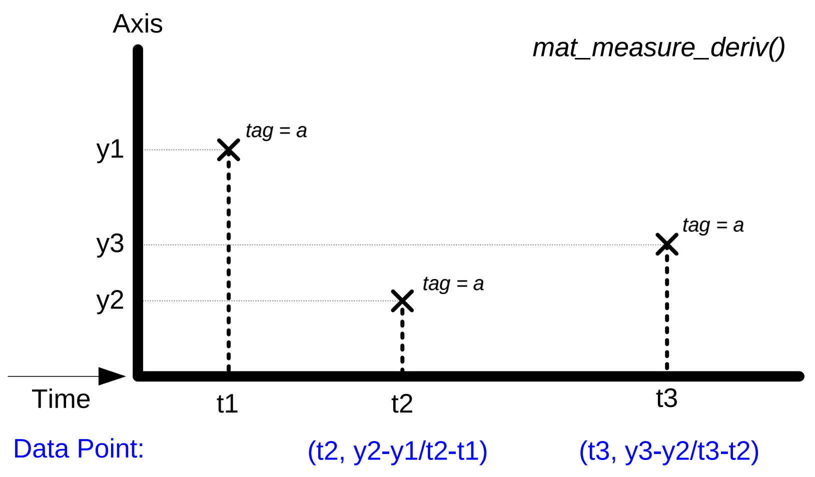

Figure 11 – Depiction of Tagged Derivative Measurements

Example Usage

The below example Verilog code shows how derivative value measurements are made.

```
always @(posedge capture_clk)
begin
   …
   // Capture the current ADC output from the signal adc_out.
   $mat_measure_deriv (“adc”, “adc_output_deriv”, adc_out);
end
```

This example shows how the **mat_measure_deriv()** command can be used to measure a discrete time derivative of the signal *adc_out*. As the *adc_out* value is captured, the previous data point (including value and time of capture) is used to create the new data point per for formula shown in Figure 11.

Tagged Time Measurement

This is in the category of advanced numerical measurement commands. The creation of a data point is dependent upon whether a previous time value measurement command was performed with the same tag name on the dataset axis.

(a) If no previous time value measurement was performed with the same tag name, the current simulation time and timeset time (if applicable) is saved in the DCIL, and no data point is generated.

(b) If a previous time value measurement was performed with the same tag name, the previous data point's time is subtracted from the current time. The result of that calculation becomes the new data point.

HDL Syntax:	**mat_measure_time** (<*dataset*>, <*axis*>, <*tag*>)	
C Syntax:	**mat_measure_time** (<*dataset*>, <*axis*>, <*tag*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the time data will be stored.
	tag (string)	Tag name used for association of data on the axis.
	sim time (floating point)	Simulation time of data capture.

Table 10 – Command Syntax of mat_measure_time()

As shown in the above table or in Figure 12 below, no value is passed in for **mat_measure_time()** because the current time (simulation and timeset time), and the previously captured data point with the same tag are the only data required to create the new data point. Note that the data point is formulated with the time that the capture occurred, which always includes simulation time, and may include the additional timeset time, if the dataset was created with timeset association.

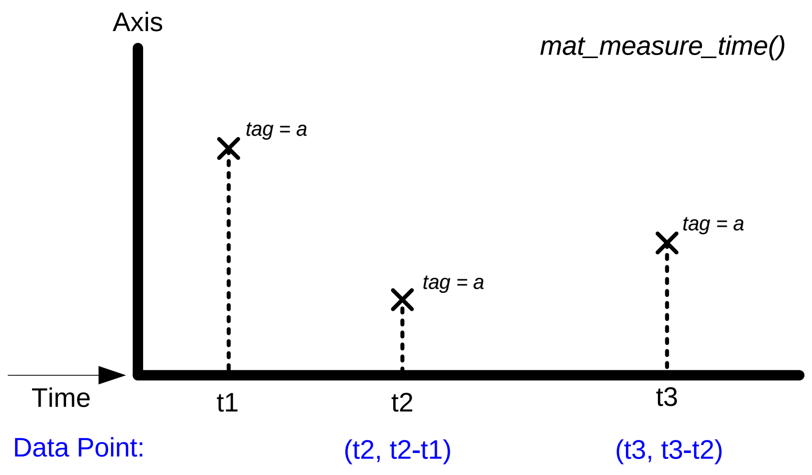

Figure 12 – Depiction of Tagged Time Measurements

Example Usage

The below example Verilog code shows how time measurements are used to assess latency.

```
always @(posedge pci_clk)
begin
  case (arbitration_state)
    `REQUEST :
    begin
      …
      // Clear the tag (lat) for the req2gnt_lat axis.  This ensures
      // the first mat_measure_time() call will establish a new set of
      // measurements.
      $mat_clear_tag (“master_perf”, “req2gnt_lat”, “lat”);
      // Make the first measurement.  Note that the first value
      // measurement for a new tag just establishes the tag and
      // first time point.  No data point is created yet.
      $mat_measure_time (“master_perf”, “req2gnt_lat”, “lat”);
    end
    …
    `GRANT :
    begin
      …
      // Grant has been received.  Measure time since request was
      // asserted.  This will create a data point for the request
      // to grant latency on the req2gnt_lat axis.
      $mat_measure_time (“master_perf”, “req2gnt_lat”, “lat”);
      …
    end
    …
  endcase
end
```

The above example shows how **mat_measure_time()** is used to capture the latency from a request to a subsequent grant. First, **mat_clear_tag()** is called to reset any internal DCIL state of the *lat* tag. Then, the call to **mat_measure_time()** establishes the time when the request is asserted. The second call to **mat_measure_time()** occurs when the grant is received, and calculates the difference in time from the original request (specified by referring to the *lat* tag) and the current time. This yields the request-to-grant latency, which becomes a data point.

Resource Event Measurement

Separate from the above numerical measurements, resource event measurements record observations during simulation. Each event data point records the name of the event, the time of observation (both in simulation time, and timeset time if applicable), and the cumulative number of observations of the event name.

HDL Syntax:	**mat_event** (<*dataset*>, <*axis*>, <*event*>)	
C Syntax:	**mat_event** (<*dataset*>, <*axis*>, <*event*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the event data will be stored.
	event (string)	Name of the event generated at the current simulation time and timeset time for the referenced axis.
	sim time (floating point)	Simulation time of data capture.

Table 11 – Command Syntax of mat_event()

As shown in Figure 13 below, **mat_event()** has been called four times during a simulation, which translated into four separate data points. Note that the event *eventA* was captured twice, and therefore the count for the latter capture at time *t4* was automatically incremented to 2 by the DCIL upon capture.

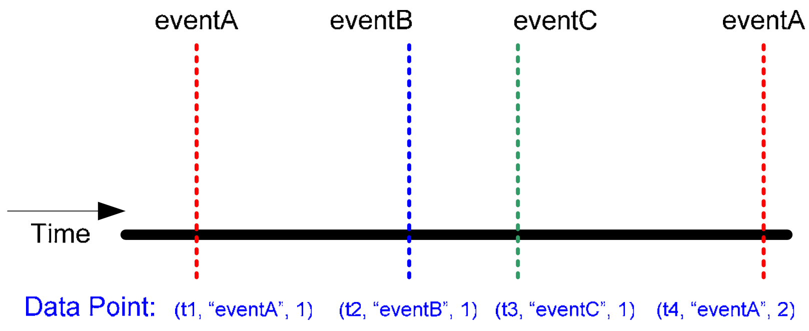

Figure 13 – Depiction of Resource Event Measurements

Example Usage

The below example Verilog code shows how events are posted to track functionality of the model.

```
always @(posedge core_clk)
begin
  …
  if (instr_cache_miss)
  begin
    // Instruction cache miss occurred.
    $mat_event (“icache_perf”, “icache_miss”, “I-MISS”);
  end
end
```

As shown in the simple example above, a resource event is posted when *instr_cache_miss* is true, presumably when the instruction cache of a processor has encountered a miss. Thus, the *I-MISS* resource event is posted every time an instruction cache miss occurs, and this data point includes the cumulative number of times the event is posted. This can enable the engineer to eventually assess when the event arises relative to other conditions during simulation, and track the number of times an instruction miss has occurred.

Resource Span Event Measurement

As an extension to the resource events described above, resource span events are dynamically grouped according to association with a tag. Each span event data point records the name of the event, the time of observation (both in simulation time, and timeset time if applicable), and the time since the previous span even was posted with the same tag.

HDL Syntax:	**mat_span_event** (<*dataset*>, <*axis*>, <*tag*>, <*event*>)	
C Syntax:	**mat_span_event** (<*dataset*>, <*axis*>, <*tag*>, <*event*>, <*sim time*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the span event data will be stored.
	span tag (string)	Tag name used for association of span events on the axis.
	event (string)	Name of the span event generated at the current simulation time and timeset time for the referenced axis.
	sim time (floating point)	Simulation time of data capture.

HDL Syntax:	**mat_span_end** (<*dataset*>, <*axis*>, <*tag*>, [<*event*>])	
C Syntax:	**mat_span_end** (<*dataset*>, <*axis*>, <*tag*>, [<*event*>, <*sim time*>])	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where the span event data will be stored.
	span tag (string)	Tag name reference to the span that will be ended.
	final event (string)	If specified, this is the name of the final span event to be generated before ending the span.
	sim time (floating point)	Simulation time of data capture.

Table 12 – Span Event Commands

The command **mat_span_event()** posts a span event on the designated dataset and axis at the current simulation time, and timeset time (if associated with the dataset.) The posted event is associated with other events in the span via the specified tag.

The command **mat_span_end()** ends a span on the designated dataset and axis, and optionally can post a span event at the current simulation time and timeset time just prior to ending the span.

As shown in Figure 14 below, four span events were posted on the *trace* axis, each within the same span via use of the span tag *tag1*. Note that the last event *eventD* may have been posted by calling **mat_span_event()** followed by **mat_span_end()**

without the optional final event, or posted via calling **mat_span_end()** alone with the optional final event specified. As each event data point is created, the time delta between the current time and previous event's time is calculated, and becomes part of the stored data point.

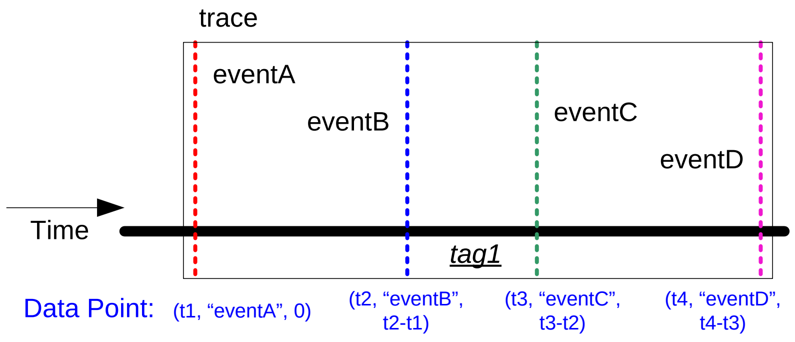

Figure 14 – Depiction of Resource Span Event Measurements

Example Usage

To illustrate how **mat_span_event()** and **mat_span_end()** enable data traces, we need to construct a small system where data flows between four functional units within a model.

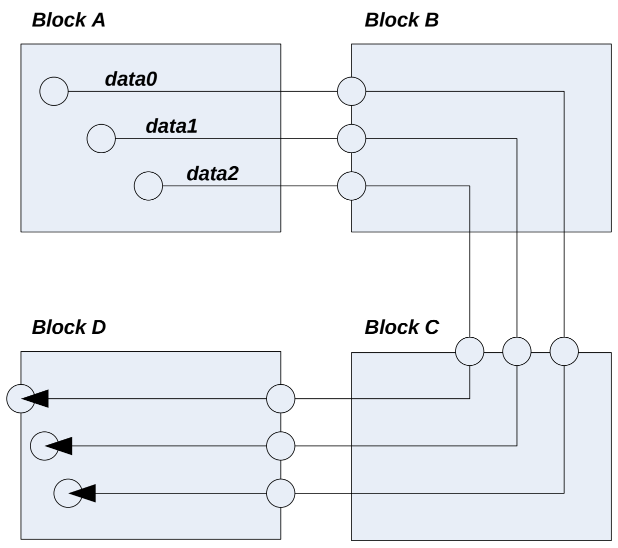

Figure 15 – Example of Data Flow Between Four Functional Units

As shown in Figure 15, four functional blocks exist named A, B, C, and D, and the data path we wish to trace originates in A, flows to B, C, and finally to D. Although there is a single data path that links the four blocks, individual datum can be pipelined such that new datum can enter the data path before previous datum terminates at block D. The circles in the diagram indicate observation points with which the data is traced with MAT. Thus, the trace begins in block A when the datum is created, and then an observation point exists as the datum arrives to block B, and similarly to block C and block D, and finally when the datum is terminated within block D.

To apply the resource span event commands to this scenario, each observation point will yield a **mat_span_event()** command within the block that specifies the dataset and axis to be used for containing the track information. In addition, each command will specify a tag, which is the internal signal or variable that contains the data. Recall that the tag is the argument that associates events into a span. Thus, the piece of datum traced through each block becomes the tag that assembles the event span. The final point at which the data trace is terminated within block D will yield a **mat_span_end()** command, which ends the span. Note that this trace example depends upon the data currently flowing in the data path to be unique. Otherwise, two spans that are destined to be distinct and disjoint may become integrated into a single long span, which is not the intention.

The below Verilog code, though incomplete, illustrates the usage of the **mat_span_event()** and **mat_span_end()** commands to accomplish the described data trace for this system.

```
module blockA (d_out)
output [32:0] d_out;

always @(posedge clkA)
  if (create_data)
    begin
      …
      $mat_clear_tag (“blocks”, “abcd_trace”, d_out);
      $mat_span_event(“blocks”, “abcd_trace”, d_out, “CREATED_A”);
    end

endmodule
```

```
module blockB (d_in, d_out)
input  [32:0] d_in;
output [32:0] d_out;

always @(posedge clkB)
  if (new_data)
    begin
      …
      $mat_span_event(“blocks”, “abcd_trace”, d_in, “ARRIVE_B”);
    end

endmodule
```

```
module blockC (d_in, d_out)
input  [32:0] d_in;
output [32:0] d_out;

always @(posedge clkC)
  if (new_data)
    begin
      …
      $mat_span_event(“blocks”, “abcd_trace”, d_in, “ARRIVE_C”);
    end

endmodule
```

```
module blockD (d_in)
input  [32:0] d_in;

always @(posedge clkC)
  if (new_data)
    begin
      …
      $mat_span_event(“blocks”, “abcd_trace”, d_in, “ARRIVE_D”);
    end
  …
  if (data_terminate)
    begin
      …
      $mat_span_end(“blocks”, “abcd_trace”, d_in, “TERM_D”);
    end

endmodule
```

Global Measurement Commands

In contrast to the prior described commands that pertain to a particular resource, commands in this category capture data related to the overall model or simulation environment. Note that the commands below do not contain a *dataset* parameter, as all global measurements belong to an internal MAT global dataset. All axes specified in the global commands implicitly belong to this global dataset.

Global State Measurement

This command captures the current state of the model or simulation, and stores it on the specified axis in the global dataset, along with the current simulation time and timeset time (if associated with the dataset.)

HDL Syntax:	**mat_global_state** (<*axis*>, <*state*>)	
C Syntax:	**mat_global_state** (<*axis*>, <*state*>, <*sim time*>)	
Parameters:	axis (string)	Name of the axis where the global state data will be stored.
	state (string)	Name of the state to apply to the axis.
	sim time (floating point)	Simulation time of data capture.

Table 13 – Command Syntax of mat_global_state()

Example Usage

The below example Verilog code shows how the arbitrary global state measurement command is used to describe the state of the model.

```
always @(posedge clk)
begin
    if (hreset)
      $mat_global_state (“hardware_reset”, “In Reset”);
    else
      $mat_global_state (“hardware_reset”, “Normal Operation”);
end
```

The above code shows that an axis *hardware_reset* is used to describe the state of the system being modeling. When the model signal *hreset* is asserted, *hardware_reset* changes to the *In Reset* state. When *hreset* is negated, the *hardware_reset* axis switches to the *Normal Operation* state. Because the *hreset* signal presumably changes during simulation to indicate that the system is either being reset or not being reset, the use of a global state to track this activity is intuitive and natural.

Global Event Measurement

Global event measurements record observations during simulation that pertain to the model or simulation environment. Each global event data point records the name of the event, the time of observation, and the cumulative number of observations of the global event name.

HDL Syntax:	**mat_global_event** (<*axis*>, <*event*>)	
C Syntax:	**mat_global_event** (<*axis*>, <*event*>, <*sim time*>)	
Parameters:	axis (string)	Name of the axis where the global event data will be stored.
	event (string)	Name of the event to be posted on the axis.
	sim time (floating point)	Simulation time of data capture.

Table 14 – Command Syntax of mat_global_event()

Example Usage

The below example Verilog code shows how the arbitrary global event measurement command is used to denote temporal partitions during simulation.

```
initial
begin
    $mat_global_event (“tests”, “Test #1”);
    do_test_1();
    $mat_global_event (“tests”, “Test #2”);
    do_test_2();
    $mat_global_event (“tests”, “Test #3”);
    do_test_3();
    …
end
```

The above code shows that an axis *tests* is used to denote when test cases are applied to the model. Presumably, each test is executed by the Verilog task *do_test_1(),* *do_test_2()*, and *do_test_3()*, and each test is preceded by a global event designating the test to subsequently run. Because each event records the time it was posted, the user knows when each test case began during the simulation.

General MAT Commands

The commands in this category pertain to management of tags, axes, and datasets throughout the simulation.

Syntax:	**mat_clear_tag** (<*dataset*>, <*axis*>, <*tag*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis with the tag to be cleared.
	tag (string)	Name of the tag to be cleared.

Syntax:	**mat_clear_axis** (<*dataset*>, <*axis*>)	
Parameters:	dataset (string)	Name of the dataset.
	axis (string)	Name of the axis where all tags will be cleared.

Syntax:	**mat_reset** ()
Parameters:	<none>

Table 15 – General MAT Commands

The command **mat_clear_tag()** allows the user to clear a tag on a specific axis, such that the next time the tag is referenced by a DCIL command, it will be as if the tag was referenced for very first time. Similarly, the command **mat_clear_axis()** behaves exactly like **mat_clear_tag()** for all tags ever referenced on the axis.

When the command **mat_reset()** is called, all axes within all datasets are have their tags cleared. This is an effective reset of the DCIL state information.

Tool Configuration

The Model Analysis Tool supports configurability via use of ASCII-text readable configuration files. MAT supports two types of configuration files, and both are formatted in XML (eXtensible Markup Language), and parsed by the DCIL when a simulation begins. These files allow the user to modify the behavior of MAT before simulation, rather than apply configurations via user commands that would require a recompilation of the simulation environment.

Global Configuration File

The MAT global configuration file is used to apply user selections for the DCIL to output data to the screen and/or *MATServ*, and to specify the alternate configuration file for dataset management. When the DCIL library is first loaded during simulation, it searches for the global configuration file named *MAT_config.xml*. An example of this global configuration file is found below.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<mat_config ver="1.0">
    <output_drivers flush_interval="5000">
        <text enable="1"/>
        <visual enable="1" ip="192.168.1.2" port="40001"/>
    </output_drivers>
    <dataset_config filename="dataset_config.xml"/>
</mat_config>
```

Table 16 – Example Global Configuration File

As shown, there are essentially two major sections of MAT configuration. The first section encompasses the *output_drivers* block, which designates the available output drivers in the DCIL. The *output_drivers* block contains a configuration for the *flush_interval*. The value assigned to this modifier determines how many MAT data objects will be queued within the DCIL before they are flushed to all the enabled output drivers. In the above example, five-thousand data objects will be enqueued before they are all flushed to the enabled output drivers. Because TCP/IP communication with *MATServ* takes time, sending each data object as it is created will immensely slow down the simulation. However, storing all data objects during simulation may require a large amount of system memory. Thus, this number should be set to an appropriate value such that the simulation is not grossly affected by data object flushes to *MATServ*, and yet the memory required to store the data objects on the simulating workstation is not proportionally large.

Within the *output_drivers* block, each output driver is configured for enablement. Currently, MAT supports the *text* output driver that displays captured data to the screen, and the *visual* output driver which routes data to *MATServ* for eventual visualization by *MATView*. Each output driver can be enabled by setting the *enable* modifier to ‘1’, or disabled by setting the *enable* modifier to ‘0’. The *visual* output driver contains two extra configurations. The *ip* field designates the TCP/IP address of the workstation where the *MATServ* server is running, and the *port* field designates the TCP/IP port number that *MATServ* is setup to listen.

The second configuration supported by the MAT global configuration file is the location of the MAT dataset configuration file. As shown in Table 16, the *dataset_config* field contains a modifier *filename* that should point to the path and filename of the dataset configuration file. If this item is not specified, no further configuration parsing will be performed by the DCIL.

Dataset Configuration File

The MAT dataset configuration file is used to fine tune which datasets or axes within datasets will be enabled for text display or visualization. In addition, the auto-generation of event axes is enabled via this file. This file is read by the DCIL after parsing of the global configuration file, which specifies the path and filename of the dataset configuration file. An example of a dataset configuration file is shown in Table 17 below.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<dataset_config>
    <dataset name="dataset1"  text="1" visual="1"/>
    <dataset name="dataset2"  text="0" visual="0"/>
    <dataset name="dataset2" axis="state1" visual="1"
             coverage_axis="cs1" coverage_count="3" />
    <dataset name="dataset2" axis="state2" visual="1"
             coverage_axis="cs2" coverage_count="4" />
    <dataset name="dataset3" axis="arb_grant" text="1" visual="1"
             coverage_axis="cvg_grant" coverage_count="3" />
</dataset_config>
```

Table 17 – Example Dataset Configuration File

As shown, only one type of configuration block exists, namely the *dataset* configuration line. Each line targets specific configuration modifiers for a particular dataset or axis within a dataset. If only the dataset is specified, all subsequent configurations on the line will apply to the entire dataset. In contrast, if a dataset and an axis is specified, all subsequent configurations on the line will apply only to the axis in the dataset. In addition, since this dataset configuration file is parsed from top to bottom, any configurations that refer to the same dataset and/or axis as previously configured above will override the above setting. For example, *dataset2* is first configured with both the *text* and *visual* output drivers disabled. Because no axis is specified, this configuration applies to all axes created within *dataset2*. However, subsequent configuration lines enable the visual output driver for axes *axis1* and *axis2*, so these axis settings override the previous disablement of all output drivers for *dataset2*.

For most dataset configuration lines, two configuration modifiers are available to set the enablement of the output drivers on a dataset basis, or dataset and axis basis. The *text* modifier is set to ‘1’ for to enable screen display of information pertaining to the specified dataset, or dataset axis. Likewise, the *visual* modifier is set to ‘1’ to enable data object transmission to *MATServ* for the specified dataset, or dataset axis. If either modifier is set to ‘0’, the corresponding output driver is disabled. If the MAT global configuration file disables an output driver, any enablement settings for that output driver in the dataset configuration file are ignored.

Automatic State Coverage Axis Generation

For axes that will contain resource state data points, which are generated by the **mat_set_state()** command, additional configuration modifiers are available in the dataset configuration file. The *coverage_axis* and *coverage_count* modifiers of a dataset configuration line are used in conjunction to enable the auto-creation of an axis, and generation of resource events on that axis based on changes on the state axis. The *coverage_axis* modifier specifies the name of the event axis to be automatically created, and used for the DCIL-generated events. Additionally, the *coverage_count* modifier specifies the number of states on the state axis (including the current state) to be used for formulate the event posted on the *coverage_axis*. The benefit of this built-in feature is that events can automatically published that show the state transition coverage of a resource. This can aid in the analysis of resource state transitions, versus the more basic analysis of tracking individual states.

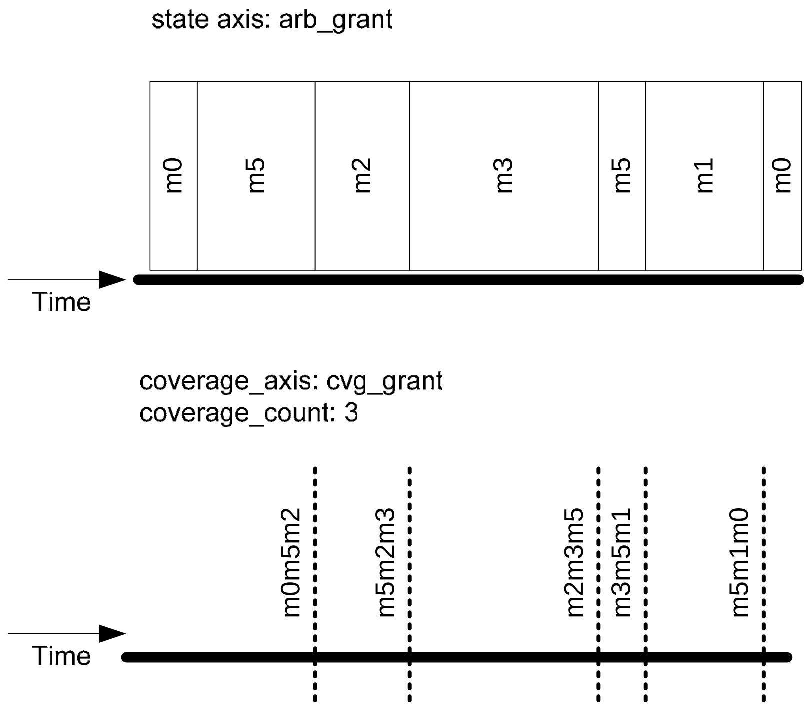

Figure 16 – Example State Axis and Auto-generated Coverage Axis

Referring to the last configuration line for *dataset3* in Table 17, and example depicting this functionality is shown in Figure 16 above. The state axis *arb_grant* consists of state data points captured during a simulation via use of the **mat_set_state()** command. Because the dataset configuration file stipulates that a separate coverage axis is to be maintained for this state axis, the *cvg_grant* axis will be created by the DCIL. Additionally, the configuration specifies that the three most recent states will be concatenated to become the posted event on the *cvg_grant* axis. Thus, when the third state transition is encountered on the *arb_grant* axis, the DCIL automatically concatenates the current and previous two states to formulate an event string, which is posted on the *cvg_grant* axis. This action occurs for every subsequent state change on the *arb_grant* state axis until the simulation completes.

MATSERV DATA SERVER

MATServ provides online storage for data objects generated during simulation by the DCIL. The *MATServ* data server is an executable application that may reside on the same workstation as the executing simulation, or on a different workstation.

When the simulation is initiated, the DCIL attempts to establish a connection with the *MATServ* server at the TCP/IP address and port number designated in the MAT global configuration file. If the connection cannot be established, the DCIL will wait until the connection is made. When the network connection is made, the DCIL allows simulation to proceed, and *MATServ* will begin to receive captured data objects over this connection.

To start the *MATServ* server, the executable binary needs to be called with command-line parameters to define the TCP/IP port numbers to listen for connection requests from both the DCIL and *MATView*. An example command-line execution of *MATServ* is found below:

```
matserv –portin 40000 –portout 40001
```

The command-line switch *-portin* denotes the TCP/IP port number that will be monitored for incoming connections from the DCIL. Similarly, the *-portout* switch denotes the port number monitored for connections from *MATView*. The port numbers must not be the same.

When executed, the server runs in the foreground and waits for connection requests from either the *DCIL* or *MATView*. Information is displayed on the screen when connections are established or dropped. The server can be exited by hitting *Control-C*, which will delete all stored data objects on the server.

MATView Visualization Client

The *MATView* visualization client is the interface between the *MATServ* and visualization of the data captured during simulation. *MATView* downloads data objects stored by *MATServ*, and translates them into the OpenDX data model which is exported to a file for use in the OpenDX *Data Explorer* [3] visualization application. *MATView* is executed via the following syntax:

```
matview –server 192.168.1.1:40001
```

When the *MATView* application executes, it first attempts to connect with the *MATServ* data server using the command-line specified TCP/IP address and port number to which the server is listening. *MATView* will wait until the connection is established, and then start communication by requesting a list of datasets for which data objects currently reside on *MATServ*. This allows *MATView* to determine which datasets have data objects currently available for download. *MATServ* also provides the current number of data objects available for each dataset. *MATView* uses this information to determine how many new data objects are available since the last communication with *MATServ*. If new objects are available, they are downloaded by *MATView* via the network connection, and recreated into memory-allocated MAT data objects. Subsequently, the objects are translated into the OpenDX data model. Once all data objects are received from *MATServ*, the OpenDX data model is exported to a text data file. This file is read by the OpenDX *Data Explorer* application to load all the data points created during simulation and allow manipulation and visualization via the *Data Explorer* tool.

Visualization Using OpenDX Data Explorer

After *MATView* has written the OpenDX data model into a text file, it can then be opened via the OpenDX *Data Explorer* application. *Data Explorer* is a front-end graphical user interface (GUI) responsible for data import, manipulation of the data model, and highly-configurable visualization via a data-driven graphical programming language. Since *MATView* exports a file containing a flattened OpenDX data model representation of the captured data points from simulation, *Data Explorer* only needs to import this file, and all datasets and axes become available by the original names used when created in MAT.

An in-depth description of the open-source *Data Explorer* application is beyond the scope of this document. Although the *Data Explorer* application is employed in this current release of MAT to ultimately handle the visualization of the collected data, future releases will encapsulate the visualization in a *MATView* GUI, and enable a more unified and user-friendly interface for manipulating and displaying the captured simulation data. As an example of the type of visualizations available, Figure 17 below shows an actual plot created in *Data Explorer* with data captured by MAT.

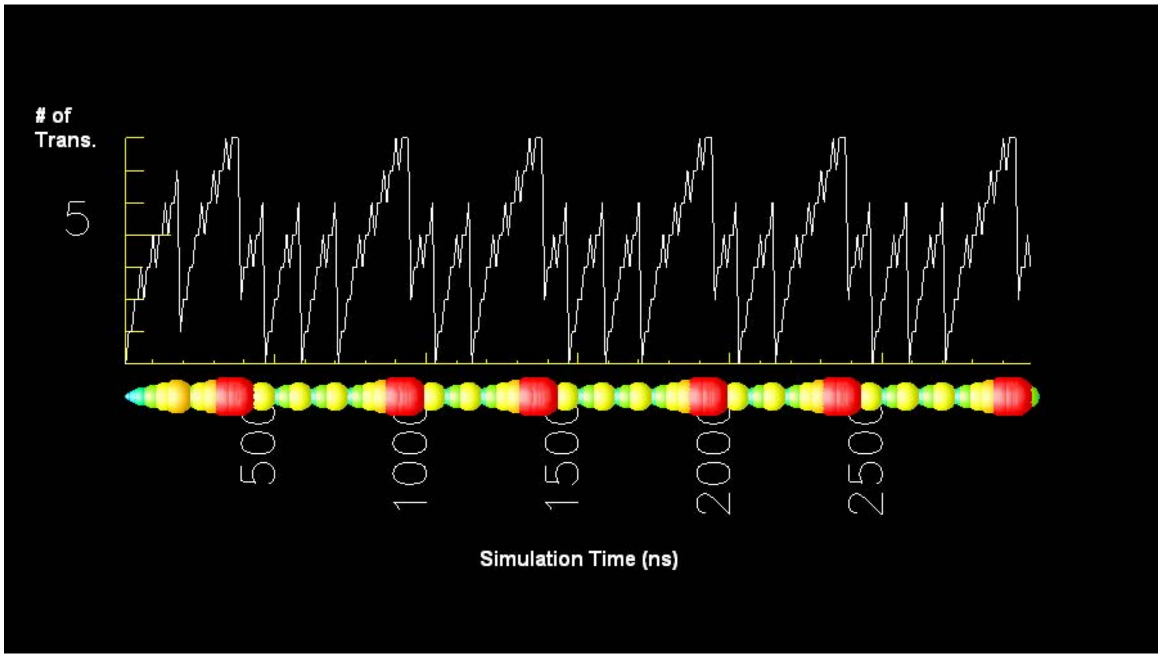

Figure 17 – Example MAT Output Viewed in OpenDX *Data Explorer*

The data rendered in the above visualization was gathered during a SystemC simulation where the number of bus transactions for a particular system interface was monitored over simulation time. The figure shows two variations of representing the same data, one overlaid upon the other. The line representation on top displays a standard two-dimensional graph of the data, plotted as bus transactions per unit time. Below the graph is an alternate colored glyph representation, where each data point in time is depicted as a sphere. As the value of each data point changes over time, the size of sphere changes proportionally, and the color shifts in spectrum to aid the visualization. Thus, as the values increase on the y-axis in the top graph, the spheres become larger, and shift from blue to green, yellow, orange, and red. The advantage of the colored glyph representation is that the sphere sizes and colors are normalized to the data, such that maximum value data points will colored in red, and minimum data points will be colored in blue. If multiple plots are observed together, the engineer can easily find low and high values measured over time by simply observing the glyph color.

CHAPTER 4

Model Analysis Tool Implementation

OVERVIEW

The Model Analysis Tool software is comprised of three separate entities that each work together to enable data capture from the simulation environment, store and maintain the MAT data model representation on a network server, and visualize the captured data. Respectively, these entities are named the *Data Capture Interface Library (DCIL)*, *MATServ*, and *MATView*. Each library or application is written in C++ using an object-oriented design methodology to maximize reuse and enable future expansion by clear partitioning of class roles and responsibilities. This chapter is dedicated to an explanation of the MAT implementation for each one of the entities that make up the Model Analysis Tool.

DATA CAPTURE INTERFACE LIBRARY

The DCIL is the interface between the simulation and *MATServ*, providing the user with all the commands delineated in the previous chapter. The construction of this interface library began with a logical partitioning of functionality, which translated into an object-oriented class organization. The functional responsibilities of the DCIL are listed in Table 18 below, alongside the abbreviated name used thereafter to reference each functional responsibility.

Functional Responsibility	Abbreviated Name
1. Provide the command interface to the user.	Command Interface
2. Provide a means of run-time configuration.	Configuration Interface
3. Provide means of routing user commands to the appropriate internal execution unit.	Command Bridge
4. Provide individual execution units to handle user commands.	Execution Units
5. Provide means of conversion of the captured raw data from the simulation environment into the internal MAT data model.	Data Model
6. Provide a standardized and extensible mechanism by which MAT data model objects flow out of the DCIL.	Output Interface

Table 18 – DCIL Functional Responsibilities

Command Interface

The DCIL *Command Interface* is comprised of command classes, and several interfaces that provide the commands to the user in the form of global function calls. Each interface is targeted for a different version of the library, depending upon the language for which the DCIL will be compiled (Verilog, C, C++, SystemC).

Each interface consists of the global functions that are called by the user, comprising the user commands available to the simulation environment. Every function creates an object of a command class, or a *command object*, which encapsulates the respective user command and associated arguments passed in via the function call. Then, the command object is submitted internally to the *Command Bridge* for routing and execution.

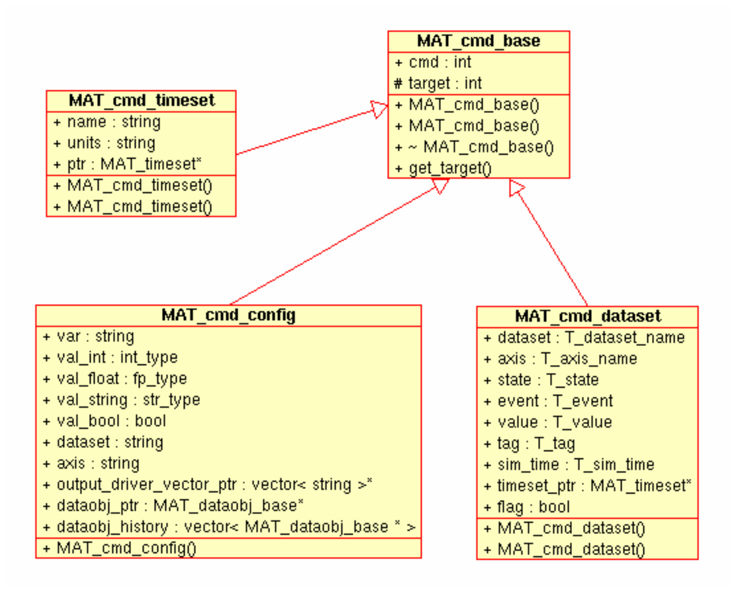

Figure 18 – UML Diagram of the DCIL Command Classes

As shown in preceding figure, all command classes derive from **MAT_cmd_base**, which is a virtual base class that provides all derived classes with a command identifier, and the target for the command. At a minimum, all command objects must set these two fields within the base class. The target defines the destination *Execution Unit* for the command within the DCIL; this will be detailed in the *Command Bridge* section below. The command identifier defines the specific command that will be executed once the command object reaches its destination. Thus, each command class acts as a template for groups of commands pertaining to a certain functional category that share common data fields. These data fields are used to pass information to the targeted *Execution Unit* within the DCIL.

There are three derived command classes that extend **MAT_cmd_base** to provide additional data fields used for supplying relevant information for each command. Command objects created from the **MAT_cmd_timeset** class are used by the timeset creation commands listed in Table 2. Command objects created from the

MAT_cmd_dataset class are used by the dataset creation and all measurement commands listed in Table 3 through Table 15. The class **MAT_cmd_config** is used to create command objects that modify the configuration of MAT on-the-fly. These commands are not accessed externally via function calls, but rather internally by the *Configuration Interface* and other units to set and read configuration data.

Configuration Interface

Besides the *Command Interface*, the *Configuration Interface* is the only other method by which information enters the DCIL. This interface is activated when DCIL is first loaded, which causes the various XML configuration files to be parsed. The parameters within these configuration files direct certain dynamic behaviors of the DCIL during simulation, as described in the previous chapter.

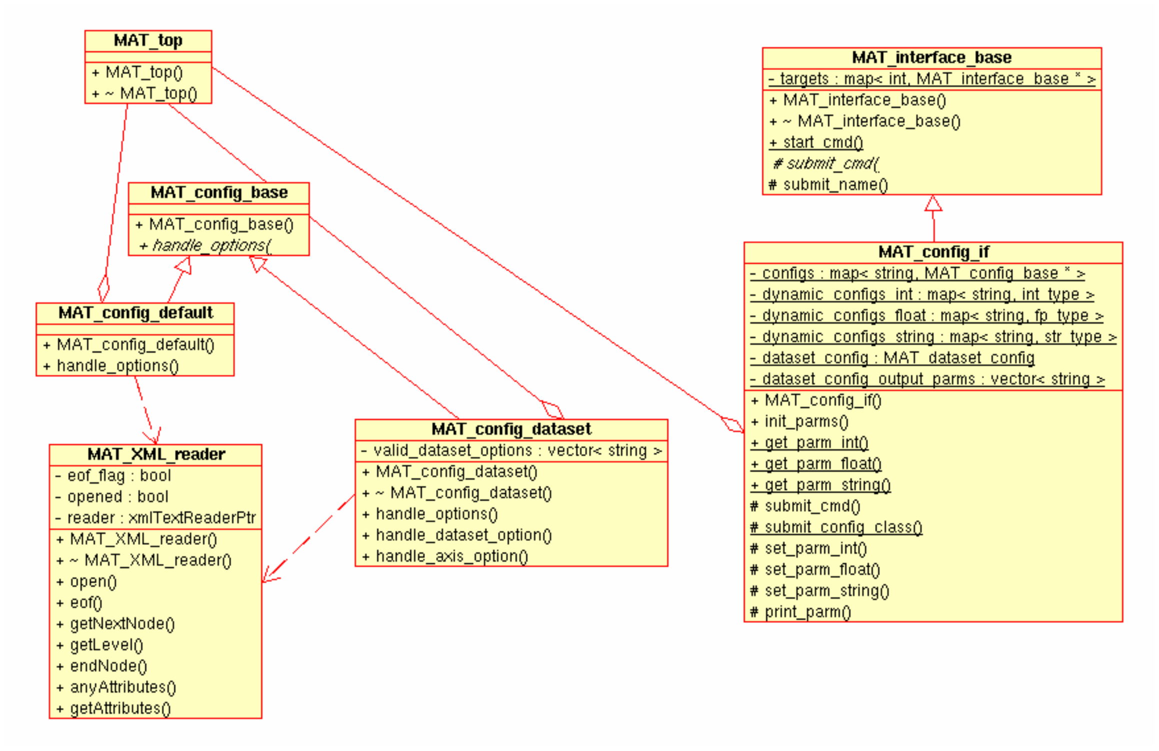

Figure 19 – UML Diagram of DCIL Configuration Classes

As shown in Figure 19 above, there are several hierarchies of classes that implement the configuration functionality. **MAT_config_base** is a virtual base class that has the **MAT_config_default** and **MAT_config_dataset** classes derived from it. Respectively, these derived classes perform the actual parsing of the main MAT configuration file, and the dataset configuration file via use of the **MAT_XML_reader** helper class. The **MAT_config_base** class maintains a class instance registry of objects that are derived from it, which is used when parameter parsing is initiated.

After the configuration *Execution Unit* class **MAT_config_if** is instantiated at start-up by the **MAT_top** top-level class, the *init_parms()* method of **MAT_config_if** is called, which cycles through all registered objects of **MAT_config_base**, calling the *handle_options()* method on each derived class. This method initiates the XML parsing of all configuration files detected, and subsequent insertion of the parameters as name-value pairs into a data structure in **MAT_config_if**. Later, when a parameter is referenced by a command object being executed by **MAT_config_if**, these same data structures are referenced to find and return the parameter value.

Command Bridge

The previously described *Command Interface* creates command objects that encapsulate the user command and associated parameters. After each command object is created, it is submitted to the *Command Bridge* to be routed for execution. The *Command Bridge* receives the command objects, and based upon their target identifier set by the *Command Interface*, the object is routed to the appropriate *Execution Unit* that will handle it.

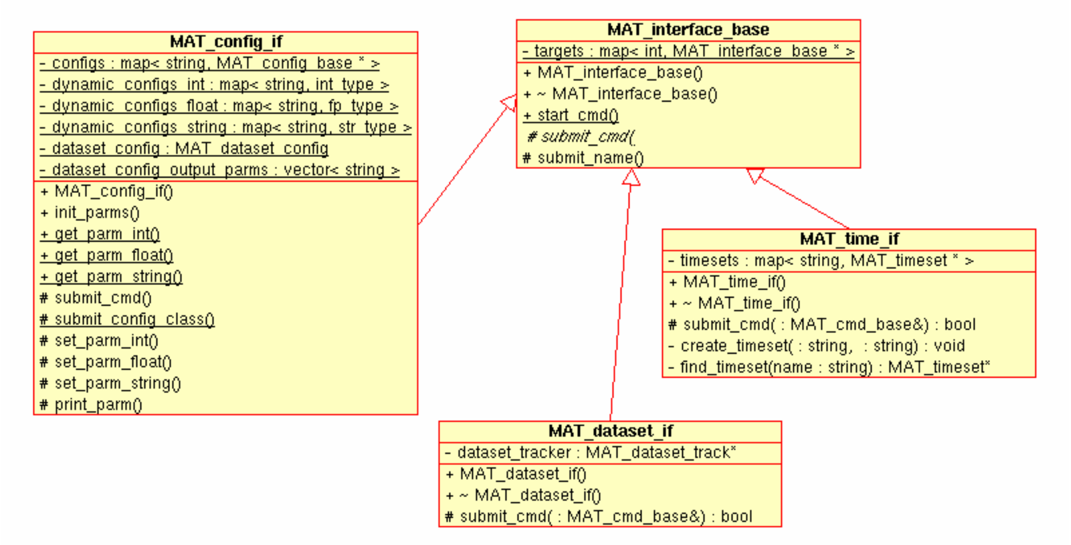
MAT_config_if
- configs : map< string, MAT_config_base * >
- dynamic_configs_int : map< string, int_type >
- dynamic_configs_float : map< string, fp_type >
- dynamic_configs_string : map< string, str_type >
- dataset_config : MAT_dataset_config
- dataset_config_output_parms : vector< string >
+ MAT_config_if()
+ init_parms()
+ get_parm_int()
+ get_parm_float()
+ get_parm_string()
# submit_cmd()
# submit_config_class()
# set_parm_int()
# set_parm_float()
# set_parm_string()
# print_parm()
MAT_interface_base
- targets : map< int, MAT_interface_base * >
+ MAT_interface_base()
+ ~ MAT_interface_base()
+ start_cmd()
# submit_cmd(
# submit_name()
MAT_time_if
- timesets : map< string, MAT_timeset * >
+ MAT_time_if()
+ ~ MAT_time_if()
# submit_cmd(: MAT_cmd_base&) : bool
- create_timeset(: string, : string) : void
- find_timeset(name : string) : MAT_timeset*
MAT_dataset_if
- dataset_tracker : MAT_dataset_track*
+ MAT_dataset_if()
+ ~ MAT_dataset_if()
# submit_cmd(: MAT_cmd_base&) : bool

Figure 20 – UML Diagram of the DCIL Command Bridge Classes

As shown in Figure 20 above, the **MAT_interface_base** is a virtual base class that currently has three classes deriving from it: **MAT_config_if**, **MAT_dataset_if**, and

MAT_time_if. When each derived class is instantiated, it calls the *submit_name()* method of the **MAT_interface_base** class to register its unique identifier and a reference to itself. Later, when the *Command Interface* creates a command object as a result of a user command call, the object is sent to the *Command Bridge* via a call to the static method *start_cmd()* in the **MAT_interface_base**. This method determines where to route the command object based upon its target identifier, which corresponds to the unique identifier specified by the derived *Execution Unit* classes during registration. If the identifier exists in the registry, the command object is sent to the *Execution Unit* via the virtual *submit_cmd()* method. If the identifier does not exist in the registry, the command is finished, and an error code is returned to the *Command Interface* designating the command did not complete successfully.

Execution Units

The *Command Bridge* routes all inbound command objects to the appropriate *Execution Unit* for handling of each command. The *Execution Units* are derived classes from the **MAT_interface_base** virtual base class, as shown in Figure 20. Each of these units have various interactions with other objects, so a discussion of each *Execution Unit* is warranted.

The **MAT_dataset_if** class is the unit responsible for handling all dataset creation and measurement commands objects. The classes utilized during the execution of these commands are shown in Figure 21.

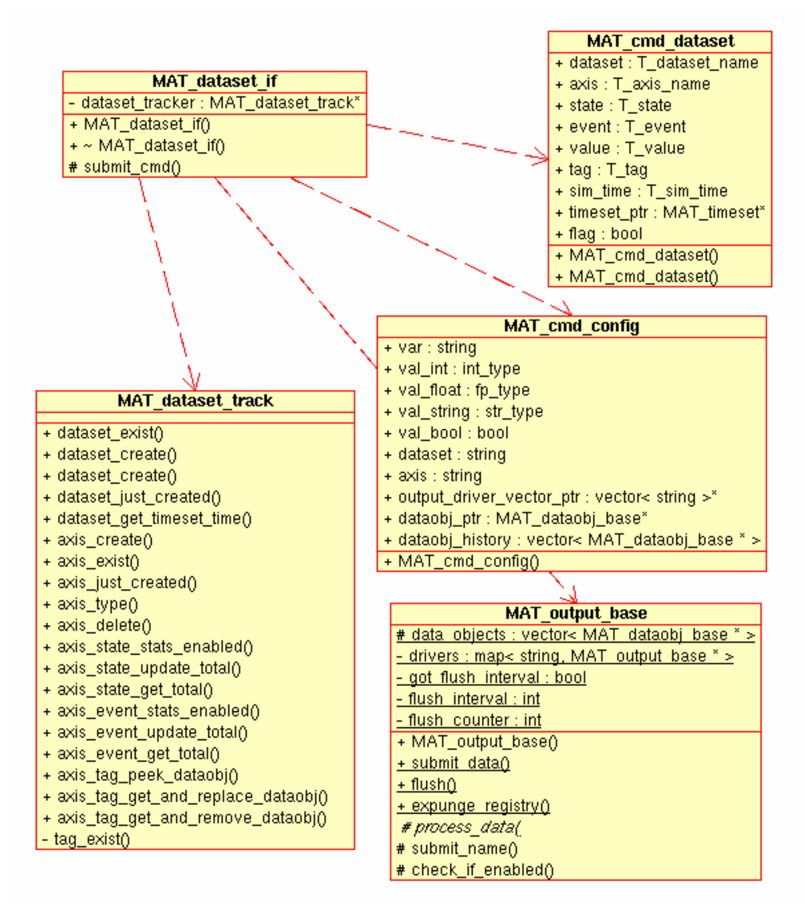

Figure 21 – UML Diagram of the DCIL Dataset Execution Unit and Related Classes

As shown in the preceding figure, the **MAT_dataset_if** class has association with a number of other classes in order to execute received dataset command objects. Command objects of the class **MAT_cmd_dataset** are received via the *submit_cmd()* method in **MAT_dataset_if**, which are subsequently parsed to determine the specific command sent in the command object. If the command object calls for the creation of a data point, the fields of the object are parsed, and the appropriate *Data Model* object is created and filled with the required data to describe the data point. The **MAT_cmd_config** class is used during some command executions to retrieve configuration parameters from the *Configuration Interface*. **MAT_dataset_track** is a data management class that enables storage of temporary *Data Model* objects for measurement commands that require use of previously created data objects. In addition, this class tracks datasets and their timeset association, as well as axes and the type of data associated with them. Finally, the **MAT_output_base** class is used via a static method call to *submit_data()* to send formulated *Data Model* objects to the *Output Interface*.

The **MAT_config_if** instance mentioned earlier in the *Configuration Interface* section is an *Execution Unit*, which manages the parsing of configuration files at DCIL load time, and responds to command objects during simulation that request parameter values. As **MAT_cmd_config** command objects are received, they are parsed by the **MAT_config_if** class to determine which specific configuration command is to be executed, and the internal configuration parameter data structures are queried to return requested parameter values (via the same command object) to the caller.

Finally, the **MAT_time_if** class is responsible for handling all **MAT_cmd_timeset** commands objects, and maintaining all timeset state information.

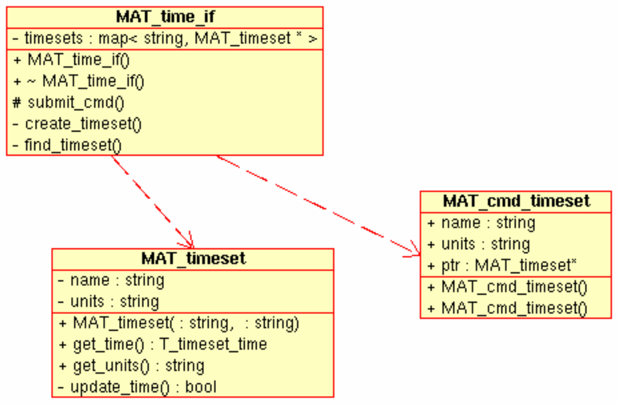

Figure 22 – UML Diagram of the DCIL Timeset Execution Unit and Related Classes

The **MAT_time_if** class instance receives **MAT_cmd_timeset** command objects from the *Command Bridge,* and subsequently parses the object to determine the specific type of command to be executed. This *Execution Unit* maintains a data structure that tracks the timeset name and the current state of that timeset via use of the **MAT_timeset** class. Every timeset is associated with its own **MAT_timeset** object, and the respective timeset time can be queried and updated (incremented) via method calls to this object. Note that when a dataset is created with timeset association, a pointer to the respective **MAT_timeset** object is obtained and used for acquiring the current timeset time directly without needing to perform the timeset name look-up via this interface.

This employed methodology of using command objects generated by a *Command Interface*, and a *Command Bridge* for routing commands to specialized *Execution Units* has the advantage of logically separating the command caller from the command handler. As new commands are added to the DCIL, additional command object classes can be derived from **MAT_cmd_base**, and new *Execution Unit* classes can be derived from **MAT_interface_base**. With the target identifier used as the link to routing command objects to the correct interfaces to operate on them, no other code within the software hierarchy needs to be modified. This yields an immediate benefit during software validation because code that was not changed does not need to be re-verified.

Data Model

The MAT *Data Model* tracks all datasets, axes, and data objects created in the dataset *Execution Unit* as a result of user commands during simulation. The *Data Model* is constructed as a set of derived classes from a virtual base class, as shown in Figure 23 below. These classes are used throughout the DCIL, *MATServ*, and *MATView* as the means by which MAT data objects are temporarily or persistently stored in each library or application.

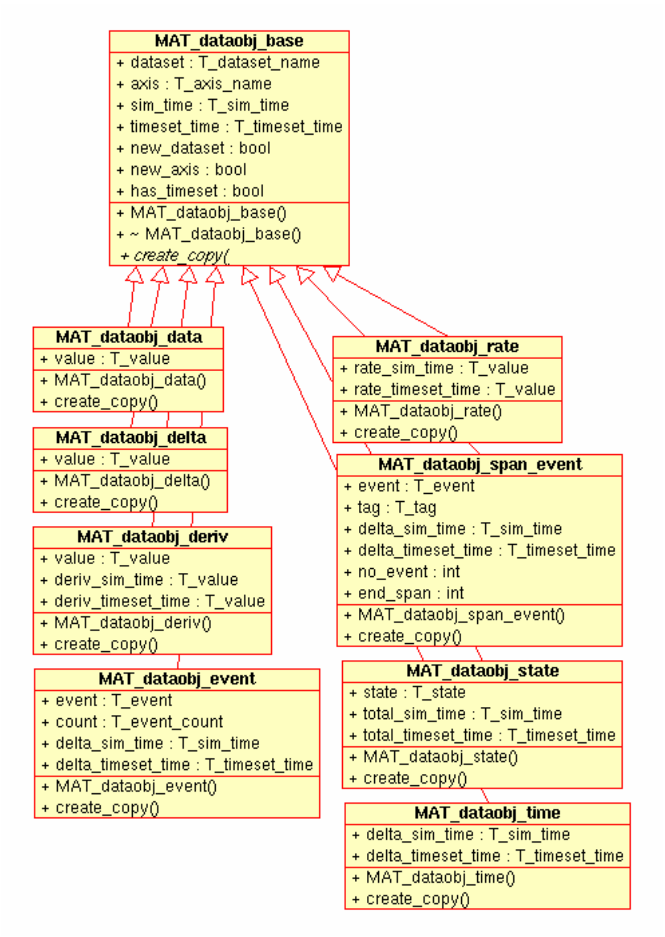

Figure 23 – UML Diagram of the MAT Data Model Classes

The virtual base class **MAT_dataobj_base** contains all the common information stored for any data point created. Every derived class contains additional fields specific to the type of data being stored. For example, the derived **MAT_dataobj_delta** class contains a field that stores the value of the data point, in addition to all the fields inherited from **MAT_dataobj_base**. Similarly, the **MAT_dataobj_event** class contains fields needed to store events data objects, such as the event name, number of times the event has occurred, and the simulation and timeset time since the previous event. The DCIL class **MAT_dataset_track** shown in Figure 21 is utilized for temporary storage of all data objects created from the above classes.

Output Interface

The *Output Interface* is responsible for receiving *Data Model* objects and translating them for subsequent serial communication on the various output drivers available.

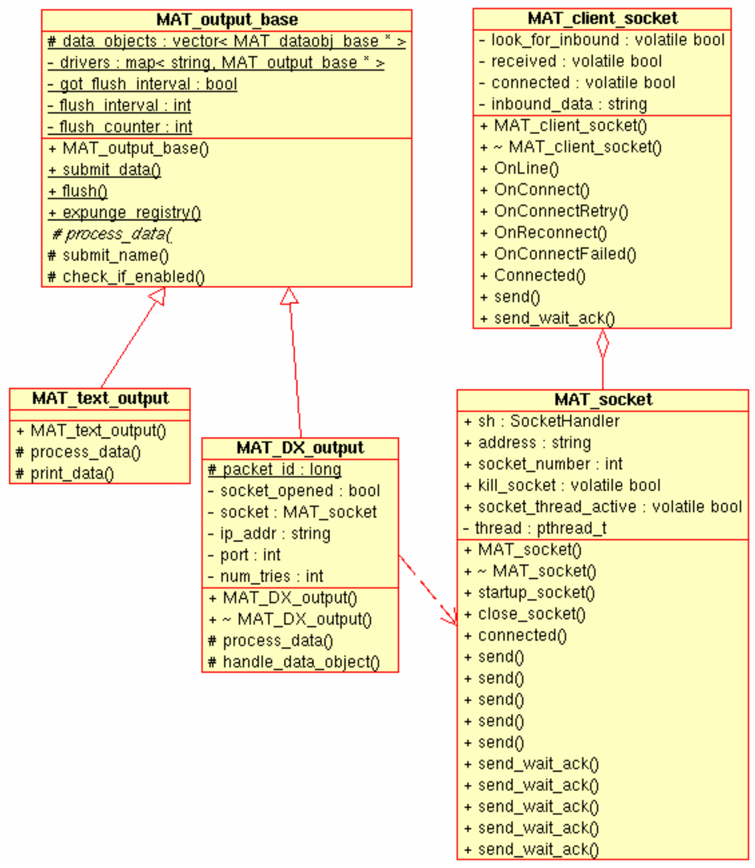

Figure 24 – UML Diagram of the DCIL Output Interface Classes

As shown in the above figure, the **MAT_output_base** is a virtual base class that receives data objects sent from the **MAT_dataset_if** *Execution Unit*. The data objects are submitted to the static *submit_data()* method in **MAT_output_base**. Once received, identical copies of the objects are created, and are queued up internally in this class until an data object count threshold is reached. The threshold is set via the *flush_interval* configuration parameter in the MAT global configuration file.

The output driver classes **MAT_text_output** and **MAT_DX_output** are derived from **MAT_output_base,** and act upon the data objects as they are flushed from the queue within the base class. At construction in **MAT_top**, each output driver class registers itself with **MAT_output_base**. In addition, **MAT_output_base** uses configuration parameters obtained by the *Configuration Interface* to determine the enablement status of each output driver. Later, when data objects are received and eventually flushed by the **MAT_output_base** class, the group of objects are sent to each registered and enabled driver in turn. Once all output drivers have been sent the group of data objects, they are dequeued and deleted from the **MAT_output_base** class.

The **MAT_text_output** driver is responsible for printing the data objects to the screen, detailing the information present for each data point. This driver is particularly useful for DCIL debugging, as well as developer verification that data objects are created as expected for given user command stimuli. The output of this driver is textually identical to debug screen output that can be generated for *MATServ* and *MATView*, which aids in ensuring that data objects are received and retransmitted without inadvertent loss of information.

The **MAT_DX_output** driver is responsible for serializing and transmitting data objects over a network channel to *MATServ* for online storage. When this driver receives a group of data objects from the base class, each object is parsed to determine the object type, and then it is sent field-by-field over the network channel. The **MAT_socket** helper class is used to accomplish the network transfer via ASCII text transfer over a TCP/IP socket. This class interfaces with an open-source TCP/IP sockets library to accomplish the network communication. The **MAT_DX_output** driver and *MATServ* observe a meta-protocol for transmission of data objects, which includes the ability to retry transmissions if data was not received properly by *MATServ*.

MATSERV

MATServ is the interface between the DCIL and *MATView*, providing online storage of received data objects during simulation, and enabling *MATView* to retrieve data objects at its own desired rate.

The construction of this application began with a logical partitioning of functionality, translating into an object-oriented class organization. The functional responsibilities of *MATServ* are listed in Table 19 below, alongside the abbreviated name used thereafter to reference each functional responsibility.

Functional Responsibility	**Abbreviated Name**
1. Provide a TCP/IP socket interface for inbound data object transmission from the DCIL.	Inbound Interface
2. Provide a means of persistent data object storage.	Data Storage
3. Provide a TCP/IP socket interface for outbound data object transmission to *MATView*.	Outbound Interface

Table 19 – *MATServ* Functional Responsibilities

The entire class hierarchy for *MATServ* can be found in Figure 25 below, and will be referenced for subsequent discussions of functionality.

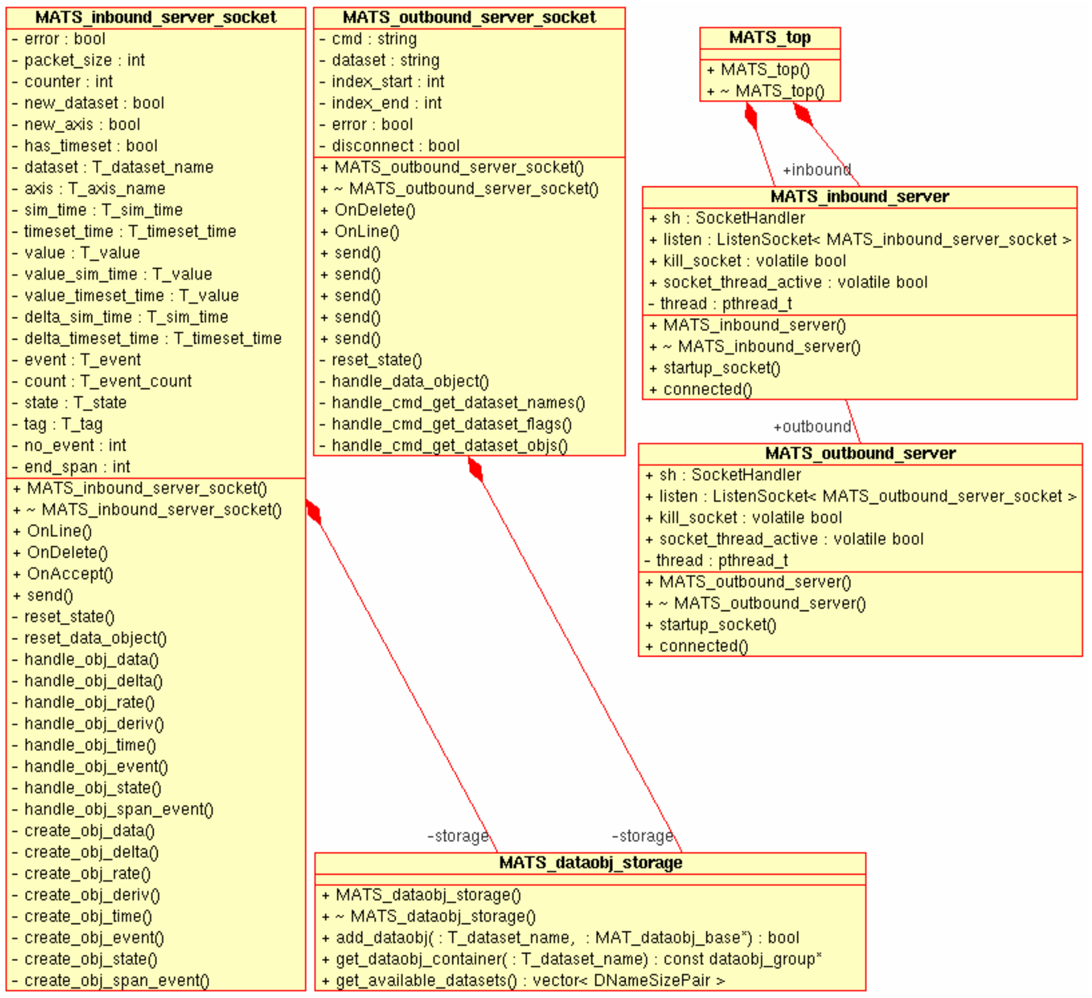

Figure 25 – UML Diagram of the *MATServ* Classes

Inbound Interface

The *Inbound Interface* is responsible for handling communication as a network server for the DCIL to receive inbound serialized data objects, and translating them into memory-allocated data objects with the appropriate class designation. As shown in the above figure, the class **MATS_inbound_server** is the interface instantiated by the application top-level **MATS_top** class to handle the receipt of data objects from the DCIL. This class instantiates **MATS_inbound_server_socket** as the actual TCP/IP socket server, and establishes this server as a free-running thread. When the DCIL *Output Interface* communicates with this *Inbound Interface*, data objects are streamed as ASCII text across the network socket, and this free-running server thread captures the text and translates it into the original data objects. The newly created data objects are then sent to the *Data Storage* repository.

Data Storage

The *Data Storage* interface is responsible for maintaining all received data objects from the *Inbound Interface*, and providing a mechanism for retrieval by the *Outbound Interface*. The **MATS_dataobj_storage** class implements the storage, and also maintains a list of the datasets with data objects currently in storage. This information is eventually used by *MATView* to determine which datasets have data points available for plotting.

As each data object is translated by the *Inbound Interface*, it is submitted to the static instantiation of **MATS_dataobj_storage** via the *add_dataobj()* method. This is the only method by which data objects are inserted into storage. Later when data objects are retrieved by the *Outbound Interface*, the *get_available_datasets()* and *get_dataobj_container()* methods are used to query the database and retrieve objects, respectively.

Outbound Interface

Over the course of a simulation, data objects arrive into *Data Storage* and are subsequently available for retrieval by *MATView*. The *Outbound Interface* is a network server that handles requests from *MATView*, and subsequently decomposes and transmits data objects over a TCP/IP socket. The **MATS_outbound_server** class instantiates **MATS_outbound_server_socket** as an object in a free-running thread that continually monitors for inbound socket connections from *MATView*. Requests may arrive for the availability of data objects, and this interface will respond with the current datasets that have data objects in storage, and the corresponding number of data objects available for each dataset. Alternately, requests may arrive from *MATView* to receive data objects from a specific dataset. In this case, *MATView* specifically requests the dataset, and the number of objects from the dataset. In this way, *MATView* can track how many objects have been received between update requests to *MATServ*, and thus it can incrementally acquire new data objects as they become available in the *Data Storage*.

MATVIEW

MATView is the visualization front-end application which communicates with *MATServ* to obtain data objects, and produces data usable by OpenDX for subsequent visualization and manipulation.

The *MATView* application has several functional responsibilities which translate into an object-oriented implementation. These responsibilities are listed in Table 20 below, alongside the abbreviated name used thereafter to reference each functional responsibility.

Functional Responsibility	**Abbreviated Name**
1. Provide a TCP/IP socket interface for inbound data object transmission from *MATServ*.	Inbound Interface
2. Provide a facility for translation of MAT data model objects to the OpenDX data model.	Model Translation

Table 20 – *MATView* Functional Responsibilities

Inbound Interface

The *Inbound Interface* is responsible for receiving serialized data objects from *MATServ* via a TCP/IP network connection, and translating them into memory-allocated data objects with the appropriate class designation.

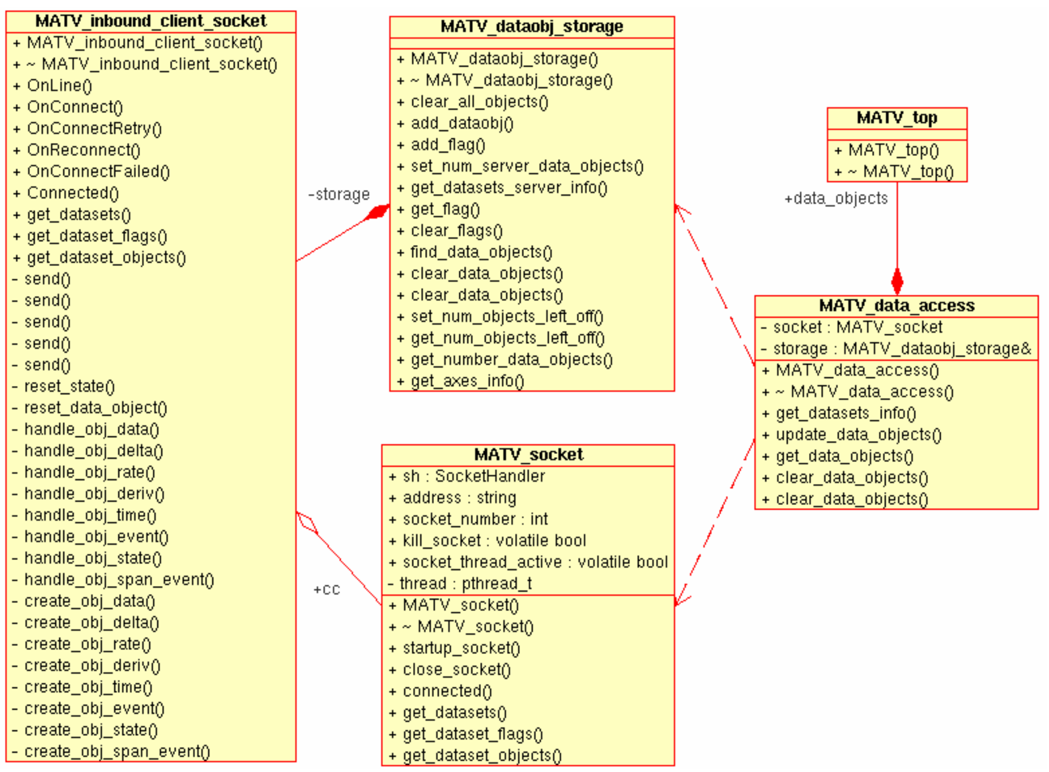

Figure 26 – UML Diagram of the *MATView* Inbound Interface Classes

As shown in the above figure, the **MATV_data_access** class is instantiated by the top-level class for this application, **MATV_top**. The **MATV_data_access** class provides methods by which the *Model Translation* interface will retrieve objects from *MATServ*, and manipulate the data object cache local to the *MATView* application. The **MATV_socket** class and its helper **MATV_inbound_client_socket** handle all TCP/IP socket communication with *MATServ*. MAT data model objects are stored in the static

MATV_dataobj_storage data management class, which is referenced by **MATV_data_access** for data retrieval once the download of data objects from *MATServ* has completed.

Model Translation

The *Model Translation* interface is responsible for translating MAT data objects received via the *Inbound Interface* into the OpenDX data model. This conversion process is necessary before the data is exported for use by the OpenDX visualization tool.

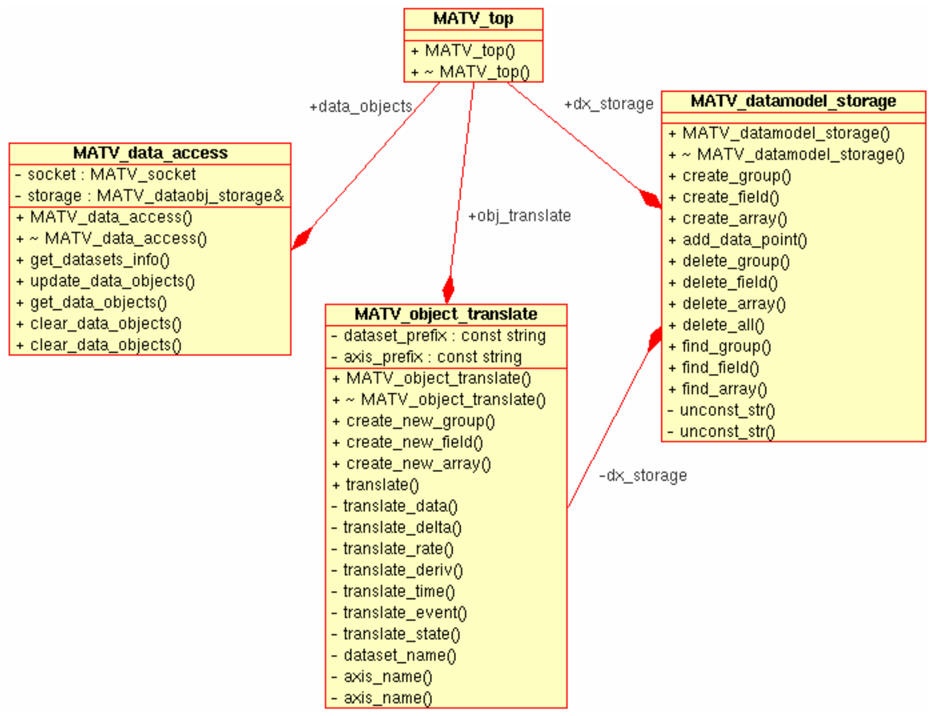

Figure 27 – UML Diagram of the *MATView* Model Translation Classes

As shown in the above figure, **MATV_top** has a reference to the same **MATV_data_access** object used for MAT data object storage after retrieval from *MATServ*. The implementation in **MATV_top** retrieves all data objects of a specified dataset, and then uses the **MATV_object_translate** class to perform a translation of each data object into the OpenDX data model. The OpenDX data model objects created after translation are stored in the **MATV_datamodel_storage** class object. Finally, after the **MATV_top** implementation retrieves all available MAT data objects for all datasets present in *MATServ*, and subsequently translates them to OpenDX data objects, the OpenDX data model is exported to a text file that can be read by the OpenDX *Data Explorer* for visualization and manipulation.

OpenDX Data Model

The data model that is centric to the OpenDX libraries and *Data Explorer* application is organized in a hierarchical fashion. The hierarchy is analogous to the hierarchy of the MAT data model. The concept of a MAT *dataset* that contains related data on separate axes corresponds to an OpenDX *Field*. Similarly, the MAT *axis* corresponds to an OpenDX *Array*, which is a component member of a field. Thus, each axis within a dataset corresponds to an OpenDX array within a field. In addition, the simulation time and timeset time axes in a dataset which are maintained by the DCIL during simulation correspond to additional and separate OpenDX arrays in the field.

OpenDX arrays are simply an ordered collections of individual datum, comparable to a simple C-language array. MAT utilizes the arrays within a field in a positional fashion, such that all elements in all arrays for a specific index are potentially related. The simulation time array component of a field contains all the times during simulation when a piece of data was captured on any axis within the dataset. For some positions within the simulation time array, every other array within the field will have a piece of data to correspond with that simulation time, which will be located in the same array index as the simulation time. This scenario occurs when data was collected for every axis in a dataset during a given simulation time. For other positions within the simulation time array, not all arrays will contain a data value that corresponds to the simulation time. This scenario occurs when not all axes in a dataset have captured data for a given simulation time. In this case, additional arrays in the field are used to denote that a particular position for an array is invalid, indicating that no data exists for the corresponding data array index position. Thus, the additional array in the same field, called an *invalids array*, is associated with a data array that contains captured data values, and indicates whether each position within the array has valid or invalid data. Typically there is one invalids array for every data array in a field.

CONCLUDING REMARKS AND FUTURE WORK

The development of this project from original concept to implementation and subsequent documentation in this report consisted of nearly one year of work, mostly taking place in the evenings. The original concept was defined and documented in a specification, which was refined via many iterations of thought long before implementation began. Rather than ascribing to the notorious "ready-fire-aim" methodology of software development, the process of completely specifying the tool functionality up front was employed, and undoubtedly saved many hours of rework that would have ensued after dead-end paths were finally realized. Conducting thought-experiments, imagining real-world usage scenarios, devising the methodology for an object-oriented class organization, and researching existing graphical visualization tools contributed to the refinement process, which lasted nearly four months. In retrospect, four months of planning was not long enough, but there were bounds on the project imposed by the desire to eventually graduate.

The remainder of the time spent on this project encompassed the software implementation. During that time, several meetings with my advisors proved beneficial as I provided the status of my progress, as well as received feedback that brought to light facets of the project that I did not originally examine. Admittedly, one portion of the project that was not completely conceived before implementation began was the partitioning of the tool into three separate entities (DCIL, *MATServ*, *MATView*). Only during the implementation phase and subsequent self-deliberation of how to drive the data visualization interface did I realize that the tool needed to be partitioned into at least two asynchronously connected pieces. In fact, I eventually determined that the tool should be partitioned into three separate pieces, each connected via a network interface to allow for a distributed usage model. This caused a substantial progress interruption as the network programming aspect of the tool was researched. After nearly a half-month of research and testing, an open-source TCP/IP C++ Sockets library was selected. Though fairly stable, several iterations of bug reporting and library fixes were required before it could be used in my application. The saving grace was that the library developer was more than willing to quickly address all the problems I encountered during its integration into my application; otherwise, development progress could have grinded to a halt.

There are plans for additional upgrades and refinement of the Model Analysis Tool after this report is published. New hardware description languages and electronic system level languages will be investigated for MAT support. Additionally, the complete integration of the OpenDX tool with *MATView* is a key upgrade slated to occur. This will remove the need to run OpenDX *Data Explorer* as a separate application. Rather, the OpenDX API will be called from within the *MATView* application, and all visualization and associated manipulation will be controlled via the *MATView* GUI. This will also relieve the user from the need to learn the extensible but often overwhelming *Data Explorer* application interface, as *MATView* will hide that complexity with a simpler GUI with preconfigured visualization options.

Additionally, I plan to explore the Java language for possible use. Several features of MAT that required external library support are native to Java. Given the stabilization of the language, feature set, portability, and performance improvements of the virtual machines, Java may be a possible candidate for a language remap of a portion or the entire tool.

Finally, I envision building a testbench environment that surrounds MAT for the purpose of tool validation. Random tests consisting of DCIL command sequences are generated, applied to the tool during simulation, and stored in *MATServ*. Subsequently the testbench will retrieve the MAT data objects from *MATServ* and compares them with the anticipated results based upon the original random test case. This self-checking environment will test the path from original data capture to MAT data model conversion and storage in *MATServ*, and will enable extensive software validation for all future revisions of the tool.

BIBLIOGRAPHY

1. Jantcsh, A., and Sander, I.: 'Models of computation and languages for embedded system design', *Proc. IEE Comput. Digit. Tech.*, 2005, **152**, (2), pp. 114-129
2. Hedström, Anders (2005) *C++ Sockets Library* [Online]. 2005. Available from World Wide Web: http://www.alhem.net/Sockets/
3. *OpenDX: The Open Source Software Project Based on IBM's Visualization Data Explorer* (2005) [Online]. 2005. Available from World Wide Web: http://www.opendx.org/

VITA

Matt Genovese was born in Reading, Pennsylvania in May, 1974, the son of Michael and Maria Genovese. After several moves, his family settled in Owego, New York – a small town located along the picturesque Susquehanna river. Upon high school graduation from Owego Free Academy in 1992, he attended the Rochester Institute of Technology with a major in Computer Engineering. After completion of the five-year undergraduate program, Matt received a Bachelor of Science degree in Computer Engineering in December 1997. In January 1998 he joined Motorola in the Semiconductor Products Sector as a Product & Test Engineer for the PowerQUICC™ line of network processors. After spending over four years in that role, Matt joined the Motorola PowerPC™ *Somerset* design center to work in functional verification on PowerPC embedded processors and systems. Matt remains employed in this role, though the Semiconductor Products Sector was spun-off by Motorola in 2004, and is now Freescale Semiconductor. In 2004, Matt entered Graduate School at The University of Texas at Austin to attain a Master of Science degree in Electrical Engineering.

Matt currently lives in Austin, Texas with his wife and two children.

Permanent Address: 417 Carismatic Lane
Austin, Texas 78748

This report was typed by the author.