1 Table of Contents

2 Introduction to Model-Integrated Computing

3 MTF: ITK and SDK ................................................................. 4
   3.1 Model Editor ................................................................. 4
   3.2 System Schema Exporter Tool ........................................ 5
   3.3 System Schema Importer Tool ....................................... 6
   3.4 FACE Data Model Exporter Tool .................................... 6
   3.5 FACE Data Model Importer Tool .................................... 6
   3.6 Datatype Generator Tool .............................................. 7
   3.7 Component Shell Generator Tool ................................... 7
   3.8 Build System Generator .............................................. 8
   3.9 TSS Generator ............................................................ 8
   3.10 OS Configuration Generator ....................................... 9
   3.11 ARINC-653 Partition Generator .................................. 10
   3.12 MTF IDL Compiler ..................................................... 11
   3.13 ARINC-653 Emulator .................................................. 11
   3.14 Timing Analyzer ........................................................ 12
   3.15 Integration Harness Generation (Prototype) ................. 12

4 Modeling Tool Architecture ................................................ 14

5 Modeling Tool Interoperability ............................................ 14

6 Installing MTF .................................................................... 15
   6.1 System Requirements ................................................... 15
   6.2 Prerequisites ............................................................... 15
   6.3 Questions/Issues .......................................................... 15

7 Getting Started: How-To Tutorials ....................................... 16
   7.1 Creating a New Model ................................................... 17
   7.2 Opening an Existing Model .......................................... 17
   7.3 Importing/Exporting a Model ......................................... 17
      7.3.1 Import from XME (GME persistence format) .............. 17
      7.3.2 Export to XME (GME persistence format) ............... 17
      7.3.3 Import from System Schema ................................. 17
2 Introduction to Model-Integrated Computing

Model Integrated Computing (MIC) is an approach to the formal development of complex software systems. The Model-driven Architecture (MDA) of the Object Management Group (OMG) is a parallel approach with overlapping directions. The key idea is to use models in all phases of the development (analysis, design, implementation, testing, maintenance and evolution). Primary benefit of using models is the added abstraction layer, which enables the designers to capture relevant aspects of a target system in a high-level domain specific language. These models can then be programmatically traversed, analyzed and transformed to produce (or modify) code, other engineering artifacts, etc. In past experiences, it has been proved that the use of models not only provides a large speed up in the development process, including rapid improvements in time taken to upgrade the system between versions. Lastly, it leads to improved test coverage of the system. Generic Modeling Environment (GME) is a freely available software tool, which provides a platform for Model Integrated Computing design and development.

The following sections describe a model integrated computing tool suite for FACE software development.

3 MTF: ITK and SDK

MTF implements functionality required by the component developer (SDK) and the system integrator (ITK). Its purposes are as follows:

- Accelerate development of FACE components and systems via automation: Code Generation, Configuration Generation, etc.
- Provide a structured environment for defining components and systems that constrain the developers to the FACE standard, encouraging conformance in FACE components
- Provide tools for creation and use of tool-independent system schema documents
- Provide an infrastructure to encourage and accelerate FACE asset reuse

Experiences in using similar model-based techniques on systems with a well-defined API and a component based development model (as in FACE) have demonstrated greater than 10x improvement in developer efficiency.

MTF’s components are described in the following sections.

3.1 Model Editor

The Model Editor is a graphical editor implementing a custom Domain-Specific Language conforming to the FACE standard and its embodied concepts. It provides a mechanism for developers to define Data Models, Portable Components, Platform-Specific Services, Units of Portability, Portable Component Segments, Platform-Specific Segments, Target System Hardware, System Partitions, and the interactions and mappings between these. A detailed example of the use of the modeling editor to define components and systems is described in “Getting Started: How-To Tutorials” below.
Benefits of Modeling:

- **Clarity:** Models created with MTF’s Model Editor formally, unambiguously describe the target system from all structural/data aspects. The models are concise and complete, and described in a graphical format. This compares to English descriptions and arbitrary figures used in an IDD/SDD, which are subject to interpretation.

- **Conformance:** Grammar and Constraints in the modeling language impose FACE concepts (and support only FACE concepts). This prohibits the user from creating non-conformant system architectures. Given the models, the tools generate FACE conformant code.

- **Developer Efficiency:** The models are computer-readable, and executable via the tools described below. Developers benefit from Code Generation, documentation generation, configuration generation, and other utilities such as build systems (see below).

- **Flexibility/Customization:** The generators are template driven. As new processes are supported, the generators can be rapidly upgraded to support these processes. (e.g. Safety)

- **Target System Consistency:** Details are specified once, and used to create multiple products. This keeps products consistent, as they derive from a common source specification.

**How to Use:** See the How-To section below which describes how to create the various model types. Also see the Modeling Language Documentation for a complete list of modeling concepts.

**Manual Approach:** If a manual approach is taken, no modeling is required, but the designer is responsible for coding, configuration, documentation, and creation of a System Schema representations of the system.

### 3.2 System Schema Exporter Tool

The System Schema Exporter Tool creates a tool-independent representation of the models. Depending on the developer role, it will create Component/Unit of Portability packages (for Component Developers), Data Model packages (for data model curators/program data management engineers), Platform Specifications (for Platform Developers), and System Specifications (for System Integrators). Note that these all fit into the schema, and only the file contents will vary by role.

**Benefits:**

- **Developer Efficiency:** There is a significant labor reduction in auto generating and updating the required schema representation. Developers do not need to manually create and maintain XML schema files.

- **System Consistency:** the schema is automatically generated from the models, and is consistent with all other artifacts generated from the model.

- **Document Consistency and Clarity:** schema files will be generated in a consistent manner each time. Schema files will not be dependent on a developer’s favorite style.

**How To Use:** See section “Export Model”
Manual Alternative: The manual user is responsible for editing the XML (either with a standard text editor or with an XML editor), and maintaining consistency of this file with all Code, Data Types, System Configurations, Build Configurations, and documentation.

3.3 System Schema Importer Tool
The System Schema importer tool imports schema components into the working model. Application Developers will import common Data Models to use in defining component interfaces. System Integrators will import Component Models and Platform Models, for use in defining the overall system.

Benefits:
- **Developer Efficiency**: Developers are not required to manually enter models.
- **Consistency**: Imported models will directly match the System Schema. The schema captures the necessary specifications for integrating an component or subsystem (Data types, communication style, ...)

3.4 FACE Data Model Exporter Tool
The FACE Data Model Exporter tool exports FACE Data Models expressed in FACE Data Model XMI from models in the GME FACE paradigm. The tool allows data model curators/program data management engineers to manage and distribute Data Models for use with other modeling environments.

Benefits:
- **Developer Efficiency**: There is a significant labor reduction in automatically exporting models to FACE Data Model XMI, because developers do not need to manually create and maintain them.
- **Consistency**: FACE Data Model XMI is automatically generated from the GME models, and is consistent with all other artifacts generated from the model.

How to Use: See section “Export FACE Data Model”

Manual Alternative: Users can manually create a FACE Data Model XMI file and manually maintain its consistency with the existing GME model.

3.5 FACE Data Model Importer Tool
The FACE Data Model Importer tool imports FACE Data Models expressed in FACE Data Model XMI into the GME FACE paradigm. The tool allows data model curators/program data management engineers to use the FACE Shared Data Model in GME and to develop component and system models with Data Models created in other modeling environments.

Benefits:
- **Developer Efficiency**: There is a significant labor reduction in automatically exporting models to FACE Data Model XMI, because developers do not need to manually create and maintain them.
- **Consistency**: FACE Data Model XMI is automatically generated from the GME models, and is consistent with all other artifacts generated from the model.
How to Use: See section “Import FACE Data Model”

Manual Alternative: Users can manually create a Data Model in the GME FACE paradigm and manually maintain its consistency with the existing FACE Data Model XMI file.

3.6 Datatype Generator Tool

The Datatype Generator tool processes Data Models created with or imported into the model editor to produce executable code. The output formats available are:

- C headers
- C++ headers
- Java classes
- Ada packages

Benefits:

- **Developer Efficiency**: via Automation. The generator automatically creates a significant amount of code, reducing programmer effort and development cost
- **Consistency**: Generated data structures are implemented in a consistent way, resulting in more regular, maintainable code
- **Conformance**: Generated data structures are FACE-conformant.

How to use: See section “Datatypes” in the “Generating Code form Example Model” section.

Manual Alternative: Users can manually develop C++ headers, Java classes, or Ada packages based on documentation and reading the System Schema descriptions.

3.7 Component Shell Generator Tool

The Component Shell Generator creates a FACE conformant component (portable component or platform-specific service) shell from a component model created with the model editor. The component shell is boiler plate code (TSS connections, configuration, etc.) needed by a FACE-conformant component. The generated code contains hooks for the developer to fill in their custom business logic.

Benefits:

- **Developer Efficiency**: Significant fraction of the code is automatically generated, saving programmer effort and development costs. This frees the programmer to focus their effort on developing the business logic code.
- **Conformance**: Code generated is conformant to the FACE standard.
- **Code Quality**: Code quality is high, and all generated code is consistent to the code generation templates.
- **Upgradability**: User-created code is isolated from low-level system calls, which tend to change between different versions of standards. Update from FACE V1.0 to V2.0, for example, was largely in the generated code.
- **Portability**: Code can be generated for ARINC-653 or POSIX configurations of FACE
• **Model/Code Integrity**: Code (and data structures, TSS, configurations, etc) are all generated from a common source (the Model or System Schema), so all details are consistent. Manual approaches have the risk of partial updates or consistency errors, where components can be configured differently or have inconsistent interfaces, or have issues resulting from improper interpretation of documentation.

• **Documentation**: Documentation for the component and system is partially generated from the models, so it will match the generated code. Labor is also saved vs. manual document preparation.

**How to use**: See the “Generating Code from Example Model” / “Portable Component/Platform-Specific Service Code” section in the How-To guide below.

**Manual Alternative**: Users can manually create all component code from scratch or hack upon an existing example. Programmers are responsible for writing conformant code, using proper include files for data type specification, and making FACE API calls. The user must also maintain the code consistency with component documentation.

### 3.8 Build System Generator

The Build System Generator, included in both the Component Shell Generator and the TSS Generator, will create build files from a component model or system model. The build files for all languages are Makefiles.

**Benefits**:  
- **Developer Efficiency**: Build files are automatically generated, saving programmer effort and development costs.  
- **Portability**: Multiple build systems are supported.  
- **Model/Code Integrity**: System build files match the component/system models and are complete. Manual creation of the build system can miss steps in the build process, resulting in incomplete compiles or old object files being used in a build.

**Manual Alternative**: The user can manually create Makefiles, Ada project files, Maven project object models, or use a build system of choice. The user must ensure that all libraries, TSS components, include files, etc. are properly referenced and dependencies maintained. The user is also responsible for making a project file for their chosen IDE (e.g. Eclipse Project).

### 3.9 TSS Generator

The TSS Generator automatically creates the necessary code to implement a TSS, including the data-dependent interface, data marshaling, and FACE API dependent code to pass data between components. C++, Ada, and Java TSS code can be generated to target POSIX operating systems; C++ TSS code can also be generated to target the ARINC653 emulator. This code can be directly built against a component generated by the Component Shell Generator. **Note**: Current data marshaling uses a high-performance, direct binary copy for marshaling (pending guidance from the FACE standard on over-the-wire format).
Benefits:

- **Developer Efficiency**: The TSS code is automatically generated, saving programmer effort and development costs. TSS code is highly repetitive, and thus prone to error.
- **Conformance**: Code generated is conformant to the FACE standard. Generated Code can conform to a selected FACE Profile (Safety/General).
- **Code Quality**: Code quality is high, and all generated code is consistent to the code generation templates.
- **Upgradability**: User-created code is isolated from low-level system calls, which tend to change between different versions of standards. Update from FACE V1.0 to V2.0, for example, was fully in the generated code.
- **Portability**: TSS Code can be generated for ARINC-653 or POSIX configurations of FACE, allowing application portability or migration to different uses.
- **Model/Code Integrity**: Code (and data structures, configurations, etc.) are all generated from a common source (the Model or System Schema), so all details are consistent. Manual approaches have the risk of partial updates or consistency errors, where components can be configured differently or have inconsistent interfaces, or have issues resulting from improper interpretation of documentation.
- **Documentation**: Documentation for the component and system is partially generated from the models, so it will match the generated code. Labor is also saved vs. manual document preparation.

How To: See the “Generating Code from Example Model” / “Portable Component/Platform-Specific Service Code” section in the How-To guide below.

Manual Alternative: Developers can manually write the TSS data management code from scratch or modify an existing example. However, note that developers are responsible for writing conformant code, using proper include files for data type specification, and properly calling FACE API functions. The developer must also repeat the process for each variation of TSS underlying infrastructure, and for each unique data type and direction of communication. This can result in a large amount of manually written code. The user must also maintain the code consistency with component documentation.

### 3.10 OS Configuration Generator

MTF generate system/partition configuration files for the target RTOS. These files specify how partitions are created and configured, and which components will run in each partition. The configuration generator also creates all component ports and channels, and creates links between the channels and the ports. Multiple formats for the files can be generated, including Standard ARINC-653 and target RTOS variations (LynxOS format)

- **Developer Efficiency**: The system configuration is automatically generated, saving programmer effort and development costs.
- **Conformance**: Configuration generated is conformant to the FACE standard.
• **Upgradability:** User-created code is isolated from low-level system calls, which tend to change between different versions of standards. Update from FACE V1.0 to V2.0, for example, will be fully in the generated code.

• **Portability:** Configurations can be generated for ARINC-653 or POSIX configurations of FACE, allowing application portability or migration to different uses. Configurations can be generated for multiple target RTOS’s, pending implementation of standard configuration files by infrastructure developers.

• **Model/Code Integrity:** Configurations are consistent with Code (and data structures) and TSS, as are all generated from a common source (the Model or System Schema), so all details are consistent. Manual approaches have the risk of partial updates or consistency errors, where components can be configured differently or have inconsistent interfaces, or have issues resulting from improper interpretation of documentation.

• **Documentation:** Documentation for the component and system is partially generated from the models, so it will match the generated code. Labor is also saved vs. manual document preparation.

**How To:** See the “Generating Portable Component/Platform-Specific Service Code” section in the How-To guide below.

**Manual Alternative:** Users can manually write TSS data management code from scratch or hack upon an existing example. Programmers are responsible for writing conformant code, using proper include files for data type specification, and properly calling FACE API functions. The developer must also repeat the process for each variation of TSS underlying infrastructure, and for each unique data type and direction of communication. This can result in a large amount of manually generated code. The user must also maintain the code consistency with component documentation.

### 3.11 ARINC-653 Partition Generator

The ARINC-653 partition generator implements the partitions as specified in a System Model, producing the code necessary to create a partition image. The partition code consists of a “main” function, which contains code to initialize the system and to execute all components that are assigned to the partition.

• **Developer Efficiency:** The system main function is automatically generated, saving programmer effort and development costs.

• **Conformance:** Configuration generated is conformant to the FACE standard.

• **Upgradability:** User-created code is isolated from low-level system calls, which tend to change between different versions of standards. Update from FACE V1.0 to V2.0, for example, will be fully in the generated code.

• **Model/Code Integrity:** Configurations are consistent with Code (and data structures) and TSS, as are all generated from a common source (the Model or System Schema), so all details are consistent. Manual approaches have the risk of partial updates or consistency errors, where components can be configured differently or have inconsistent interfaces, or have issues resulting from improper interpretation of documentation.
- **Documentation**: Documentation for the component and system is partially generated from the models, so it will match the generated code. Labor is also saved vs. manual document preparation.

**Manual Alternative**: Users can manually write integration code from scratch or hack upon an existing example. Programmers are responsible for writing conformant code, using proper function calls, and application life-cycle management. The user must also maintain the code consistency with system documentation.

### 3.12 MTF IDL Compiler

The MTF IDL Compiler is a standalone tool that compiles IDL files defined in the FACE standard into language-specific code. The available output languages are C, C++, Java, and Ada.

**Benefits:**
- **Developer Efficiency**: via Automation. The generator automatically compiles to the desired language, reducing programmer effort to manually interpret IDL from the standard
- **Consistency**: FACE IDL is always compiled the same way using the MTF IDL compiler, so maintainability and cross-vendor consistency is ensured
- **Conformance**: Generated code is FACE conformant.

**How to use**: See section “Compiling FACE IDL”.

**Manual Alternative**: Users can manually write language-specific code that conforms to the FACE IDL in the FACE standard.

### 3.13 ARINC-653 Emulator

The ARINC 653 Emulator is a core port of the SDK. This Emulator provides a full ARINC 653-compatible runtime for system & application development and testing.

The ARINC-653 Emulator implements the full APEX Services and includes an APEX Module Manager. ARINC-653 compatible partitions execute on the emulator. Multiple ARINC-653 processes are supported, as are ARINC-653 communication ports and channels. When coupled with the TSS Generator, OS Configuration Generator, Partition Generator, and Component Shell Generator, a full FACE-Conformant system can be developed, implemented, and tested.

Real-Time partition time-separation specifications are faithfully executed by the emulator, as are memory space partitioning.

The ARINC-653 Emulator is implemented on top of a standard Linux Kernel and POSIX Thread Library. This allows the system to be executed on commodity PC/Laptop hardware with commonly available, open source Linux distributions.

**Note** that the emulator is not intended for deployment on flight systems; rather it should be used to rapid development, including debugging and testing. Implementation on top of Linux prohibits the
emulator from operating with guaranteed, hard-real-time specifications. Timing jitter is suitable for development, typically less than 10 ms. Please see the ACM Tools Guide for details, implementation, and API of the ARINC-653 emulator.

The ARINC-653 Emulator has several benefits:

- **Developer Efficiency**: Developers are able to develop and test on standard desktops, and standard Linux. This allows the use of all tools (editors, IDE’s, debuggers, profilers, etc) available on Linux.
- **Software Cost**: The emulator is free. Typical developer seats on a commercial RTOS cost $20K or more.
- **Hardware Cost**: The system runs on any PC or laptop compatible with Linux. These machines are much more available and less expensive than embedded hardware.
- **Conformance**: The ARINC-653 emulator is fully conformant with the APEX API.
- **Portability**: The Emulator executes on standard Linux distributions, so can be carried along for demonstrations. The emulator can be executed inside a Virtual Machine (VM), however at a performance penalty.

**Manual Alternative**: Users can develop and test with a commercial, FACE conformant (ARINC) RTOS. These are relatively expensive, and consequently typically few of these resources will exist within a company. The user must also deal with the limitations of running/debugging on embedded platforms, with reduced access to standard desktop tools.

### 3.14 Timing Analyzer

The Timing Analyzer consists of two parts. The first part is implemented in the ARINC Emulator as instrumentation. This software records the execution time of all components and communications.

The second part allows visualization of the timing data, producing a timeline of execution. This capability is extremely useful in assessing the real-time properties of the system, and locating the source of any timing errors.

### 3.15 Integration Harness Generation (Prototype)

The integration harness generator creates data drivers/loggers for TSS connections. This allows the replay of data streams across a TSS interface, and logging of data received from a TSS interface. Data can be in standard formats (e.g. MATLAB *.MAT formats, XML).

These test components are valuable in implementing unit tests, subsystem tests, and system tests. They provide accurate stimulation of components, reproducing test data, and logging results for component/subsystem/system functional verification.

This approach has several benefits:

- **Developer Efficiency**: Data drivers do not need to be manually developed. Data drivers are required to test and prove system accuracy in a methodical manner.
• **Conformance:** Generated code is conformant to the FACE standard. The required System Schema for the components and system are automatically generated, reducing the cost of implementing a full package, especially if the integration harnesses are to be delivered with the system as supporting artifacts.

**Manual Approach:** Users can manually write test harness code from scratch. Programmers are responsible for writing conformant code, using proper function calls, and ensuring all data is properly managed, including correct formatting of all input and output data. The user must also maintain the code consistency with system documentation.
4  Modeling Tool Architecture

MTF utilizes the Generic Modeling Environment (GME), created at Vanderbilt’s Institute for Software Integrated Systems (ISIS). It implements the Modeling Editor, with all of the FACE concepts enforced as language concepts and modeling constraints. MTF includes a variety of plug-in GME model interpreters, which implement each of the Software Development Kit and Integrators Tool Kit components as described above. The ARINC-653 Emulator provides a platform for real-time execution.

Several companion documents provide details on the individual tools:
- Modeling Language Documentation: Describes the concepts, model types, constraints, views of the MTF FACE modeling language
- ARINC-653 Emulator Documentation: Describes the emulator architecture, design, and application.

5  Modeling Tool Interoperability

Modeling Ecosystem Overview

The Generic Modeling Environment (GME) provides two model storage formats: MGA and XME. MGA is a custom binary format which allows for fast access and compact storage of models. XME is a custom XML format which is guaranteed to be compatible across versions of GME. Before migrating to a different version of GME it is recommended to export models in XME format. The MGA format is not guaranteed to be compatible across versions of GME.

MTF includes a plugin that enables import/export of models (in System Schema format) to/from the GME Editor. The System Schema is an open interchange format which enables the exchange of models between modeling tools from different vendors. This also enables creation of tools which can operate directly on models independently of any specific modeling environment.
6 Installing MTF

6.1 System Requirements
- Supported Operating Systems
  - Microsoft Windows XP
  - Microsoft Windows Vista
  - Microsoft Windows 7
- Hardware Requirements
  - No additional requirements beyond those imposed by Windows.

6.2 Prerequisites
The following prerequisites are needed by portions of MTF. The installer will give warnings if any of these prerequisites are not installed on the system. It is recommended that the user install all prerequisites, however, the tools may be installed without some prerequisites with limited functionality.

- [Generic Modeling Environment (GME) 14.12.4 (32-bit)]
- [Python 2.7 for Windows]
- [Python for Windows Extensions (pywin32-py2.7)]
- [Microsoft .NET v4.0]
- [Java 1.7 (32-bit)]

6.3 Questions/Issues
Please direct inquires about MTF to [face@isis.vanderbilt.edu]
7 Getting Started: How-To Tutorials

Using the SDK tools to implement an application follows the following process for creating application components:

The ITK tools support modeling and creation of systems:

MTF includes an example model (cop_demo) which is used in several of the How-To Tutorials. By default this is in “C:\Program Files (x86)\MTF\examples” which is read-only to non-administrators. Follow instructions in Section 7.3.1 to import the example model “models\cop_demo.xme”.
7.1 Creating a New Model
- Open GME (“Start->All Programs->GME->GME“)
- Select “File->New Project”
- In the select paradigm dialog choose “FACE”
- Click the “Create New...” button
- Click “Next”
- Choose a file name (i.e. MyModel.mga)

7.2 Opening an Existing Model
- Open GME (“Start->All Programs->GME->GME“)
- Select “File->Open Project...”
- Choose a GME project (*.mga file) to open

7.3 Importing/Exporting a Model

7.3.1 Import from XME (GME persistence format)
For an explanation of model formats see section 3
- Open GME (“Start->All Programs->GME->GME“)
- Select “File->Import XML...”
- Choose a “*.xme” file to open
- Click “Next >”
- Choose a file name (e.g. MyModel.mga)

7.3.2 Export to XME (GME persistence format)
For an explanation of model formats see section 3
- Open the “*.mga” file to be exported
- Select “File->Export XML...”
- Choose a file name (e.g. MyModel.xme)
- Click “Save”

7.3.3 Import from System Schema
For an explanation of model formats see Section 5
- Create a new or open an existing “*.mga” file
- Click the model import button on the tool bar. Hover over the button and the tooltip text will say: “Import Model from System Schema”

- Choose a “*.xml” file (conformant to the MTF System Schema) to open. This must be
- The GME Console will indicate if the import was successful
7.3.4 Export to System Schema
For an explanation of model formats see Section 5
- Open an existing “*.mga” file
- Click the model export button on the tool bar. Hover over the button and the tooltip text will say: “Export Model as System Schema”
- Choose a file name (i.e. MyModel.xml)
- The GME Console will indicate if the export was successful

7.3.5 Import Data Model from FACE XMI
For an explanation of model formats see Section 5
- Open an existing *.mga file
- Click the model import button on the tool bar. Hover over the button and the tooltip text will say: “Datamodel Importer”
- Choose a file name (i.e. MyDatamodel.face)
- The GME Console will indicate if the export was successful

7.3.6 Export Data Model to FACE XMI
The Data Model Exporter is currently a command-line only tool.
- Open a DataModel model in your GME project
- Click the model export button on the tool bar. Hover over the button and the tooltip text will say: “Datamodel Exporter”
- Choose a file name (i.e. MyDatamodel.face)
- The GME Console will indicate if the export was successful
7.4 Windows in GME
GME features multiple windows that can be used to view and edit models. These are dockable windows that can be arranged or hidden to suit the user’s preferences. If a particular window is not visible it may need to be made visible by selecting it under the “View” menu.

7.4.1 Part Browser
The Part Browser window shows the parts that can be inserted into the current model in the current aspect. It shows all parts except for connections. At the bottom of the Part Browser, tabs show the available aspects of the current model. Clicking on a tab will change the aspect of the current model to the selected one. It also attempts to change the aspect of all the open models. If a particular model does not have the given aspect, its current aspect remains active.

The Part Browser can be used to drag a single object at a time and drop it either in any editor window or in the Model Browser. If a reference is dragged, a null reference is created because the target object is unspecified. Remember that references (null references included) can be redirected at any time by dropping a new target on top of them (see more detailed discussion where the drag and drop operations are described).

Note that the Part Browser window, just like the Model Browser window, is dockable; it can float as an independent window or it can be docked to any side of the GME Main Window.

7.4.2 Main Window
When a model is selected for editing, a graphical editor window opens up in the Main Window to allow editing of that model. The editor window shows the contents of the selected model in one aspect at a time. An Aspect in GME is a predefined filter which will only show a specific subset of a model. Selecting different Aspect will provide a different combination of visible model parts and choices in the Part Browser.

Many of the included tools (i.e. code generators) require that a model is open in the Main Window, this lets the code generator know which model should be used to generate code.

7.4.3 GME Browser
Once a project has been loaded, the GME opens a Model Browser window. The GME Browser is primarily used to organize the individual models that make up an overall project, while the graphical editor is used for actually constructing the project's individual models.
The most important high-level features of the GME Browser are accessible through the three tabs displayed at the top of the GME Browser. These tabs deal with the Aggregate, Inheritance, and Meta hierarchies.

The Aggregate tab contains a tree-based containment hierarchy of all folders, models, and parts from the highest level of the project, the Root Folder. The aggregate hierarchy is ignorant to aspects, and is capable of displaying objects of any kind. More information on the aggregate hierarchy will be provided shortly.

The Inheritance tab supports advanced features related to type inheritance that are not necessary for using MTF. For more details about the Inheritance tab consult the GME Manual.

The Meta tab shows the modeling language at a glance: it displays the legally available array of Folders and objects that can be added at any level within the aggregate hierarchy. For example, at the "Root Folder" level we can add "Folder" folders. Within these folders, we can add models Primitive and Compound. From these models, more parts can be added.

7.4.4 Object Inspector
Attributes and Preferences are available in a modeless dialog box, called the Object Inspector. There is no OK button; changes are updated immediately. More precisely, changes to toggle buttons, combo boxes (i.e. menus) and color pickers are immediate. Changes to single line edit boxes are updated when either “Enter” is hit on the keyboard or the edit box loses the input focus, i.e. you click outside the box. The only difference for multiline edit boxes is that they use the Enter key for new line insertion, so hitting it does not update the value.
Selecting an object or inserting, dropping or pasting it selects that object for the Attribute browser. If more than one object is selected – in the Model Browser or in the Model Editor - the attribute browser will allow only the common attributes of these objects.

At the top of the dialog there are three tabs, one for the attributes one for the preferences and another for the properties. Note that the Attribute Browser window, just like the Model Browser window, is dockable; it can float as an independent window or it can be docked to any side of the GME Main Window.

### 7.4.5 Console

The console window is by default docked to the bottom of GME and is where info, errors and warnings are logged. The console can be cleared by clicking Clear button in the bottom right of the console window.

### 7.5 Creating Model Elements

#### 7.5.1 Create a Data Model Element

The example Data Model in cop_demo has an IDLStruct “Location_alt_lat_long” with three compositions altitude, latitude, and longitude. The “altitude” composition is a reference to the IDL_Double “WGS84PositionHeightMeasurementAxis”. The “latitude” and “longitude” compositions are references to the IDL_Doubles “WGS84PositionLatitudeMeasurementAxis” and “WGS84PositionLatitudeMeasurementAxis”.

You can view these compositions by double-clicking the “Location_alt_lat_long” element under the PlatformDataModels->COP_Demo_Platform_Model->IDLPrimitives in the GME Browser:
Let’s create a new IDLStruct “Location_alt_lat_long_floats” that uses IDL_Floats instead of IDL_Doubles.
To do this, first we need to create three new IDL_Floats:
- “WGS84PositionHeightMeasurementAxis_float”
- “WGS84PositionLatitudeMeasurementAxis_float”
- “WGS84PositionLongitudeMeasurementAxis_float”

7.5.1.1 Create IDLPrimitive

To create a new IDL_Float:
- Double click on the IDLPrimitives object in the GME Browser
- Right click in the Main Window and select “Insert New Atom->IDL_Float”
- Rename the atom in the top line of the Object Inspector

7.5.1.2 Connecting Primitive to Logical Data Model

All IDLPrimitives in a Platform Data Model must realize a ValueElement in a Logical Data Model. To create this realization:
- Switch to the “Realization” aspect (located on the top of the Main Window)
- Method 1:
  o Switch to “Connect Mode” (located on the left-hand side of the program or by pressing (Ctrl+2)
  o Add a realization connection from the new Platform IDLPrimitives to the existing references to Logical Simple Measurements (i.e.: WGS84PositionHeightMeasurementAxis_ref) by left-clicking on the IDLPrimitive and then left-clicking on the Logical Simple Measurement
Switch back to “edit Mode” (located on the left-hand side of the program of by pressing (Ctrl+1)

- Method 2:
  - Right-click on the new Platform IDLPrimitives and select “Connect” then left-click on the Logical Simple Measurement (i.e.: WGS84PositionHeightMeasurementAxis_ref)

### 7.5.1.3 Create IDLStruct to contain IDLPrimitives

Next, create a new IDLStruct called “Location_alt_lat_long_floats” by:
- Right clicking in the Main Window and select “Insert New Model ->IDLStruct”
- Rename the atom in the top line of the Object Inspector to “Location_alt_lat_long_floats”
- Open the new model by doubleclicking on the model in the Main Window or in the GME Browser
- Insert three IDLCompositions by dragging the Composition part from the Part Browser into the Main Window
- IDLCompositions are references to IDLPrimitives, so we must next drag our new IDLPrimitives from the GME Browser onto the appropriate compositions in the Main Window
- Rename the atoms by clicking on them in the Main Window or GME Browser and change top line of the Object Inspector
7.5.1.4 Connect IDLStruct to Logical Measurement

Finally, because all Platform IDL Structs must realize a Logical Composite Measurement, we must add a realization connection to our newly created object:

- Double click on the IDLPrimitives object in the GME Browser
- Connect “Location_alt_lat_long_floats” to “WGS84PositionMeasurement” with one of the methods described in 7.5.1.2

Creation of the new IDL Struct is now complete.

7.5.1.5 Create a View

- In the GME Browser navigate to: “cop_demo->COP_Demo_USM->PlatformDataModels->COP_Demo_Platform_Model->Views”
- Right-click “Views” in the GME Browser and select “Insert Model->P_View”
- Rename the view in the top line of the Object Inspector
- IDK what to do from here, but I would like to create a view and continue the guide all based on the original Struct we created. This would make sure all steps are documented.

7.5.2 Create a Component (Portable Component or Platform Specific Service) Model

Let’s create a simple portable component that receives messages of type BSOLocation. First, create the portable component:

- In the GME Browser navigate to: “cop_demo->COP_Demo_USM->ARINC653”
- Right-click “ARINC653” and select “Insert Model -> PortableComponent”
- Name the component “bso_report_logger”
Double-click “bso_report_logger” to open the model editor:

Next, let’s model our portable component to receive BSO Reports by adding an InboundMessage port that references the BSOReport View:

(cop_demo -> COP_Demo_USM -> PlatformDataModels -> COP_Demo_Platform_Model -> Views -> BSOReport)

- Add an InboundMessage referencing the BSOReport model.

**There are three ways to do this:**

1. Create a null Subscriber reference by dragging a Subscriber from the part browser into the model editor.

Then drag the BSOReport View onto the null InboundMessage reference:
2. Copy the BSOReport View, paste a reference to the BSOReport by right-clicking in the bso_report_logger model and selecting “Paste Special->As Reference”, and select “InboundMessage” as the Role Type for the reference:

3. Drag a BSOReport reference into the component with Click + [Ctrl][Shift] + Drag.

(GME uses modifiers to create references and instances. These are called “drag-and-drop” modifiers.)
   - Reference: Click + [Ctrl] [Shift] + Drag
   - Instance: Click + [Alt] + Drag

NOTE: To redirect the type referenced by a Reference object to a new type, simply drag a Reference to the new type (see method 3 above) and drop on top of the Reference.

- Rename the InboundMessage to “bso_report_subscriber” by clicking on them in the Main Window or GME Browser and change top line of the Object Inspector
- Set the InboundMessage’s attributes as desired

Finally, save the model. Again, the options are:
- GME Project File (*.mga) – “File -> Save Project As” (binary format; tied to GME version)
- GME XME File (*.mga) – “File -> Export XML” (XML-based; works across GME versions)
- System Schema (*.xml) – Run “System Schema Exporter”
7.5.3 Create a System Model

Let’s create a simple system model that connects two components with POSIX Message Queues. We’ll start with the COP Demo model described in the Section 7 intro. For reference, there already exists a system model named “GuidanceDemoMQ”. To demonstrate the creation steps we’ll create a model named “GuidanceDemoMQ_v2”:

- In the GME Browser right click on the root folder “cop_demo” and select “Insert Model -> POSIXOperatingSystem”
- Name the new system model “GuidanceDemoMQ_v2”
- Double-click “GuidanceDemoMQ_v2” to open the model editor:

![GME Browser screenshot]

Now let’s deploy two components to this system and add a message queue to connect them:

- In the GME Browser navigate to: “cop Demo->COP_Demo_USM->UoP_Model->POSIX”
- Create a reference to “GPS_Sensor_PSS” by holding [Ctrl] [Shift] while dragging on the canvas
- Create a reference to “GuidanceSystem” by holding [Ctrl] [Shift] while dragging on the canvas
- Add a new MessageQueue to the system model by dragging from the Part Browser onto the canvas and name it “mqA” from the Object Inspector Window
- Select “mqA” and set the following properties in the Object Inspector:
  - MQ Name: mqA
  - MaxNumberOfMessages: 10
Now let’s connect the two components through the message queue:

- Switch to “Connect Mode” (located on the left-hand side of the program or by pressing (Ctrl+2)
- Click the “pla” port on “GPS_Sensor_PSS” as the source of a connection and then click on “mqA” as the destination
- Next, click “mqA” the source of a connection and then click on the “gui” port of “GuidanceSystem” as the destination
7.5.4 Multiplex TSS Messages in a System
The tools support multiplexing TSS messages over a single OS transport mechanism. For an example, see the example model ‘examples\2.0\apex\mux_demo\model\mux_demo.xme’, which uses standalone TSS components to multiplex messages over POSIX message queues and ARINC653 queueing ports.

7.6 Checking Constraints
The code generators, document generator, and export tools all require a validated model (one that violates no constraints) for correct operation. The MTF Constraints Interpreter checks that your model contains no metamodel violations, and that your Datamodel(s) have no OCL violations. To check constraints on a model, run the MTF Constraints Interpreter:

- Hover over the component generator button on the tool bar and the tooltip text will say: “MTF Constraints Interpreter”
- Click the interpreter button, and the GME Console will display which, if any constraints are violated by the model.
- Navigable links are provided for model components causing constraint warning/violations.

Note: You should ensure your model has no constraints violations before running any code generators, importers, or exporters. These tools expect a valid model, and are not guaranteed to run correctly or produce correct output if given an invalid model as input.
7.7 Compiling FACE IDL
The MTF IDL Compiler is a standalone, command-line only tool. (See the IDL Compiler’s documentation for more information.)
8 Code Generators

8.1 Generating Code from Models

First, create a directory structure to store the generated files. The recommended structure is:

- `<model_name>`
  - `datatypes`
  - `components`
  - `tss`

As an example, open “<MTF>\examples\2.1\models\cop_demo.xme” in GME.

8.1.1 Generating Portable Component/Platform-Specific Service Code

The Component Generator interprets a Unit of Portability (Portable Component or Platform-Specific Component) model and generates a component shell containing boiler plate code (TSS connections, configuration, etc.) and hooks for custom behavioral logic. To demonstrate the tool, let’s generate a component shell from the cop_demo model:

- Ensure your model has no constraints violations (see Section 7.9).
- Open a Portable Component or Platform-Specific Service model to generate code from.

(The Component Generator operates on whatever component model is currently open.)

- In the GME Browser navigate to:
  - “cop_demo -> COP_Demo_USM -> UoP_Model -> ARINC653 -> cop_pc”

![GME Browser Screenshot](image)
Double-click “cop_pc” to open the model view.

(To help simplify the viewability of complex models, models contain aspects in which only certain parts of the model are visible. These aspects often correspond to FACE Segments. For example, PortableComponent models have a “TransportServicesSegment” aspect, which shows only the input and output TSS ports of the component.)

- Hover over the component generator button on the tool bar and the tooltip text will say: “Component Generator”

- Click on the generator button (there may be a short pause while the model is being validated) to be prompted to:
  - Select an output directory (e.g. \cop_demo\components).
  - Select a language to generate (C++, Ada, or Java)
    (If Java is chosen, you must specify a package for the generated classes/interfaces)
  - Select a template set

- The GME Console will display which files were generated.

This tool can also be invoked from the command line as an executable JAR: (java -jar <MTF>/bin/ComponentGenerator-x.x.x.jar). The “--help” option will display usage.
8.1.2 Generating Integrator Code, TSS Library and Configuration Files

- Ensure your model has no constraints violations (see Section 7.6).
- Open a `POSIXOperatingSystem/ARINC653OperatingSystem` model to generate code from:
  - In the GME Browser navigate to: “cop_demo -> COPDemoSystem”
  - Double-click “COPDemoSystem” to open the model view.

- Hover over the system generator button on the tool bar button and the tooltip text will say: “TSS Generator”

- Click on the generator button (there may be a short pause while the model is being validated) to be prompted to:
  - Select an output directory (e.g. \cop_demo \tss).
Select a template set

(The currently, the system generator only supports systems where all components are configured to the same language, although mixed-language systems may be supported in the future.)

- The GME Console will display which files were generated and show if there were any errors or warnings during generation.

8.1.3 Generating Datatypes

- Ensure your model has no constraints violations (see Section 7.6).
- Open a DataModel model to generate code from:
  - In the GME Browser navigate to: “cop_demo -> COP_Demo_USM”
  - Hover over the system generator button on the tool bar button and the tooltip text will say: “Datatype Generator”
  - Click on the generator button (there may be a short pause while the model is being validated) to be prompted to:
    - Select an output directory (e.g. \cop_demo\datatypes).
Select a language (C, C++, Ada, or Java)
(If Java is chosen, you must specify a package for the generated classes/interfaces)

The GME Console will display which files were generated and show if there were any errors or warnings during generation.

This tool can also be invoked from the command line as an executable JAR:
\( \text{java -jar MTF/bin/DataTypeGenerator-2.1.0.jar} \). The “--help” option will display usage.

8.2 Hand Writing Behavior Logic for Components
The Component Generator creates a service shell with all the boilerplate code needed to manage the creation and destruction of a portable component or platform-specific service. This enables the developer to focus on writing the behavioral logic instead of dealing with the details of creating and destroying connections, threads, etc.

The generated code is organized such that auto-generated code is segregated from hand-written code. For the hand-written code, files with empty functions for users to fill in are generated. These functions include application lifecycle management hooks. The files containing these empty functions are created once and not overwritten if they already exist in the output directory. Build files are designed to allow configuration using environment variables.

<table>
<thead>
<tr>
<th>Completely Auto-Generated Files</th>
<th>Files requiring hand-written modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>- <code>&lt;datatype&gt;_&lt;reader/writer&gt;.hpp/cpp</code></td>
<td>- <code>&lt;app&gt;_behav.hpp/cpp</code></td>
</tr>
<tr>
<td>- <code>&lt;app&gt;.hpp/cpp</code></td>
<td></td>
</tr>
<tr>
<td>- <code>&lt;app&gt;_main.cpp</code></td>
<td></td>
</tr>
<tr>
<td>- TS.cpp</td>
<td></td>
</tr>
<tr>
<td>- Makefile</td>
<td></td>
</tr>
<tr>
<td>Ada</td>
<td></td>
</tr>
<tr>
<td>- <code>&lt;datatype&gt;_&lt;reader/writer&gt;.ads/adb</code></td>
<td>- <code>&lt;app&gt;_behav.ads/adb</code></td>
</tr>
<tr>
<td>- <code>&lt;app&gt;.ads/adb</code></td>
<td></td>
</tr>
<tr>
<td>- <code>&lt;app&gt;_main.adb</code></td>
<td></td>
</tr>
<tr>
<td>- Makefile</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>- <code>&lt;datatype&gt;_Reader/Writer&gt;.java</code></td>
<td>- <code>&lt;app&gt;.java</code></td>
</tr>
<tr>
<td>- <code>&lt;app&gt;_Base.java</code></td>
<td></td>
</tr>
<tr>
<td>- Main.java</td>
<td></td>
</tr>
<tr>
<td>- Makefile</td>
<td></td>
</tr>
</tbody>
</table>
As an example, follow the steps in Sections 8.11 and 8.1.1 to create an example portable component “bso_report_logger” and generate its code. The generated boilerplate code contains hooks for behavioral code:

```c
void BEHAV_INITIALIZE(void)
{
    printf("FACE Application 'bso_report_logger' behavior initialize...\n");
    // hand written code goes here...
}

void BEHAV_STARTUP(void)
{
    printf("FACE Application 'bso_report_logger' behavior startup...\n");
    // hand written code goes here...
}

void BEHAV_FINALIZE(void)
{
    printf("FACE Application 'bso_report_logger' behavior finalize...\n");
    // hand written code goes here...
}
```

Let’s implement some simple behavior - receiving and logging a message – by creating a thread that invokes the generated typed TSS receive interface for BSOReports. In the behavioral code header, add declarations for the thread entry point and body:

```c
namespace BSO_REPORT_LOGGER
{
    // Behav Entry Points
    void BEHAV_INITIALIZE(void);
    void BEHAV_STARTUP(void);
    void BEHAV_FINALIZE(void);

    // User-created code:
    void *LOGGER_THREAD_entrypoint(void *ptr);
    void LOGGER_THREAD(void);
}
```

In the behavioral code body, add a thread that receives a BSOResult on every iteration, and add code to start the thread in BEHAV_INITIALIZE:
Notice that this behavior code does not use the FACE TSS methods directly. Instead, the code uses reader/writer objects which are also generated by the component generator. These reader/writer abstract away the TSS details by managing the connection id and other connection parameters.
8.3 Build Environment (C++)
The Makefiles created by the code generators target gcc 4.8 and expect the following variables:

<table>
<thead>
<tr>
<th>Generator:</th>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTF Component Generator</td>
<td>MTF_ROOT</td>
<td>Location of the infrastructure files distributed with the release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default location: C:\Program Files (x86)\MTF\infrastructure</td>
</tr>
<tr>
<td>MTF System Generator</td>
<td>MTF_PROJECT_ROOT</td>
<td>Location of project root (recommended directory containing ‘components’, ‘datatypes’ and ‘tss’).</td>
</tr>
</tbody>
</table>

(See <MTF>\examples for examples of how these should be set/used.)

8.4 Build Environment (Java)
The Makefiles created by the code generators expect the following environment variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTF_ROOT</td>
<td>Location of the MTF infrastructure files distributed with the release.</td>
</tr>
<tr>
<td></td>
<td>Default locations is: C:\Program Files (x86)\MTF\infrastructure</td>
</tr>
<tr>
<td>MTF_PROJECT_ROOT</td>
<td>Location of project root (recommended directory containing ‘components’, ‘datatypes’ and ‘tss’).</td>
</tr>
</tbody>
</table>

(See <MTF>\examples for examples of how these should be set/used.)

8.5 Build Environment (Ada)
The Makefiles created by the code generators target gnat 4.9 and expect the following variables:

<table>
<thead>
<tr>
<th>MTF Component Generator</th>
<th>MTF_ROOT</th>
<th>Location of the infrastructure files distributed with the release.</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td></td>
<td>Default location: C:\Program Files (x86)\MTF\infrastructure</td>
</tr>
<tr>
<td>MTF System Generator</td>
<td>MTF_PROJECT_ROOT</td>
<td>Location of project root (recommended directory containing ‘components’, ‘datatypes’ and ‘tss’).</td>
</tr>
</tbody>
</table>

(See C:\Program Files (x86)\MTF\examples for examples of how these should be set/used.)
8.6 Example Applications
MTF’s installation includes demonstration systems (default C:\Program Files (x86)\MTF\examples) which were created with the model code generators. All examples include a GME model and a FACE Data Model XMI file corresponding to the Data Model used in the GME model. The ARINC653 example contains C++ source code and build files; the POSIX example includes C++, Java, and Ada source code and build/project files. TSS and IOS headers/classes/libraries, and an example IOS implementation are included in the “<MTF>\infrastructure” directory. For instructions on how to build the examples see the README.txt files included in the examples.