ACM Design Tools and ACM Runtime are based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration.
# Table of Contents

1. **Introduction** ............................................................................................................. 0
   - 1.1 ARINC-653 Component Model ................................................................................. 0
   - 1.2 Model-Based Software Health Management ......................................................... 0

2. **ACM Linux Runtime** ................................................................................................ 3
   - 2.1 ARINC-653 Component Framework Overview ....................................................... 3
     - 2.1.1 APEX Service Library ...................................................................................... 3
     - 2.1.2 APEX Module Manager .................................................................................. 4
     - 2.1.3 Distributed Modules ...................................................................................... 4
     - 2.1.4 APEX Partition Scheduler .............................................................................. 5
     - 2.1.5 Middleware and Component layers ................................................................... 5
   - 2.2 System Requirements ............................................................................................. 5
     - 2.2.1 Optional ........................................................................................................... 6
   - 2.3 Installation ............................................................................................................... 6
   - 2.4 Installing Deliberative Reasoner ................................................................. **Error! Bookmark not defined.**
   - 2.5 Documents and Example ...................................................................................... 7

3. **Overview of the ACM Modeling Environment** ......................................................... 8
   - 3.1 Types ...................................................................................................................... 8
   - 3.2 Interface .................................................................................................................. 9
   - 3.3 Component ............................................................................................................. 9
   - 3.4 Correctness Contracts ........................................................................................... 12
   - 3.5 Component Health Managers .............................................................................. 13
   - 3.6 Subsystem ............................................................................................................. 15
   - 3.7 Assembly (System Design) ................................................................................... 16
   - 3.8 System Health Management ............................................................................... 17
   - 3.9 SLHM mitigation strategy .................................................................................... 18
   - 3.10 Deliberative Reasoning ......................................................................................... 20
     - 3.10.1 Functionalities and functional dependencies .................................................... 20
     - 3.10.2 Function Allocation ........................................................................................ 21
   - 3.11 Deployments ........................................................................................................ 27

4. **ACM Modeling Tools Setup (windows)** ..................................................................... 29
   - 4.1 System Requirements ............................................................................................ 29
4.2 Installation ........................................................................................................... 29
4.3 Questions/Issues ............................................................................................... 29
4.4 Model Storage Formats .................................................................................... 29

5. Working with the Modeling Environment ........................................................... 30
5.1 Create a New Model ......................................................................................... 30
5.2 Open a Model .................................................................................................... 30
5.3 Import Model (XME Format) ............................................................................ 30
5.4 Export Model (XME Format) ............................................................................ 30

6. Samples ................................................................................................................ 31
6.1 GPS-No-Fault .................................................................................................... 31
6.2 GPS-Sensor-No-Alarm ....................................................................................... 31
6.3 GPS-w-sensor-user-code-exception (with Sensor Alarm) .................................. 31
6.4 ADIRU – Emulation of Portion of Boeing 777 ADIRU Unit ............................... 32
6.5 IMU – Inertial Measurement Unit ..................................................................... 32
6.6 Demonstration Movies ...................................................................................... 32
   6.6.1 SimpleGPSDemo .......................................................................................... 34
   6.6.2 ExecutionOfSimpleGPSDemo .................................................................... 34
   6.6.3 ExecutionofIMUDemo ............................................................................... 34
   6.6.4 CHM_SHM .................................................................................................. 34

7. Steps to generate and execute models using ACM Tool ....................................... 35
7.1 Open Model: ..................................................................................................... 35
7.2 Interpret .............................................................................................................. 36
7.3 Run Time Setup: ............................................................................................... 37
7.4 Deliberative Reasoner ....................................................................................... 38
   7.4.1 Run Time Setup .......................................................................................... 38
   7.4.2 Offline Experimentation ............................................................................ 38
1. Introduction

1.1 ARINC-653 Component Model
The ARINC-653 component model (ACM) is built upon the concepts of ARINC-653, an avionics standard for safety critical operating systems. ARINC-653 systems group processes into spatially and temporally separated partitions, with one or more partitions assigned to each module (i.e. a processor), and one or more modules forming a system.

The ARINC-653 component model allows the developers to group a number of ARINC-653 processes into a reusable component. A component is a group of processes that share a common state. Components interact with each other via well-defined interaction patterns (chosen from a fixed set), facilitated by ports. In ACM, a component can have four kinds of interaction ports: publishers, consumers, facets (provided interfaces) and receptacles (required interfaces).

Each port has an interface type (a named collection of methods) or an event type (a structure). The component can interact with other components through synchronous call/return interfaces (assigned to facet or receptacles), and/or via asynchronous publish/subscribe events (assigned to publisher and consumer ports). Additionally, a component can host internal methods that are periodically triggered. Most of these interactions borrow concepts from other software component frameworks, notably from the CORBA Component Model (CCM).

The component model also provides guidance on allocation of task units to a component. Since this framework is geared for hard real-time systems, it is required that each port is statically allocated to an ARINC-653 process. Given that a facet interface can have more than one method, every method is allocated to a separate process. Furthermore, the access to component state is synchronized by a component-wide lock. Please see the [http://onlinelibrary.wiley.com/doi/10.1002/spe.1083/abstract](http://onlinelibrary.wiley.com/doi/10.1002/spe.1083/abstract) for detailed description.

1.2 Model-Based Software Health Management
Software Health Management (SHM) is a systematic extension of classical software fault tolerance techniques that aims at implementing the vision of self-adaptive software using techniques borrowed from System Health Management of complex engineering systems. System Health Management typically includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline), on the system level. It is performed at run-time, and it includes detection, isolation, and mitigation actions to remove fault effects.

Tools described in this document provide an approach for implementing software health management functions for component-based systems, specifically the ARINC-653 component model. The health management in the framework is performed at two levels. The Component-level Health Manager
(CLHM) provides localized and limited service for managing the health of individual software components. A higher-level System Health Manager (SLHM) manages the health of the overall system.

SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model automatically synthesized from the component assembly; the engine reasons about fault effect cascades in the system, and isolates the fault source components. This is possible because the data / behavioral dependencies and hence the fault propagation across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework. Once the fault source is isolated, the necessary system level mitigation action is taken. Our tools provide two options for specifying and implementing mitigation policies:

1. Reactive system mitigation. This type of mitigation is done by the code generated from the system mitigation specification written as timed state machines. These machines are specified in the ACM modeling environment.
2. Deliberative system mitigation. This type of mitigation is implemented by a reasoner installed with the runtime. This reasoner uses the list of faulty components, and the function allocation models (mapping of functions to components) to determine the alternative system configuration that can be used to recover the function. More detail of deliberative reasoner will be found in the modeling environment section.

Our approach is supported by a model-based design environment where developers can create models of the system and its components, as well as specify how fault mitigation will take place. A suite of software generators produce glue code that allows developer-supplied functional code or ‘business logic’ to form a collection of applications running on the ARINC-653 platform.

![ACM Tool Suite](Image)

Figure 1 ACM Tool Suite

Overall, our solution consists of two parts:
• A Linux-based runtime environment
  o This includes the ACM Component Framework, and
  o The Deliberative engine for implementing system-wide deliberative mitigation actions
• A modeling environment built using Generic Modeling Environment\(^1\) and associated design tools, as shown in Figure 1.

\(^1\) http://www.isis.vanderbilt.edu/Projects/gme
2. ACM Linux Runtime

2.1 ARINC-653 Component Framework Overview

Figure 2 describes the layers of our framework that implements the ARINC Component Model (ACM) runtime. The main purpose of this runtime is to provide support for developing and experimenting with component-based systems using ARINC-653 abstractions on top of Linux. The secondary goal is to design the top layers: component and processes such that they can be easily rebuilt over an actual ARINC-653 kernel.

The first two layers are a physical communication network and the physical computing platform. We use Linux as the operating system because it is widely available, supports a real-time scheduling policy SCHED_FIFO, and provides an implementation of the POSIX thread library. Memory partitioning between Linux processes provided by the Linux Kernel is used to implement the spatial partitioning between ARINC-653 partitions. Other layers from bottom to top are as follows.

2.1.1 APEX Service Library

APEX Services Library is the next layer. This library provides implementation of ARINC-653 interface specifications for intra-partition process communication that includes Blackboards and Buffers. Buffers provide a queue for passing messages and Blackboards enable processes to read, write, and clear single message. Intra-partition process synchronization is supported through Semaphores and Events. This library also provides process and time management services as described in the ARINC-653 specification.
Inter-partition communication is provided by sampling ports and queuing ports. Inter-Partitions communication can also be provided using sockets or remote procedure calls supported by the ORB. Overall, this layer was implemented in approximately 15,000 lines of C++ code.

This layer implements ARINC-653 processes as POSIX threads. ARINC653 processes, just like POSIX threads share the address space. Processes, both periodic and aperiodic, can only be created at initialization, following the ARINC-653 specification. Specified process properties include the expected worst case execution time, which cannot be changed at run-time

2.1.2 APEX Module Manager

Module Manager is the next layer. It is responsible for providing temporal partitioning among partitions (i.e., Linux processes). Each partition inside a module is configured with an associated period that identifies the rate of execution. The partition properties also include the time duration of execution. The module manager is configured with a fixed cyclic schedule with pre-determined hyperperiod.

In order to provide periodic time slices for all partitions, the time of the CPU allocated to a module is divided into periodically repeating time slices called hyperperiods. The hyperperiod value is calculated as the least common multiple of the periods of all partition in the module. In other words, every partition executes one or more times within a hyperperiod. The temporal interval associated with a hyperperiod is also known as the major frame. This major frame is then subdivided into several smaller time windows called minor frames. Each minor frame belongs exclusively to one of the partitions. The length of the minor frame is the same as the duration of the partition running in that frame. Note that the module manager allows execution of one and only one partition inside a given minor frame.

The module configuration specifies the hyperperiod value, the partition names, the partition executables, and their scheduling windows., which is specified with the offset from the start of the hyperperiod and a duration. The module manager is responsible for checking that the schedule is valid before the system can be initialized i.e. all scheduling windows within a hyperperiod can be executed without overlap. Additional documentation about Module manager can be found in the ModuleManager.pdf.

2.1.3 Distributed Modules

ACM runtime also supports distributed systems with multiple modules. Communication between partitions on different modules is implemented using Sockets. ACM modeling tools, if used, automatically generate the necessary code to setup and initialize the multi-module communication.

A multi-module system can run in one of the following two modes:

1. Unsynchronized: In this mode, each module is initialized with its own schedule and configuration file. All modules run independently.
2. Synchronized: In this mode, one of the modules is designated as system module. The system module sets up the system-wide hyper-period. Each module in the system has its own major frame and minor frames; however, their hyperperiod is set to be same as the system hyperperiod. In this mode, the system module synchronizes the start of each major frame across
all modules by sending a startup byte. All module except the system module always block at the end of one

2.1.4 APEX Partition Scheduler
APEX Partition Scheduler, the next layer, is instantiated using the APEX services emulation library for each partition. It implements a priority-based preemptive scheduling algorithm. This scheduler initializes and schedules the (ARINC-653) processes inside the partition based on their periodicity and priority. It ensures that all processes finish their execution within the specified deadline. Upon (deadline violation), the faulty process is prevented from further execution, which is the specified default action. It is possible to change this action to allow a restart.

2.1.5 Middleware and Component layers
Object Request Broker (ORB) is the next layer. This framework by default uses an open source CCM implementation, called MICO (http://www.fpx.de/MicoCCM/). However, it has been lately ported to also support CIAO (http://www.cs.wustl.edu/~schmidt/CIAO.html). The main ORB thread is executed as an aperiodic ARINC-653 process within the respective partition. For controllability, the ORB runs at a lower priority than the partition scheduler. Since ARINC does not allow dynamic creation of processes at runtime, the ORB is configured to use a predefined number of worker threads (i.e. ARINC-653 Processes) that are created during initialization.

Component and Process Layers include the glue code (generated from the ACM Modeling tools) and the user-provided implementation code. We advise you to use the associated windows-based modeling tools if you are using the component layer. The developer is responsible for specifying the necessary process properties such as periodicity, priority, stack size, and deadline in the models. The code-generator included in the associated windows installer provides glue code that maps each component interface method to an ARINC-653 process.

2.2 System Requirements
- Root Privileges
- kernel >2.6.9 will work. However, for best performance (and high resolution timers) we suggest that you use kernel >2.6.28.
- g++(GCC) > 3.4.6 will work. However, we suggest that you use any stable g++ release >4.0. http://gcc.gnu.org/
- ltmain.sh (GNU libtool) > 1.5.6 http://www.gnu.org/software/libtool/
- autoconf (GNU Autoconf) > 2.59 http://www.gnu.org/software/autoconf/autoconf.html
- automake (GNU automake) > 1.9.2 http://www.gnu.org/software/automake/
- doxygen http://doxygen.org/
- pkg-config http://pkg-config.freedesktop.org/wiki/
- udm (udm_3.2.9-1_i386.deb , udm-dev_3.2.9-1_i386.deb) http://repo.isis.vanderbilt.edu/
2.2.1 Optional
Optionally, you can use the eclipse CDT environment to edit and build the code produced from the modeling tools. Following are required for using the CDT environment:

2. Eclipse Auto tools plugin from: http://www.eclipse.org/linuxtools/projectPages/autotools/

2.3 Installation
Please note that prior to installing ACM, all the packages mentioned in Section 2.2 should be installed. It should be noted that currently UDM is available as a deb package only. Support for Red Hat based systems is coming soon. Follow these steps to build and install ACM. These have been tested on Ubuntu 10.04 32-bit installation.

1) Unzip the file. ($ unzip ACMRuntime-1.0.3.zip)

2) Run ./autogen.sh

3) To run the build with the appropriate flags run “./configure” with the appropriate flags.

**Build Configuration Flags**

- **Installation Folder** ( --prefix =INSTALL_FOLDER_PATH)
  --prefix=<path>/ACMINSTALL
  The user has to specify the path to the ACMINSTALL directory.

- **Logging level** ( --enable-logging-LEVEL )
  This flag allows the user to set the log level in ACM. The user should replace “LEVEL” in --enable-logging-LEVEL by the desired level. Typical options include --enable-logging-hmevent or --enable-logging-app. Type “ ./configure --help” for a list of all options.

- **Enabling CORBA support** ( --enable-corba )
  This flag allows the user to enable the component layer in ACM. Not specifying this option will generate the ACM library with only the pure ARINC emulation library.
  *Note:* The default CORBA support is with MICO. To enable CORBA support, user should download and install ACE, TAO &CIAO and set the flag as --enable-corba=ciao.

- **Enabling Deliberative Reasoner (DR)** ( --enable-DR )
  This flag allows the user to enable the Deliberative Reasoner (DR) support.

**Examples**

**ARINC emulation library only**

$ ./configure --prefix=<path>/ACMINSTALL --enable-logging-hmevent

**Component layer and ARINC emulation library**

$ ./configure --prefix=<path>/ACMINSTALL --enable-corba --enable-logging-hmevent

**Component layer, ARINC emulation library, Deliberative Reasoner (DR)**

$ ./configure --prefix=<path>/ACMINSTALL --enable-corba --enable-logging-hmevent --enable-DR

4) make ($ make)
5) make install ($ make install)
2.4 Documents and Example
This runtime provides the necessary library and header files for building ARINC-653 Programs on Linux. Also included are libraries necessary for building ACM programs.

Please refer to section “Component Framework Description” in the ACMRuntime-REFMAN.pdf for details on various layers of this runtime system. This should be followed by reading ModuleManager.pdf
3. Overview of the ACM Modeling Environment

Users may refer to the GME-tutorials (installed with GME software) to better understand the basic operations / terms described while working with a GME model.

Details of the modeling entities and their attributes can be found in ACM-ModelingTool-Manual.pdf. It is also installed in the docs folder under ACM in program files.

An ACM model contains 4 types of folders

3.1 Types

It holds data type definitions. A new data type can be added by right-clicking on the Type folder and choosing the appropriate option under “Insert Atom” or “Insert Model”. This includes primitive, enumerated and aggregated data types. A new Type can be a Primitive Type, an Enum type or an Aggregated Type. An Enum model holds EnumValue entities that are values of the enumerated type. An Aggregated model holds Member entities that are members of the aggregated type. The Member entities should refer to a valid type in the model.

Figure 3 Data Types as defined in the GPS No Fault Sample
3.2 Interface
It holds Interface definition. An interface model can contain one or more Method entities that represent the methods inside an Interface. Each Method can hold one or more Argument entities which are arguments of the method. The Argument entities are references to Types defined in the model. The position of the argument in the method definition is based upon its graphical location ordered from top to bottom and left to right. The INOUTType attribute allows the modeler to specify whether the argument is an input/output or input & output to the method. Method attributes - deadline, pre-conditions and post-conditions – further qualify the method.

3.3 Component
It holds Component definition. A new Component can be added by right-clicking on the folder and choosing the appropriate option under “Insert Model”. A component model contains one or more of the following:

Publisher – Represents an Event Source interface of the Component. The event type is specified by the Type entity referred to by the Publisher. It can be further qualified by the Publisher attributes such as period, deadline, pre-conditions and post-conditions

Consumer – Represents an Event Sink interface of the Component. The event type is specified by the Type entity referred to by the Publisher. It can be further qualified by the Publisher attributes such as period, deadline, pre-conditions and post-conditions
**Provides** - Represents a Provided-Interface of the Component. The interface type is specified by the instance of an Interface (defined under an Interface folder) contained inside the provided model. The Method instances inside the Interface instance can be further qualified by filling in their attributes - deadline, pre-conditions and post-conditions.

**Requires** - Represents a Required/Uses-Interface of the Component. The interface type is specified by the instance of an Interface (defined under an Interface folder) contained inside the required model. The Method instances inside the Interface instance can be further qualified by filling in their attributes - deadline, pre-conditions and post-conditions.

**Method** – Represents a Method inside the Component. The method is defined directly inside the Component and is not a reference or instance of a Method defined elsewhere. The properties of a method can be further qualified by filing in its attributes of period, deadline, pre-conditions and post-conditions.
**State Variables:** They represent the externally observable state of the component. Unlike CCM attributes they cannot be modified from outside. Each state variable is associated with a valid type by creating a reference to the type.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
<th>Default</th>
<th>Remark</th>
<th>Applicable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validity period</td>
<td>Double</td>
<td>([0, \infty))</td>
<td>(\infty)</td>
<td>Marks the time limit for an event to be valid (not stale).</td>
<td>Consumers</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>Boolean</td>
<td>True, False</td>
<td>False</td>
<td>Marks if the port can update the state of the component or not</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Deadline</td>
<td>Double</td>
<td>([0, \infty))</td>
<td>(\infty)</td>
<td>Marks the time limit by which the process should finish execution</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Deadline type ENUM</td>
<td>ENUM</td>
<td>HARD, SOFT</td>
<td>HARD</td>
<td>Marks the nature of the deadline</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Period</td>
<td>double</td>
<td>([0, \infty)) for SOFT deadline</td>
<td>(\infty)</td>
<td>Marks the rate at which the scheduler launches the process. Notice that the maximum response for hard deadline tasks should be less than or equal to the period</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Call type ENUM</td>
<td>ENUM</td>
<td>TWO WAY, ONE WAY</td>
<td>TWO WAY</td>
<td>Marks whether an RMI call is blocking or non-blocking(same as in CCM/CORBA)</td>
<td>Methods on synchronous ports</td>
</tr>
<tr>
<td>Post-condition String</td>
<td>String</td>
<td>N/A</td>
<td>N/A</td>
<td>The contract to be satisfied at the end of the execution</td>
<td>All ports</td>
</tr>
<tr>
<td>Pre-condition String</td>
<td>String</td>
<td>N/A</td>
<td>N/A</td>
<td>The contract to be satisfied at the beginning of the execution</td>
<td>All ports</td>
</tr>
<tr>
<td>Invariant String</td>
<td>String</td>
<td></td>
<td></td>
<td>The contract to be satisfied during the operation of trigger</td>
<td>Component triggers</td>
</tr>
</tbody>
</table>

*Figure 6 Attributes of Component Ports*
**Parameters**: Are configuration attributes which once set during initialization remain constant during the life cycle of that component

### 3.4 Correctness Contracts

Each functional entity in ACM can be annotated with correctness criteria, which are specified as pre-conditions and post-conditions. These conditions can be specified over the current value, or the history of the value, or rate of change of values of certain data elements. These data elements can be part of (a) the event-data of asynchronous calls, or (b) the function parameters of synchronous calls, or (c) the state variables of the component. While pre-conditions are assumptions that must be true before execution of a call, post-conditions are guarantees, which will be true after the execution.

The entry for Pre-condition and Post-condition attribute need to follow the format

```
CONDITION_NAME=EXPRESSION
```

The EXPRESSION is defined over the Publisher/Consumer event, Component State Variables, Method Arguments. The EXPRESSION support two kinds of utility function

![Figure 7 Nav Display Component. Also shown is the post condition expression on the receptacle method getgpsdata](image-url)
DELTA(x) – which computes \( x - x_{\text{past}} \) and returns a double value

RATE(x) – which computes the rate of \( x \) and return a double value.

Formally, all monitoring criteria can be expressed as

```
<PreCondition>::=<Condition>
<PostCondition>::=<Condition>
<Deadline>::=<double value> /* from the start of the process associated with the port to the end of that method */
<Data Validity>::=<double value> /* Max age from time of publication of data to the time when data is consumed*/
<Lock Time Out>::=<double value> /* from start of obtaining lock*/
<Condition>::=<Primitive Clause><op><Primitive Clause>|<Condition><logical op><Condition>|!<Condition> |True| False
<Primitive Clause>::=<double value>|Delta(Var)|Rate(Var)|Var /* A Var can be either the component State Variable, or the data received by the publisher, or the argument of the method defined in the facet or the receptacle*/
<op>::= < j|> | <= | >= | == | !=
<logical op>::=&& | ||
```

Figure 7 shows the example of the nav display component from the GPS sample models.

### 3.5 Component Health Managers

Each Component can contain a ComponentHM model that captures the behavior of the Health Manager associated with the Component. The behavior is captured as a state machine model. Component HM is a specialization of a state machine. For further details on ACM State Machines, please refer to the state machine documentation in the same folder.

The behavior expressed through the state machine is triggered when an error conditions is witnessed during the operation of the component. These error-conditions are captured in the Component HM model as “ErrorConds”. ErrorConds are pre-defined errors on specific processes associated with the component. The process is captured as the entity referenced by the ErrorCond - Publish/Consumer/Method interface (defined in the Component) or Method instance (defined in the Provides/Requires model defined in the Component). The specific error is captured through attributes on the ErrorCond. ErrorType attribute indicates whether the error is an APPLICATION_ERROR (pre-condition or post-condition violation) or other ErrorTypes such as DeadlineMissed, ValidityFailed etc. An APPLICATION_ERROR is further qualified by the name of Pre-Condition or Post-Condition set in the ConditionName attribute.

The trigger condition (on the transitions of the state-machine model) is expressed in terms of the ErrorConds captured in the Component HM model. The action attribute (in the transition or the Entry/Exit/During action tribute of the state) is expressed in terms of one of the following keywords –
IGNORE/ ABORT/ USE_PAST_DATA/ START/ STOP. Alternately, these actions can be expressed by the name of a Response entity defined in the Component HM model.

Each Response entity captures a pre-defined action on a particular process associated with the Component. The process is captured as the entity (Publisher/Consumer/Method) referenced by the Response entity. The action is captured through an attribute that can be set to IGNORE/ ABORT/ USE_PAST_DATA/ START/ STOP.

Figure 8 The CLHM of the NavDisplay component from the GPS Models
3.6 Subsystem

A Subsystem, as the name suggests, is a building block for a system model. It captures a group of interacting components.

A subsystem model holds instances of one or more components. The interaction between the Component ports (Publisher, Consumer, Provides & Requires) are captured as connections - a Publisher can be connected to one or more Consumer(s), a Provides interface can be connected to one or more Requires Interface(s). It is important to note that source and destination port in each interaction do not belong to the same component.

Alternately, a Subsystem model can hold Pseudo ports. Each of these pseudo ports is connected to exactly one component-port, thereby exposing the component-port to interactions outside the subsystem model.

Figure 9 A GPS subsystem from the IMU Model
3.7 Assembly (System Design)

An Assembly model captures the system design. It holds instances of component(s) and subsystem(s) and the interaction across the ports of these components and subsystems.

It is created inside a folder of type “Assemblies”. The entire system design may be captured in a single assembly model or spread out across multiple assembly model(s). All the related assembly model(s) are expected to be defined under the same folder. References of Component and Subsystem instances can be used if the Component/Subsystem instance needs to be used in multiple Assembly models/sheets.

The system integrator can set the initial states of all components (component instances) in the system (assembly models). A component can be in one of the following three states: Active, Inactive and Semi-active. When a component is in Inactive state, none of the ports in the Component perform their task. The Active state of a component is the exact opposite of the Inactive, and all the component ports perform their task. In a Semi-Active state, only the Consumer and Receptacle ports of a component are operational, the Publisher and Provided ports are disabled. Semi-active state is typically assigned to component instances that serve as passive replicas. A component is set to an inactive status if it is part of a redundant architecture and its services are not required currently.

Figure 10 GPS Assembly from the GPS No Fault Model. Note all the connections across component ports.
3.8 System Health Management

System Level Health Manager (SLHM) is at the top level in our health management strategy. SLHM is enabled in an ACM model by instantiating the SHM block in an assembly. This is done by creating a SHM sheet inside the Assemblies folder that holds the Assembly model(s) pertaining to the system. Based on this information, the ACM tools automatically augment the system assembly with three additional components: Alarm Aggregator, Diagnosis Engine, and SystemHMResponse Engine as shown in Figure 11.

![Figure 11 SLHM Architecture. SLHM Components are automatically configured by the ACM design tools.](image)

- **Alarm Aggregator** is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). It hosts an aperiodic consumer that is triggered by the data (alarm, and local mitigation command) provided by the Component Level Health Managers. The Alarm Aggregator assimilates the information received from the CLHM-s in a moving window (two hyper periods long) and sorts them based on their time of occurrence. A periodic publisher in the Alarm Aggregator feeds this sorted data to the Diagnosis Engine.

- **Diagnosis Engine** hosts an instance of a Timed Failure Propagation Graph reasoning engine. This engine is initialized by a Timed Failure Propagation Graph (TFPG) model that captures the failure-modes, discrepancies and the failure propagation across the entire system. The reasoner uses this model to isolates the most plausible fault-source (component) that could explain the observations i.e. monitors triggered and the CLHM commands issued. The result i.e. faulty component(s) is reported through an aperiodic publisher to the next component - SystemHMResponse Engine - that hosts the system level mitigation strategy.
• SystemHM Response Engine receives the diagnosis results - the set of faulty components - and responds with an appropriate system-level command to mitigate the fault and its effects. This engine hosts a timed state-machine that executes the SLHM mitigation strategy specified by the designer (described later in this section). The updated fault-status of the components in the assembly is used to trigger the SLHM state-machine. The output generated by the state-machine is translated and sent (published) as mitigation commands to the appropriate components.

ACM design tools enable the automatic addition of these three SLHM components to any assembly. During this process, templatized models of these three components are instantiated and interconnected. Furthermore, each functional component in the existing Assembly model is provided an additional publisher: HMPublisher, and consumer: HMConsumer. The publisher is used by the CLHM to feed local detection and mitigation data to the Alarm Aggregator. The consumer is used to receive and execute commands from the SystemHM Mitigation Engine.

3.9 SLHM mitigation strategy
This automated synthesis and weaving of the SLHM into the existing assembly model requires two additional pieces of information from the system integrator - a state machine model of the SLHM strategy for the concerned system/assembly, the deployment details for the SLHM components. The code generators translate this information appropriately into code and configuration information, thereby architecting the entire SLHM layer.

Just like the Component HM model (which captures the component level health manager), the system wide mitigation strategy is expressed using an extension of the ACM State-Machine model. Refer to the ACM-Statemachine.pdf for more information on the ACM state machine models in general. The SHM reacts to fault diagnosis from a system level diagnosis engine and issues appropriate mitigation action.

The table below captures information pertaining to the SHM specific guard conditions that can be expressed in the SHM state machine. These include commands to check the fault-status & operational status of a component.

<table>
<thead>
<tr>
<th>Command</th>
<th>Semantics</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_FAULTY (COMPONENT_NAME)</td>
<td>This command is used to check if a component has been diagnosed as faulty.  Returns true if a component is diagnosed as faulty, false otherwise.</td>
<td>IS_FAULTY(GPS)</td>
</tr>
<tr>
<td>IS_ACTIVE (COMPONENT_NAME)</td>
<td>This command is used to check if a component is in ACTIVE state. Returns true if a component is active, false otherwise.</td>
<td>IS_ACTIVE(GPS)</td>
</tr>
</tbody>
</table>
IS_INACTIVE (COMPONENT_NAME)  This command is used to check if a component is in IN_ACTIVE state. Returns true if a component is inactive, false otherwise.  IS_INACTIVE(GPS)

IS_SEMIACTIVE (COMPONENT_NAME)  This command is used to check if a component is in SEMI_ACTIVE state. Returns true if a component is semi-active, false otherwise.  IS_SEMIACTIVE(GPS)

IS_READY (COMPONENT_NAME)  This command is used to check if a component has executed all the commands sent to it by the system health manager. Returns true if the component has executed all commands, false otherwise.  IS_READY(GPS)

The table below captures the mitigation actions that can be expressed in a SHM state machine. These include commands to START/STOP/RESET a component, and command to rewire a facet-receptacle (provides-requires) connection.

Table 2 ACTION STATEMENTS in SHM

<table>
<thead>
<tr>
<th>Command</th>
<th>Semantics</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>START(COMPONENT_NAME)²</td>
<td>Command issued to start a component. It sets the component’s status to ACTIVE.</td>
<td>START(GPS)</td>
</tr>
<tr>
<td>STOP(COMPONENT_NAME)²</td>
<td>Command issued to stop a component. It sets the component’s status to IN_ACTIVE.</td>
<td>STOP (GPS)</td>
</tr>
<tr>
<td>RESET (COMPONENT_NAME)²</td>
<td>Command issued to reset a component.</td>
<td>RESET(GPS)</td>
</tr>
<tr>
<td>REWIRE (COMPONENT_NAME, INTERFACE_NAME, NEW_PROVIDER_COMPONENT_NAME)</td>
<td>Command issued rewire an interface (parameter 2) in a component (parameter 1) with a suitable provider from another component (parameter3)</td>
<td>REWIRE(NAV_DISPLAY, GET_GPS_DATA, GPS2)</td>
</tr>
</tbody>
</table>

Comment: Rewires interface GET_GPS_DATA in component NAV_DISPLAY to the appropriate provider from the GPS2 component.

² The parameter supplied to these functions could be a subsystem name. If a subsystem name is passed then the command (START/STOP/RESET) would be applied to each component in the subsystem.
3.10 Deliberative Reasoning

Instead of specifying the reactive state machine for performing mitigation at the system-level, one can choose to use the deliberative approach. The deliberative approach uses the deliberative reasoner mentioned earlier in the document. This deliberative reasoner requires the knowledge of different functions provided by the system assembly and how different system components are used to provide those functions.

Following concepts are used to map functional decomposition of the system to the components in the assembly. This information is used by the deliberative reasoner to perform system health management based on the diagnosis information provided by the diagnoser.

3.10.1 Functionalities and functional dependencies

The ACM language allows one to capture the functionalities that are expected to be serviced from the assembly/system. The broader sets of functionalities that are expected from a system are expressed in a “FunctionSheet” model/sheet. This can be created inside a “Functions” folder. In a Function-Sheet the modeler can capture the broad set of functionalities (Function(s)) that are expected from the assembly. The modeler can also express dependencies between the Functions, essentially capturing the higher-level functions and their decomposition into lower-level functions.

The figure below captures the functionalities and their decomposition for the IMU model.
3.10.2 Function Allocation

The next step is to associate these functionalities with the appropriate set of components that provide these functionalities. This model of allocating the appropriate set of components that provide the service associated with a function is captured in in one or more Function-Allocation models. These models can be created as sheets in the Allocations folder. The “Allocations” folder can be created as a sub-folder inside the “Assemblies” folder that hosts the Assembly model(s).

The Function Allocation model holds a reference to one or more Functions. It can also contain references to Components or allow the modeler to define Group(s) of Components. The Function reference is connected to the component reference(s) and Group(s) that are required to provide the service expressed by the Function.

Sometimes assembly model holds redundant components. So the Function can be serviced by a group of components among the redundant set. This redundancy logic can be captured in the Function Allocation model by grouping the components using AND/ XOR/ MOFN logic. The Function Allocation model provides different types of Groups – ANDGroup, XORGroup, and MOFNGroup - to express the appropriate redundancy logic. A Group can hold a set of component references. It can also contain other Groups thereby allowing further nesting of the redundancy logic.
<table>
<thead>
<tr>
<th>Group Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>All the elements of the group are required to provide the functionality</td>
</tr>
<tr>
<td>XOR</td>
<td>Exactly one element of the group is required to provide the functionality</td>
</tr>
<tr>
<td>MOfN</td>
<td>M elements of a total of N are required to provide the functionality</td>
</tr>
</tbody>
</table>

The figure below shows the Function Allocation for “BodyAccelerationMeasurement” functionality in IMU model. At the root level, the function (BodyAccelerationMeasurement) requires one of the Alternatives from the XORGroup. The XOR group has two alternatives- PrimaryAND, SecondaryAND – both of which are AND groups. The figure follows the hierarchy through the AND group PrimaryAND. PrimaryAND has three required MOfN groups – AccelerometerGroup, FilterGroup & VoterGroup. The figure also shows the make-up of the AccelerometerGroup - 6 Accelerometers. The figure shows that the AccelerometerGroup has a minimum number requirement of 4 Accelerometers (attribute on the right bottom).
Figure 12 Functional Allocation for “BodyAccelerationMeasurement” functionality in IMU model.
3.10.3 Component Operational Redundancy
Apart from the function allocation model, that captures the set of components that are required to provide a specific functionality, the assembly/design model captures the dependency between the components. For any component to be functional, its consumer and requires port need to be functional – i.e. obtain service from their publisher and provider ports respectively. In case of a consumer, exactly one (XOR) of the publisher port (and hence it’s associated component) need to be active. In case of a requires port, at least one (MOFN with M=1) of the provider port (and hence it’s associated component) needs to be active.

3.10.3.1 Implicit Operational Redundancy Logic
This rule can be used to derive a Boolean expression of when a component can be truly functional. The implicit rule is that a component is active when all its consumer and requires ports are serviced. The implicit operational rules (OR) for the “NavDisplay” component in Assembly model shown in Figure 12 can be expressed in terms of the operational logic for Requires Port (GPS_DATA_SOURCE) and Consumer Port (DATA_IN).

![Figure 13 Example Assembly Model](image)

Implicit OR-NavDisplay component = AND (OR-Requires Port, OR-Publisher Port)

OR-Requires Port GPS_DATA_SOURCE = MOFN_{M=1}(GPS, GPS2)
OR-Publisher Port DATA_IN = XOR(GPS, GPS2)

Implicit OR-NavDisplay component = AND (MOFN_{M=1}(GPS, GPS2), XOR(GPS, GPS2)) which reduces to

= XOR(GPS, GPS2).

Implicit OR- GPS component = OR-Consumer-Port = Sensor

Implicit OR- GPS2 component = OR-Consumer-Port = Sensor2

Given these operational rules (ORs) derived from the Assembly model, the functional allocation model for the Assembly can be expressed in terms of just the NavDisplay component. In combination with the operational rules (OR), this would imply that either one of the following set of components need to be active to provide the functional requirements

- Sensor/GPS/NavDisplay
- Sensor2/GPS2/NavDisplay
3.10.3.2 Explicit Operational Rules
The ACM modeling language allows the modeler to specify the operational rules for any component in terms of its consumer and requires port. Like the function allocation model, the Operational Rules model for a component can be described using XOR, AND and MOFN groups over the component’s consumer and requires ports.

In the Assembly model described in Figure 13, the GPS component’s Implicit Operational Logic would imply that both Sensor and Sensor2 be functional to service the two consumer ports in the GPS component. This can be overridden by an Operational Rules model within the GPS component expressed in terms of the consumer ports and the requires port (Figure 14). This Explicit Operational Rules (OR) model for the GPS component specifies that M-OF-N, i.e. at least one of the consumer ports should be serviced which translates to the rule that either Sensor or Sensor2 or both should be active for GPS to be active.

3.10.3.3 Disable Generation of Operational Rules
Alternatively, the modeler can suppress the generation of the operational rules for a component by setting the attribute “Generate Implicit Logic” to false.

3.10.4 Deliberative Reasoner and its Variants
In the ACM Runtime, the required functionality, its decomposition, the function allocation and the redundancy information are captured across the following models - Functionality/ Function Decomposition/ Function Allocation/ Component Operational Rules. The Deliberative Reasoner uses this information to suggest valid reconfiguration when faults are detected in component assembly. The reconfigurations are computed such that the functionality affected by the faulty components can be restored using the redundancy information expressed in the function allocation and the component operational rules.
The ACM Runtime provides support for two variants of this reasoning engine –
- **DR** – a Deliberative Reasoner that operates based on the DAG created using only the Function Allocation model. Document (doc\DeliberativeReasoner.pdf) describes the algorithms and the operation of this deliberative reasoner.
- **DR +SAT** - A Deliberative Reasoner that operates using a Satisfiability (SAT) solver. It translates the information in both the Function Allocation and Component Operational Rules into Conjugate Normal Form (CNF) and feeds it to the SAT solver (CryptoMinisat v2.9.1). The output of the SAT solver is translated back into Component activation and deactivation decisions.

### 3.10.4.1 Known Issues with DR, DR+SAT

The purely DAG based approach in DR currently does not take the operational rules of the component into account. It relies solely upon the Function Allocation model. This requires that the function allocation model explicitly specify all the alternative component configurations that are required to service the functional requirements. Further, the DAG based approach in DR may not handle ALT or XOR rules correctly if the same component appears under multiple ALR or XOR Groups (in the Function Allocation model).

The DR+SAT deliberative reasoner does not suffer from the limitations described above. It combines the information from the function allocation model with the operational rules (implicit or explicit) for all the components in the assembly. It converts the derived Boolean expressions into Conjugate Normal Form (CNF) that is then fed to the SAT solver (CryptoMinisat v2.9.1). Since this approach uses the component operational rules from the assembly model, the function allocation model can be specified in a concise format, dealing with only the most essential components for any function. The DR+SAT based approach has some known issues where in it does not handle the M-OF-N cases in a manner that is consistent with the other approach. The DR+SAT currently guarantees that only the minimal (M) set of constraints in M-OF-N are satisfied. This implies that out of N components it will turn active only M components even though more than M components can be turned active.
3.11 Deployments

Once the assembly is specified logically, the integrator provides the physical deployment details. This stage also specifies the temporal separation requirements. That is, they specify the various modules or processors to be used in the system and the partitions. Integrators then specify the allocation of components to each partition.

This information is provided under the deployment folder. This folder holds Deployment and Platform definition. A new Deployment/Platform can be added by right-clicking on the folder and choosing the appropriate option under “Insert Model”.

The platform model can contain Module(s) which in turn contains Partitions. Each module is linked to one physical or virtual machine. One has to specify the modules IP address or its unique DNS name. It is possible to set the a property in the platform such that the system executes in a synchronized hyperperiod across all modules. If this option is chosen, one of the modules in the platform has to be designated as the system module.

Deployment model contains PartitionRefs - reference to the Partitions defined in the Platform and ComponentInstances - reference to the ComponentRefs defined in the Assembly model. PartitionRef may be connected to one or ComponentInstances to indicate the Components hosted by the Partition. The PartitionRefs are further qualified by attributes pertaining to Partition Deployment - Period and Duration.

In the current release, it is important that all Partitions, referred by the PartitionRefs in a Deployment model, are contained inside the same Module. With this restriction it is suggested that in order to keep the design clean, each Deployment folder contain one and only one Platform and Deployment model.

![Module and Partitions](image)

Figure 16 Details of a Platform Model from GPS No Fault.mga
Figure 17 Deployment Model for GPS No Fault.mga
4. ACM Modeling Tools Setup (windows)

4.1 System Requirements

Supported Operating Systems
- Microsoft Windows XP
- Microsoft Windows Vista (not tested)
- Microsoft Windows 7

Hardware Requirements
- No additional requirements beyond those imposed by Windows.

The following prerequisites are needed by portions of the ACM Modeling Tools. 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 the prerequisites with limited functionality.

a. Generic Modeling Environment (GME) 10.11.24 or newer
   - http://repo.isis.vanderbilt.edu/tools/get_tool?GME
b. Python 2.6.x for Windows

4.2 Installation

The ACM Modeling Tools installer is available in the repository: ACM_11_8.zip. Unzip the file and launch executable and follow prompts.

4.3 Questions/Issues

Please direct inquires about the ACM Modeling Tools to the Model Based Software Health Management Users list at http://list.isis.vanderbilt.edu/mailman/listinfo/mbshm-users

4.4 Model Storage Formats

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. All sample models included in the installation package are in the XME format.

----------

3 http://www.isis.vanderbilt.edu/Projects/gme/
5. Working with the Modeling Environment

5.1 Create 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)

5.2 Open a Model
- Open GME ("Start->All Programs->GME->GME")
- Select "File->Open Project..."
- Choose a "*.mga" file to open

5.3 Import Model (XME Format)
For an explanation of model formats see section 4.4
- Open GME ("Start->All Programs->GME->GME")
- Select "File->Import XML..."
- Choose a "*.xme" file to open
- Click "Next >"
- Choose a file name (i.e. MyModel.mga)

5.4 Export Model (XME Format)
For an explanation of model formats see section 4.4
- Open the "*.mga" file to be exported
- Select "File->Export XML..."
- Choose a file name (i.e. MyModel.xme)
- Click "Save"
6. Samples

ACM-Modeling Tools includes several sample models. Locate these models by selecting: “Start->All Programs -> ACM-> Samples”. This opens the folder with examples. By default this folder is read-only to non-administrators. You should copy the samples folder to a new location that doesn’t have write protection. Currently, these tools come with following samples:

6.1 GPS-No-Fault
This is a three component nominal example. First component is the sensor component that publishes a data type SensorOutput periodically every 4 sec. The second component is the GPS component that requires the input from a Sensor and then filters it and updates its internal data structure. It publishes the updated information through a port aperiodically. The GPS has the ability to be queried remotely via an RMI call for the current GPS value. The last component is a Navigation Display component, which can receive an updated SensorOutput and also query a remote GPS interface. In the nominal model, all components are connected and deployed on a three partition system, one for each component. The README.txt file in this folder describes how to setup and run this experiment.

6.2 GPS-Sensor-No-Alarm.
This model is the same as No fault. However, the component ports are instrumented for anomaly detection. The assembly model is associated with a system health manager. The deployment includes 4 Partitions - 1 for each component + 1 for SHM. The provided business logic code is written such that it introduces a fault in the Sensor publisher. Due to this, two alarms are raised and detected GPS-Consumer-Validity-Violation and NavDisplay-Consumer-post-condition-violation. These are sent to the SHM, which during runtime identifies the sensor as the faulty component. Subsequently, SHM resets the sensor component. The README.txt file in this folder describes how to setup and run this experiment.

6.3 GPS-Sensor-Alarm
This is similar to the example above. Business logic is also same. Deployment is also same. The differences include

1. Sensor’s publisher port monitors for user code exception and reports it to the local component health manager as USER_CODE_FAILURE error. In this example, an additional alarm is raised Sensor-Publisher-User-Code-Error.
2. The model includes a Function Allocation model which can be used by the Deliberative Reasoner for issuing the appropriate reconfiguration commands when components are diagnosed as faulty.
3. The mitigation action can be performed either with the State-Machine modeled in SHM model or by using the Deliberative Reasoner.
The README.txt file in this folder describes how to setup and run this experiment with

- The Mitigation using the State Machine modeled in SHM.
- The mitigation using the Deliberative Reasoner algorithm – DR
- The mitigation using the Deliberative Reasoner algorithm – DR +SAT.
- The command-line DRExec tool to study the working of the Deliberative Reasoner (DR, DR+SAT)

6.4 ADIRU – Emulation of Portion of Boeing 777 ADIRU Unit
This model was the basis of a case study presented at SEAMS 2011. This case study approximately emulated the working of the Boeing 777 Air Data Inertial Reference Unit (ADIRU). This model shows how SHM architecture can be used to detect, diagnose, and mitigate the effects of component-level failures such that the system-wide functionality is preserved. The detailed case study report is included within the sample folder.

6.5 IMU – Inertial Measurement Unit.
This experiment shows the demonstration of a large inertial measurement unit system constructed using ACM Modeling tool. Just like the sample 3 above (GPS-w-sensor-user-code-exception), this model includes Function Allocation models. So the mitigation commands could be issued either by using the Deliberative Reasoner or through mitigation commands modeled in the state-machines of the SHM model. The baseline code is included in the generated code. You should regenerate the code from IMU-Model included in the parent folder over this generated code. This will fill the deployment code. Later on you can transfer the generated code to Linux and compile. Details of this model and an associated case study are included in the samples folder. The README.txt file in this folder describes how to set up and run the experiment (and its variants) associated with this example.

All described above samples include business logic code. The utility of business logic code will become clear as we discuss the interpretation, building and execution on the Linux Runtime.

6.6 Deliberative Reasoner Examples
As explained earlier, the Deliberative Reasoner uses two different algorithms

1. DR : Uses the DAG formed from the Function Allocation model. Currently it does not consider the Explicit and Implicit Component Operational Rules from the Component model and the Assembly model.
2. DR + SAT : Uses the DAG formed from Function Allocation model, Explicit component Operational Rules model, Implicit Operational Rules from the Assembly model. Further, it translates the DAG information into Conjugate Normal Form (CNF) and uses a SAT solver to solve the problem.

The following sample models include a Function Allocation model and hence can demonstrate the workings of the Deliberative Reasoner (DR or DR+SAT).

- GPS-Sensor-Alarm
  - This example illustrates Deliberative Reasoning with both DR and DR+SAT.
Models in GPS-Sensor-Alarm/DR_SAT_Examples
- The examples in this folder illustrate the Deliberative Reasoner using only DR+SAT.

The Deliberative Reasoner can be demonstrated using two different approaches

1. **With ACM Runtime**
   When the ACM model includes a Function Allocation Model, the interpreter generates the required code to host the Deliberative Reasoner (DR or DR+SAT) in the Mitigation Component (ResponseEngine Component) of the Systems Health Management (SHM).
   - The build process needs to be setup with the appropriate flags to autogen.sh to run the mitigation engine with DR or DR+SAT.
     - To run with DR set up the build process with the command
       - `./autogen.sh --with-drengine`
     - To run with DR+SAT set up the build process with the command
       - `./autogen.sh --with-drengine --with-sat`

   During runtime the results of the Diagnosis Engine component are fed to the Mitigation Engine which feeds the list of faulty components to the DR-engine. Depending on the configuration (DR or DR+SAT), the DR-Engine uses the appropriate algorithm and produces the reconfiguration commands that are issued to the appropriate components in the Assembly.
   - The results of the DR-Engine are logged into a html file (dr.htm) in the module hosting the SHM partition.

   **Examples**
   More information on setting up and running these experiments can be found in the README files of the following samples:
   - GPS-Sensor-Alarm
   - GPS-Sensor-Alarm/DR_SAT_Examples/redundancy_explicit
   - GPS-Sensor-Alarm/DR_SAT_Examples/redundancy_implicit

2. **DREEXEC : Command-line DR tool**
   The ACMTools installation includes a command line utility – DREEXEC – that can be used to test the operation of the Deliberative Reasoning algorithms (DR or DR+SAT). It can be run with the following command:
   ```
   DRExec <Model File (Model.xml) > <Deployment Folder Name> <Output File> <Fault list File> [Optional Flags]
   ```
   - **Model File (Model.xml)** – This file (Model.xml) is produced by the interpreter. It is present in the module folder hosting the SHM partition.
   - **Deployment Folder Name** – Name of the Deployment folder in the model that is being tested.
   - **Output file** – The name of the file (.htm) where the DR-Engine results are logged.
   - **Fault list File** – The name of the file that holds the list of components that turn faulty.
     - Each line holds the name of one faulty component.
When a component is present inside a subsystem, then the component name is specified as “SUBSYSTEM_NAME/COMPONENT_NAME”

**Optional Flags**
- `-s`: Run with SAT solver (DR+SAT)
- `-r`: Invoke DR-Engine to check for Reconfiguration commands after each line in the fault file i.e. after each faulty component.
- `-v`: Verbose mode.

**Examples**
More information on running experiments with DRExec can be found in the README of the following samples:
- GPS-Sensor-Alarm
- GPS-Sensor-Alarm/DR_SAT_Examples/redundancy_explicit
- GPS-Sensor-Alarm/DR_SAT_Examples/redundancy_implicit
- GPS-Sensor-Alarm/DR_SAT_Examples/offline_samples

### 6.7 Demonstration Movies
Following videos can be downloaded from the ACM Tools website.

#### 6.7.1 SimpleGPSDemo
This video demonstrates how to create an ACM project in GME. It covers the simple GPS (no-fault) sample which is supplied with the ACM installation. It also covers some of the basics of how to use GME. Demonstrates the creation/definition of data types, interfaces, components, assemblies, and deployments. Also demonstrates running the interpreter and supplying user-generated code.

#### 6.7.2 ExecutionOfSimpleGPSDemo
This video demonstrates how to execute the ACM-interpreted deployment in Linux (using the GPS-no-fault example). It demonstrates setting up the environment in the terminal, building the system, modifying the code (using eclipse), setting up/changing the partition scheduling (using eclipse), and executing the system.

#### 6.7.3 ExecutionOfIMUDemo
This video demonstrates running the event simulator in GME (using the ACM-provided IMU sample). It describes setting up the event-sim (using the sample files) and how the environment labels the alarms, faults, and active/inactive components. It runs through the IMU sample and shows the propagation of faults through the system, while describing how the SLHM manages the fault mitigation.

#### 6.7.4 CHM_SHM
This video demonstrates how to create a Component Health Manager (CHM) and a System Level Health Manager (SLHM). It shows how to define different conditions which can be triggered, how to define the state transition of the CHM/SLHM, and how to define the guard conditions and response actions associated with the transitions.
7. Steps to generate and execute models using ACM Tool

Following are the steps for executing models using ACM tool. The discussion below is with reference to the GPS-NO-Fault model. ($ACM/models/GPS-No-Fault/GPS-No-Fault.xme)

7.1 Open Model:
After installing the ACM Tool Suite, open the GPS-NO-Fault model by importing the file ($ACM/models/GPS-No-Fault/GPS-No-Fault.xme into GME. The figure below shows the screen shot of the model in GME.

- The window on the right shows a tree browser that shows the contents of the model. This includes a library of the models of data-types (Types), interfaces and components.
- The top window on the left shows the assembly model created from the instances of components defined in the component library.
- The next two windows in the left show the three Partitions associated with a module and the deployment of the components to individual Partitions.

Figure 18 GPS No Fault model
7.2 Interpret

The ACM model interpreter interprets the models inside a Deployment folder. The interpreter is initiated/started by opening any of the deployment models (inside the appropriate Deployment folder) and clicking on the interpreter icon as shown in the figure below. In this discussion, the “3Partition” deployment model inside the “3Partition” deployment folder is interpreted. In the subsequent discussions, $DeploymentName refers to the name of the deployment folder that is interpreted.

While interpreting the model, specify the target folder to dump the generated code. The interpreter generates contents into three top level folders

- gen_datatypes
  - contains files (.h, .idl) specific to data types
- gen_components
  - contains a separate folder for each component
  - each component subfolder contains files specific to the component (.h, .cc, .cpp, .idl)
- gen_deployments
Contains sub-folder for each Deployment Scenario i.e. the Deployment folder name (gen_deployments/$DeploymentName)

- gen_deployments/$DeploymentName contains sub-folders for each Module in the Deployment.

- Each Module sub-folder contains
  - a separate folders for each Partition hosted in the Module
  - a configuration file for the module
  - a start-up script (startmodule.sh) to start the partitions in the module

- Each Partition sub-folder contains the files (.h,.cpp) associated with the Partition.

- Script to start and view the generated files through Eclipse (start_eclipse.sh)

### 7.3 Run Time Setup:

The following steps helps set up the environment to edit/ build/ execute the project

During the course of this discussion it will be assumed that ACM-Run-Time is installed in the directory $ACMINSTALL and the interpreter generated code is copied into the folder $PRJDIR. $DeploymentName refers to the folder-name of the Deployment that was interpreted.

a. Copy the generated files into a Linux machine i.e. to $PRJDIR
b. The following command sets up the ACM related environment variables
   
   ```
   source $ACMINSTALL/bootstrap.sh
   ```
c. The following command to sets up environment variables for this experiment
   
   ```
   cd $PRJDIR /gen_deployments/$DeploymentName
   source ./prepareExperiments.sh
   ```
d. Following command sets up the make-files
   
   ```
   cd $PRJDIR /gen_deployments/$DeploymentName
   chmod +x ./autogen.sh
   ./autogen.sh
   ```
e. **Eclipse Environment:** Following command opens the generated files in an Eclipse environment.
   
   ```
   cd $PRJDIR /gen_deployments/$DeploymentName
   ./start_eclipse.sh.
   ```

   Then import projects into eclipse workspace using the existing projects option. Then, point to the working directory. Eclipse should show you the valid projects. **Do no select copy files to workspace.**

Figure below shows a screen-shot of the Eclipse project.
7.4 Deliberative Reasoner

7.4.1 Run Time Setup

In order to use the Deliberative Reasoner to issue the appropriate Reconfiguration Commands (when one or more components are diagnosed as faulty), the setup process described above needs a small change.

Step (d) above where autogen.sh is run to set up the make files, needs to be slightly modified to pass command-line parameter that suggest the use and integration with Deliberative Reasoner (DR).

Run the following command to set-up with Deliberative Reasoner (DR)

```
./autogen.sh --with-drengine
```

Note: Using Deliberative Engine requires that the libraries associated with the Deliberative Reasoner have been installed in the ACM installation directory. Please refer to section 2.4 for more detail.

7.4.2 Offline Experimentation

The ACM Modeling Tools-Suite in Windows contains an executable (bin/DRExec.exe) that allows running the Deliberative Reasoner in an offline mode i.e. without integrating with the ACM RunTime environment.

This executable takes as input

- the xml file generated from the ACM Model Interpreter (Model.xml in the Deployment folder),
- the name of Deployment model
- a log file (html) that will contain the output commands of the Deliberative Reasoner
- an input file with the list/sequence of faulty components

Examples of running this offline Deliberative Reasoner can be found in the following sample models folder

- GPS/GPS-Sensor-Alarm
- IMU
These steps can be performed alternatively in other editors i.e. emacs, vi etc.
1) **Edit (code/schedule):** Once the files are opened in Eclipse, they can be edited from within the environment.

a. **Edit Code:** With regards to the generated code in the gen_component folder and its component specific sub-folders, the users can edit any file with the name - *_impl.h, *_impl.cc

Each of these files has specific marked sections for the users to add their code. Code written in these marked portions will be preserved even if the interpreter is used to re-generate the code.

These sections are marked as

```cpp
//@REGION NAME - User code begins
//fill in user-code here
//@REGION NAME - User code ends
```

The following figure shows a User-code section in GPS component. It corresponds to the user-code for the consumer data_in in the GPS component (GPS_impl::handle_data_in function)
b. **Edit Schedule:** Each module-folder has a configuration file – Configuration.cfg.in. The User needs to fill in the schedule information in Configuration.cfg.in file(s).
2) **Build:** Building the source tree can be done from within Eclipse (Menu: Project/Build All). Alternately, one can use open a terminal/shell and setup the environment as described in step (3) above. Then type

```bash
cd $PRJDIR/gen_deployments
make
```

3) **Execute:** In order to execute the project, the user will need root privileges. The steps for single and multi-module experiments are listed below.

**Executing Single Module Experiment**

a. Login as root. Any of the following commands can be used for that
   ```bash
   sudo -s
   su
   ```

b. Set-up the ACM Environment variables.
   ```bash
   source $ACMINSTALL/bootstrap.sh
   ```

c. Change Directory to the Module folder.
   In this project there is only one module (aModule) and it is inside the deployment folder(3Partition). So you will type
   ```bash
cd $PRJDIR/gen_deployments/3Partition/aModule
   ```
d. Type the following command to start the experiment
   ```bash
./startmodule.sh
   ```

**Executing Multi-Module Experiment**
In a multi-module experiment, for each machine/module involved in the exercise,

a. Login as root.

b. It is assumed that ACM RunTime is installed in each machine in the folder $ACMINSTALL. Set-up the ACM Environment variables, by opening a terminal/shell and typing in source $ACMINSTALL/bootstrap.sh

c. Change Directory to the appropriate Module folder (/$MODULENAME) that is to be run on the machines. The module folders are present within the deployment folder (/$DEPLOYMENTNAME) $PRJDIR/gen_deployments/$DEPLOYMENTNAME/$MODULENAME.

d. Type the following command to start the experiment in each Module.

./startmodule.sh

Note: In case the experiments are run in a synchronized manner, the module that is marked as the System Module in the model needs to be started first.

4) **Information/Log Files:**

   Each Partition in a Module is hosted on a separate Linux Process. A Module-Manager process in each module is responsible for managing these Partitions (Linux Processes).

   In every Module, a log file is created for each Partition and the Module Manager process.