FUSION FOR ENERGY

MARTe2 Users Meeting
RealTime Applications II

Andre Neto, Filippo Sartori

May, 2019

Signal properties

Type	The signal type as any of the supported Types or a structure type.
DataSource	The name of the DataSource from where the signal will read/written from/to.
Frequency	Only meaningful for input signals. The frequency at which the signal is expected to be produced (at most one signal per real-time thread) may have this property set.
Trigger	Only meaningful for output signals. Trigger the DataSource when this signal is written.
NumberOfElements	The number of elements (1 if the signal is a scalar).
NumberOfDimensions	The number of dimensions (0 if scalar, 1 if vector, 2 if matrix).

Signal properties

Samples	The number of samples to read from a DataSource. This number defines the number of samples that the DataSource shall acquire for each control cycle. Note that each sample may contain an array.
Ranges	In the case of a vector read/write only a subset. The format is a matrix, indexed to zero, of the ranges that are to be read (e.g. {{0, 1}, {3, 5}} would read elements 0, 1, 3, 4 and 5 of the array).
Alias	The name of the signal in the DataSource (which can be different from the name of the signal in the GAM).
Default	The default value to be used in the first control cycle (if needed, i.e. if it depends from a value of the previous cycle).

Signal properties

```
+GAMDisplay = {
 Class = IOGAM
  InputSignals = {
   Counter = {
    DataSource = DDB1
    NumberOfElements = 1
    NumberOfDimensions = 0
    Type = uint32
   }
   GainCounter = {
    DataSource = DDB1
    Type = uint32
   }
   State1_Thread1_CycleTime = {
    Alias = State1.Thread1_CycleTime
    DataSource = Timings
    Type = uint32
   }
   Signal3 = {
    DataSource = DDB1
    Type = uint32
    Ranges = {{0,0}, {2, 2}}
    NumberOfElements = 3
    NumberOfDimensions = 1
   }
```

```
+Timer = {
 Class = LinuxTimer
 SleepNature = "Default"
 Signals = {
  Counter = {
   Type = uint32
  }
  Time = {
   Type = uint32
  }
 }
}
```

Registered types as signals

Structured signals

GAMs can also use structured types as signals.

```
struct ModelGAMExampleStructInner1 {
  MARTe::float32 f1;
  MARTe::float32 f2;
  MARTe::float32 f3[6];
};
struct ModelGAMExampleStructSignal {
  MARTe::uint32 u1;
  ModelGAMExampleStructInner1 s1;
  ModelGAMExampleStructInner1 s2;
};
```

```
...
InputSignals = {
  Signal1 = {
    DataSource = DDB1
    Type = ModelGAMExampleStructSignal
  }
}
```

```
...
InputSignals = {
  Signal1 = {
    u1 = {
      DataSource = DDB1
      Type = uint32
    }
    s1 = {
      f1 = {
        DataSource = DDB1
        Type = float32
      }
      f2 = {
        DataSource = DDB1
        Type = float32
      }
      f3 = {
        DataSource = DDB1
        Type = float32
        NumberOfDimensions = 1
        NumberOfElements = 6
      }
    }
```

Note

Structure will be automatically expanded into the equivalent signal configuration structure

Registered types

```
...
InputSignals = {
  Signal1 = {
    DataSource = DDB1
    Type = ModelGAMExampleStructSignal
    Defaults = {
      Signal1.s1.f1 = 2
      Signal1.s1.f2 = 3
      Signal1.s1.f3 = {1, 2, 3, 4, 5, 6}
      Signal1.s2.f1 = -2
      Signal1.s2.f2 = -3
      Signal1.s2.f3 = {-1, -2, -3, -4, -5, -6}
    }
    MemberAliases = {
      //Rename of a structured member
      Signal1.s2.f2 = Signal1.s2.g2
    }
  }
...
```

Property	Meaning
MemberAliases	The name of the structured member signal in the DataSource (which can be different from the name of the signal in the GAM).
Defaults	The default value for a given member of the structure.

Registering types

Registers a structure as defined by a StructuredDataI.

- Each node is a structure member and shall have the field Type defined.
  - The name of the Object is the name of the structure to register.

```
+MyTypes = {
  Class = ReferenceContainer
  +MyStructEx1 = { //name of the structured type to register.
   Class = IntrospectionStructure
   Field1 = {
    Type = uint32
    NumberOfElements = 1
   }
   Field2 = {
    Type = float32
    NumberOfElements = {3, 2} //3x2 matrix
   }
  }
  +MyStructEx2 = { //name of the structured type to register.
   Class = IntrospectionStructure
   Field1 = {
    Type = MyStructEx1
    NumberOfElements = {3, 2, 1} //3x1x2 matrix of MyStructEx1
   }
  …
```

More rules!

- The nodes **Functions**, **Data**, **States** and **Scheduler** shall exist;
- At least one **GAM** shall be declared;
- At least one **DataSource** shall be declared;
- Exactly one **TimingDataSource** shall be declared;
- At least one **state** shall be declared;
- For **each state**, at **least one thread** shall be declared;
- For **each thread**, at **least one function** (GAM) shall be declared;

Signal rules

- For every thread
  - The input port of each GAM or DataSource shall be connected to exactly one signal
    - From another GAM or from a DataSource
  - The output port of a given GAM or DataSource may be connected to zero or more signals (in another GAM or DataSource);
  - At most one signal shall define the property Frequency (i.e. at most one synchronisation point per thread).

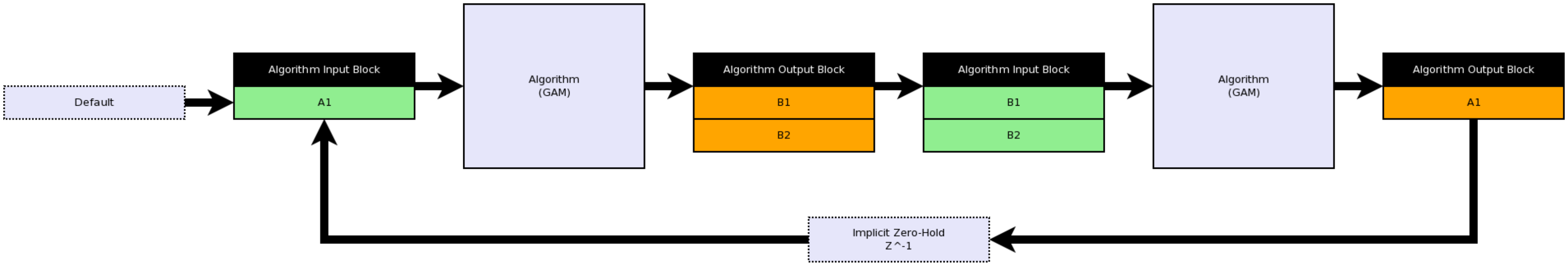
Signal rules

- The properties of each signal shall be fully consistent between the signal producer and the signal consumer
  - Type
  - Number of elements
  - Number of dimensions
- If the number of elements is not defined, one is assumed;
- If the number of dimensions is not defined, zero is assumed (scalar signal);
- If no Default value is specified, zero is assumed.
- The signal type shall be defined either by the signal producer or by one of the signal consumers:

Signal rules

Zero-hold

- If a GAM requires a signal that is produced by a subsequent GAM, an implicit zero-hold is introduced in the cycle and the signal is initialised to its Default value.



Dynamic type discovery

```
A = +GAM1 = {
  ...
  InputSignals = {
    A1 = {
      DataSource = DS1
      NumberOfElements = 2
    }
  }
    ...
}
+DS1 = {
  ...
  Signals = {
    A1 = {
      Type = uint32
    }
  }
}
```

=>

```
A = +GAM1 = {
  ...
  InputSignals = {
    A1 = {
      DataSource = DS1
      NumberOfElements = 2
      Type = uint32
      NumberOfDimensions = 0
    }
  }
    ...
}
+DS1 = {
  ...
  Signals = {
    A1 = {
      Type = uint32
      NumberOfElements = 2
      NumberOfDimensions = 0
    }
  }
}
```

Type must be defined at least once

```
A = +GAM1 = {
   ...
   InputSignals = {
     A1 = {
       DataSource = DS1
       NumberOfElements = 2
     }
   }
     ...
}
+DS1 = {
   ...
   Signals = {
     A1 = {
       NumberOfElements = 2
     }
   }
}
```

=>

```
Fails

Signal A1 type not defined!
```

Types must be consistent

```
A = +GAM1 = {
  ...
  InputSignals = {
    A1 = {
      DataSource = DS1
      Type = uint32
      NumberOfElements = 2
    }
  }
    ...
}
+DS1 = {
  ...
  Signals = {
    A1 = {
      NumberOfElements = 4
    }
  }
}
```

=>

Fails

The definition of the signal is not consistent!

Ranges

Ranges

Allow to access elements of an array, e.g.: **Ranges = {{0,0}, {2, 2}}**

```
A = +GAM1 = {
  ...
  InputSignals = {
    A1 = {
      DataSource = DS1
      NumberOfElements = 4
      Ranges = {{0,1}}
    }
  }
    ...
}
+DS1 = {
  ...
  Signals = {
    A1 = {
      Type = uint32
      NumberOfElements = 4
    }
  }
}
```

=>

```
A = +GAM1 = {
  ...
  InputSignals = {
    A1 = {
      DataSource = DS1
      NumberOfElements = 4
      Type = uint32
      NumberOfDimensions = 0
    }
  }
    ...
}
+DS1 = {
  ...
  Signals = {
    A1 = {
      Type = uint32
      NumberOfElements = 4
      NumberOfDimensions = 0
    }
  }
}
```

RealTimeApplication Initialisation

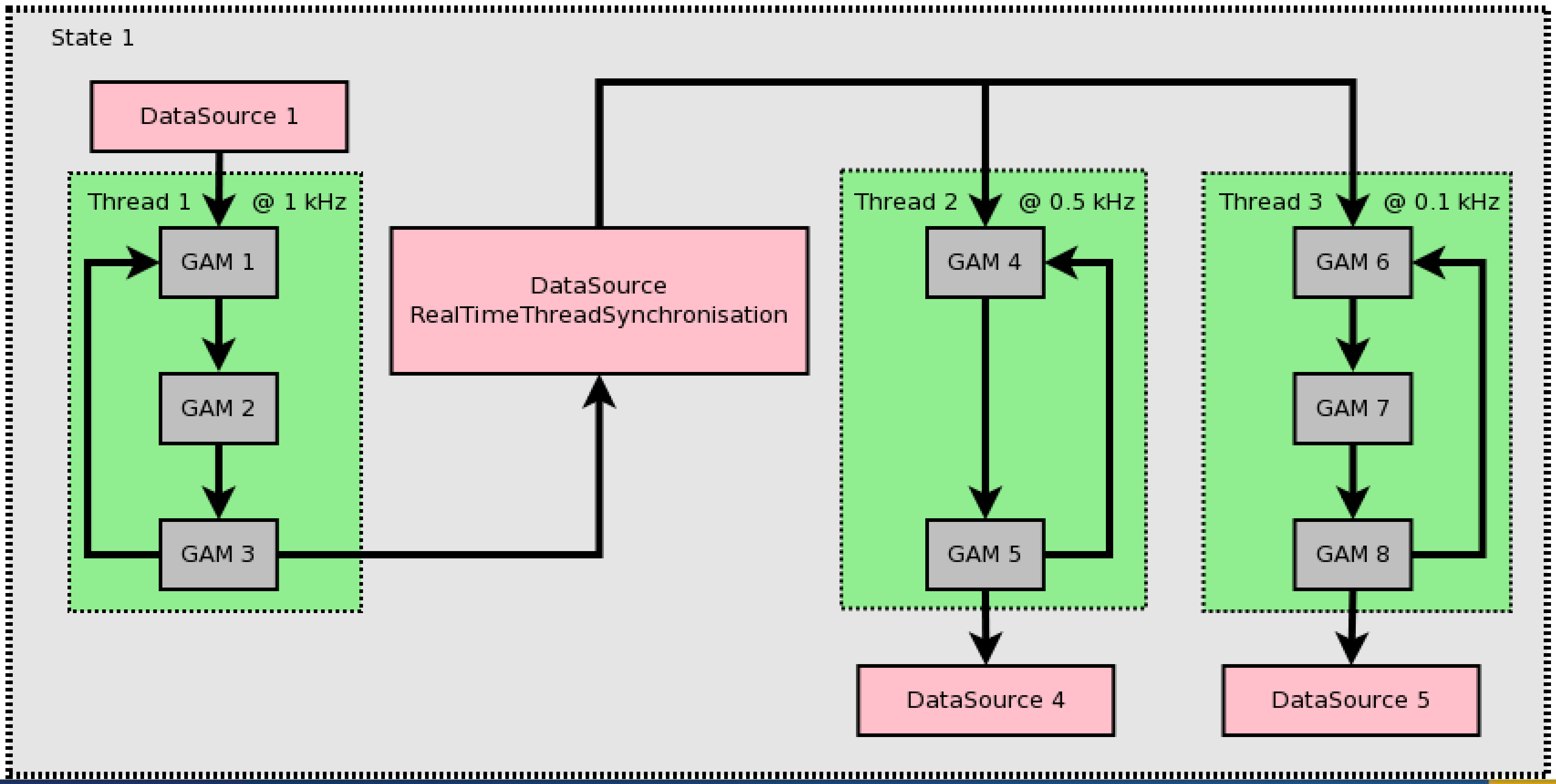
- Part 1
  - Calls **Initialise** on all components
  - Components expected to read properties from configuration source
  - GAMs/DataSources should not make any assumptions on inputs/outputs
- Part 2
  - All rules validated
  - All signals resolved (who reads/write from/where)
  - Calls **Setup** on the GAMs
  - Calls **SetConfigurationDatabase** on DataSources
  - GAMs and DataSources should verify if inputs/outputs are compliant with requirements

Synchronising multiple threads

RealTimeThreadSynchronisation

Performed using the **RealTimeThreadSynchronisation** DataSource component.

RealTimeThreadAsyncBridge component also allows to exchange data between threads without an explicit synchronisation point. This means that the consumer threads will use the latest available data.



Synchronising multiple threads

```
+GAMT1TSynchOut = {
  Class = IOGAM
  InputSignals = {
    GainCounter1Thread1 = {
      DataSource = DDB1
      Type = uint32
    }
    GainCounter2Thread1 = {
      DataSource = DDB1
      Type = uint32
    }
  }
  OutputSignals = {
    GainCounter1Thread1 = {
      DataSource = RTThreadSynch
      Type = uint32
    }
    GainCounter2Thread1 = {
      DataSource = RTThreadSynch
      Type = uint32
    }
  }
}
```

```
+GAMT1T2Interface = {
     Class = IOGAM
     InputSignals = {
       GainCounter1Thread1 = {
         DataSource = RTThreadSynch
         Type = uint32
         Samples = 2 //Run at half the frequency of thread 1
       }
       GainCounter2Thread1 = {
         DataSource = RTThreadSynch
         Type = uint32
         Samples = 2 //Run at half the frequency of thread 1
       }
     }
     OutputSignals = {
       GainCounter1Thread2 = {
         DataSource = DDB1
         Type = uint32
         Samples = 1
         NumberOfDimensions = 1
         NumberOfElements = 2 //2 elements for each cycle (as it waits for 2 samples)
       }
       GainCounter3Thread2 = {
         NumberOfDimensions = 1
         NumberOfElements = 2 //2 elements for each cycle (as it waits for 2 samples)
       }
```

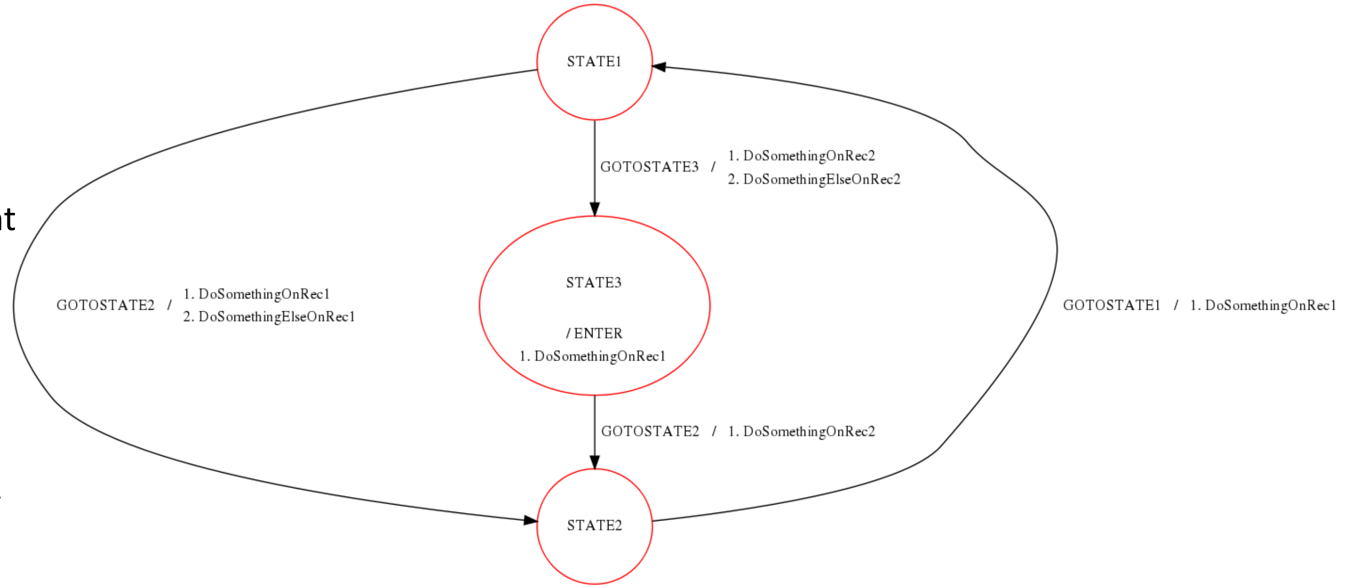
StateMachine

- Key component which is used in many MARTe applications to synchronise the application state against the external environment
- Each state contains one, or more, **StateMachineEvent** elements.
- The StateMachine can be in one (and only one) state at a given time.
- Upon receiving of a Message:
  - StateMachine verifies if the Message function is equal to the name of any of the declared StateMachineEvent elements **for the current StateMachine state**.
  - Sends a configurable set of Messages before changing state

Note
If a state change requests arrives while the state is being changed, this request will be queued and served once the previous state transition is completed.

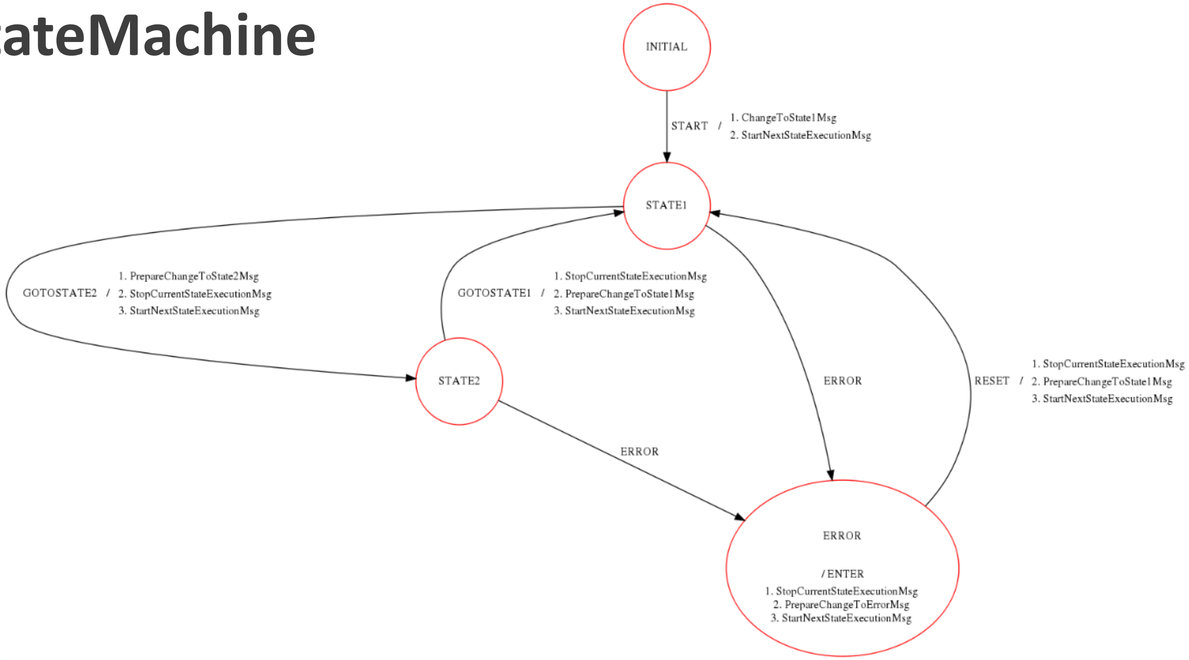
StateMachine

```
+StateMachineExample1 = {
        Class = StateMachine
        +STATE1 = {
          Class = ReferenceContainer
          +GOTOSTATE2 = {
            Class = StateMachineEvent
            NextState = STATE2
            NextStateError = STATE1
            Timeout = 0
            +DoSomethingOnRec1 = {
              Class = Message
              Destination = Receiver1
              Mode = ExpectsReply
              Function = Function1
              +Parameters = {
                Class = ConfigurationDatabase
                param1 = 2
                param2 = 3.14
              }
            }
            +DoSomethingElseOnRec1 = {
              Class = Message
              Destination = Receiver1
              Mode = ExpectsReply
              Function = Function0
            }
          }
```



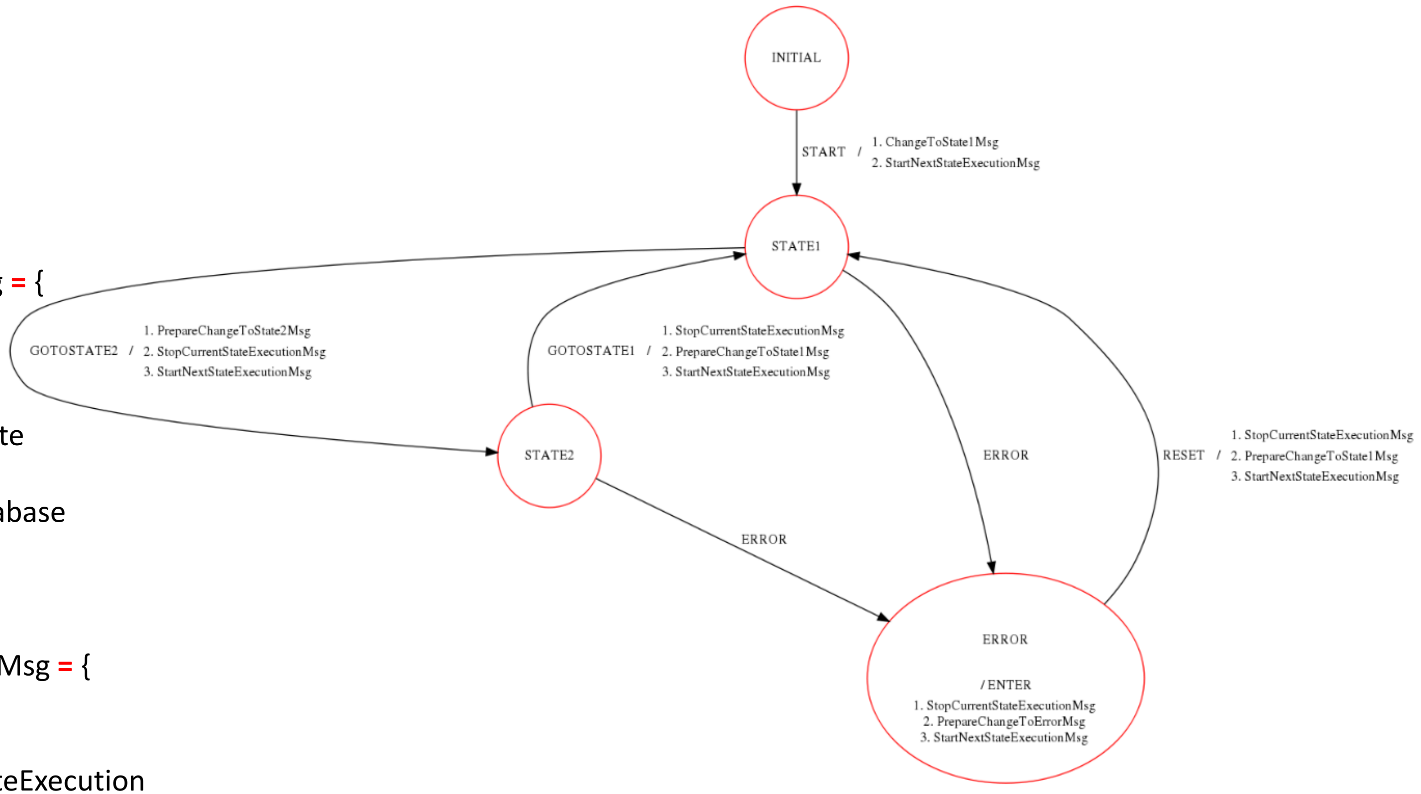
RealTimeApplication state change

- State can be changed by calling the methods **PrepareNextState**, **StopCurrentStateExecution** and **StartNextStateExecution** on the RealTimeApplication
  - Methods are registered as RPC functions and thus can be triggered using the messaging mechanisms
  - Typically the interface to the state changing mechanism is provided by the **StateMachine**



RealTimeApplication state change

```
+StateMachine = {
  Class = StateMachine
+STATE1 = {
    Class = ReferenceContainer
    +GOTOSTATE2 = {
      Class = StateMachineEvent
      NextState = "STATE2"
      NextStateError = "ERROR"
     +PrepareChangeToState2Msg = {
        Class = Message
        Destination = TestApp
        Mode = ExpectsReply
        Function = PrepareNextState
        +Parameters = {
          Class = ConfigurationDatabase
          param1 = State2
        }
      }
      +StopCurrentStateExecutionMsg = {
        Class = Message
        Destination = TestApp
        Function = StopCurrentStateExecution
        Mode = ExpectsReply
      }
      +StartNextStateExecutionMsg = {
        Class = Message
        Destination = TestApp
        Function = StartNextStateExecution
        Mode = ExpectsReply
      }
      ...
```



time elapsed from the beginning of the cycle until this GAM_NAME has finished its execution

Measuring performance

- Each RealTimeApplication will automatically add to the TimingDataSource the following signals:
  - STATE_NAME.THREAD_NAME_CycleTime
    - STATE_NAME = name of the state where the thread is running
    - THREAD_NAME = name of the RealTimeThread object instance
  - GAM_NAME_ReadTime
    - Time elapsed from the beginning of the cycle until all the input brokers for this GAM_NAME have been executed
  - GAM_NAME_WriteTime
    - Time elapsed from the beginning of the cycle until all the output brokers for this GAM_NAME have been execute
  - GAM_NAME_ExecTim
    - Time elapsed from the beginning of the cycle until this GAM_NAME has finished its execution

GAM API useful methods

virtual bool Setup ()=0

uint32 GetNumberOfInputSignals () const

uint32 GetNumberOfOutputSignals () const

bool GetSignalName (const SignalDirection direction, const uint32 signalIdx, StreamString &signalName)

bool GetSignalIndex (const SignalDirection direction, uint32 &signalIdx, const char8 *const signalName)

TypeDescriptor GetSignalType (const SignalDirection direction, const uint32 signalIdx)

bool GetSignalNumberOfElements (const SignalDirection direction, const uint32 signalIdx, uint32 &numberOfElements)

https://vcis-jenkins.f4e.europa.eu/job/MARTe2-docs-master/doxygen/classMARTe_1_1GAM.html

DataSource API useful methods

```
virtual bool      Synchronise ()=0

virtual bool      AllocateMemory ()=0

virtual bool      GetSignalMemoryBuffer (const uint32 signalIdx, const
uint32 bufferIdx, void *&signalAddress)=0

virtual const char8 *    GetBrokerName (StructuredDataI &data, const
SignalDirection direction)=0

virtual bool      PrepareNextState (const char8 *const
currentStateName, const char8 *const nextStateName)=0
```

https://vcis-jenkins.f4e.europa.eu/job/MARTe2-docs-master/doxygen/classMARTe_1_1DataSourceI.html

DataSource API useful methods

FUSION FOR ENERGY

```
uint32  GetNumberOfSignals () const

bool    GetSignalIndex (uint32 &signalIdx, const char8 *const signalName)

bool    GetSignalName (const uint32 signalIdx, StreamString &signalName)

TypeDescriptor    GetSignalType (const uint32 signalIdx)

bool    GetSignalNumberOfElements (const uint32 signalIdx, uint32 &numberOfElements)
```

https://vcis-jenkins.f4e.europa.eu/job/MARTe2-docs-master/doxygen/classMARTe_1_1DataSourceI.html

List of existent components

https://vcis-gitlab.f4e.europa.eu/aneto/MARTe2-components

Component	Documentation
BaseLib2GAM	Encapsulate and execute GAMs from BaseLib2 in MARTe2
ConversionGAM	GAM which allows to convert between different signal types
ConstantGAM	Generate constant values that can be updated with messages.
FilterGAM	GAM which allows to implement FIR & IIR filter with float32 type
HistogramGAM	Compute histograms from the input signal values.
Interleaved2FlatGAM	Allows to translate an interleaved memory region into a flat memory area (and vice-versa)..
IOGAM	GAM which copies its inputs to its outputs. Allows to plug different DataSources (e.g. driver with a DDB).
MuxGAM	Multiplexer GAM that allows multiplex different signals.
PIDGAM	A generic PID with saturation and anti-windup.
SSMGAM	A generic State Space model with constant matrices and float64.
StatisticsGAM	GAM which provides average, standard deviation, minimum and maximum of its input signal over a moving time window.
TimeCorrectionGAM	GAM which allows to estimate the next time-stamp value in a continuous time stream.
TriggerOnChangeGAM	Triggers MARTe::Message events on the basis of commands received in the input signals.
WaveformGAM	GAM which provides average, standard deviation, minimum and maximum of its input signal over a moving time window.

List of existent components

https://vcis-gitlab.f4e.europa.eu/aneto/MARTe2-components

Component	Documentation
DAN	Allows to store signals in an ITER DAN database.
EPICSCAInput	Retrieve data from any number of PVs using the EPICS channel access client protocol.
EPICSCAOutput	Output data into any number of PVs using the EPICS channel access client protocol.
EPICSPVAInput	Retrieve data from any number of PVA records using the EPICS PVA client protocol.
EPICSPVAOutput	Output data into any number of PVA records using the EPICS PVA client protocol.
FileReader	Read signals from a file using different formats.
FileWriter	Write signals to a file using different formats.
LinuxTimer	Generic timing data source.
LinkDataSource	Read/write signals from/to a MemoryGate.
LoggerDataSource	Prints in the MARTe logger the current value of any signal.
MDSReader	Allows to read data from an MDSplus tree.
MDSWriter	Allows to write data into an MDSplus tree.
NI1588TimeStamp	Circular buffer time stamp acquisition using the NI-1588 PCI-Express board.

List of existent components

https://vcis-gitlab.f4e.europa.eu/aneto/MARTe2-components

NI6259ADC	Provides an input interface to the NI6259 board.
NI6259DAC	Provides an analogue output interface to the NI6259 board.
NI6259DIO	Provides a digital input/output interface to the NI6259 board.
NI6368ADC	Provides an input interface to the NI6368 board.
NI6368DAC	Provides an analogue output interface to the NI6368 board.
NI6368DIO	Provides a digital input/output interface to the NI6368 board.
NI9157CircularFifoReader	Circular buffer acquisition from an NI-9157 FIFO.
NI9157MxiDataSource	NI9157 MXI interface implementation.
RealTimeThreadAsyncBridge	Enables the asynchronous sharing of signals between multiple real-time threads.
RealTimeThreadSynchronisation	Enables the synchronisation of multiple real-time threads.
SDNSubscriber	Receive signals transported over the ITER SDN.
SDNPublisher	Publish signals transported over the ITER SDN.

List of existent components

https://vcis-gitlab.f4e.europa.eu/aneto/MARTe2-components

Component	Documentation
BaseLib2Wrapper	Load BaseLib2 objects into a BaseLib2 GlobalObjectDatabase.
EPICSCAClient	Trigger Messages as a response to an EPICS PV value change.
EPICSPVA	Library of EPICSPVA based components that allow to implement PVA record databases and PVA RPC interfaces to MARTe.
MemoryGate	Allows asynchronous communication between any MARTe components.
SysLogger	LoggerConsumerI which outputs the log messages to a syslog.

List of brokers

Name Description	
MemoryMapInputBroker	Copies the signals from a memory area declared in the DataSource.
MemoryMapInperpolatedInputBroker	Interpolates the signals from the DataSource.
MemoryMapMultiBufferBroker	Copy from/to a DataSourcel multi-buffer memory address. A different buffer is allocated for each signal.
MemoryMapMultiBufferInputBroker	Input version of the MemoryMapMultiBufferBroker.
MemoryMapMultiBufferOutputBroker	Output version of the MemoryMapMultiBufferBroker.
MemoryMapOutputBroker	Copies the signals to a memory area declared in the DataSource.
MemoryMapSynchronisedInputBroker	Calls the `Synchronise` method on the DataSource before copying the signals.
MemoryMapSynchronisedMultiBufferInputBroker	Synchronised input implementation of the MemoryMapMultiBufferBroker.
MemoryMapSynchronisedMultiBufferOutputBroker	Synchronised output implementation of the MemoryMapMultiBufferBroker.
MemoryMapSynchronisedOutputBroker	Calls the `Synchronise` method on the DataSource after copying the signals.
MemoryMapAsyncOutputBroker	Asynchronously (i.e. in the context of another, decoupled, thread) calls the `Synchronise` method on the DataSource after copying the signals into a circular buffer.
MemoryMapAsyncTriggerOutputBroker	Only stores data based on an event trigger (with pre and post windows). Asynchronously (i.e. in the context of another, decoupled, thread) calls the `Synchronise` method on the DataSource after copying the signals into a circular buffer.

Exercise

- Correct bugs in configuration for != types of rules
- Alias and Ranges
- GAM with parameters
- GAM with structured signals
- DataSource
- Synchronised RTThreads
- Async RTThreads
- StateMachine

Find the bug 1

Objective: verify and fix rules in a MARTe configuration file

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-1.cfg -s State1
```

- Modify Configurations/RTApp-2-1.cfg
- Fix the errors
- Run the application again

Success: application executes and console is regularly updated

Find the bug 2

FUSION FOR ENERGY

Objective: verify and fix rules in a MARTe configuration file

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-2.cfg -s State1
```

- Modify Configurations/RTApp-2-2.cfg
- Fix the errors
- Run the application again

Success: application executes and console is regularly updated

Alias and Ranges

Objective: learn how to use the Alias and Range parameters

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-3.cfg -s State1
```

- Note:
  - 10 elements are print into the output
  - The GAMDisplay uses the Alias to access the signal in the DDB1
- Modify Configurations/RTApp-2-3.cfg
- Use the Ranges parameter in GAMDisplay to only output the **first** and the **last** element of the array
- Use the Alias parameter to rename the signal Output (in the GAMDisplay) to OutputReference
- Run the application again

Success: application executes and console is regularly updated only with the first and last element renamed as OutputReference

Learn more:

- Did you notice that no Signals were set in the FileReader DataSource?

https://vcis-jenkins.f4e.europa.eu/job/MARTe2-Components-docs-master/doxygen/classMARTe_1_1FileReader.html#details

Modify existent GAM 2

Objective: modify existent GAM and read parameters

- Modify GAMs/FixedGAMExample1/FixedGAMExample1.cpp
- Read the **Offset** parameter from the configuration and add it to the input signal
- Compile

```
cd ~/Projects/MARTe2-demos-padova/
export MARTe2_DIR=~/Projects/MARTe2-dev
export MARTe2_Components_DIR=~/Projects/MARTe2-components/
make -f Makefile.x86-linux
```

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-4.cfg -s State1
```

Success: application executes and console is regularly updated with value multiplied by the gain + the configured Offset

Modify existent GAM 3

Objective: modify existent GAM and adapt to different I/O

- Modify GAMs/VariableGAMExample1/VariableGAMExample1.cpp
- Implement the Execute method so that it multiplies each output signal by the corresponding gain

```
cd ~/Projects/MARTe2-demos-padova/
export MARTe2_DIR=~/Projects/MARTe2-dev
export MARTe2_Components_DIR=~/Projects/MARTe2-components/
make -f Makefile.x86-linux
```

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-5.cfg -s State1
```

Success: application executes and console is regularly updated with all the output values multiplied by the corresponding gains

Structured signal

Objective: use structured types in a configuration file

- Modify Configurations/RTApp-2-6.cfg
- Add the Model signal as a structure to the list of signal outputs by the GAMDisplay
- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-6.cfg -s State1
```

Success: application executes and console is regularly updated with the value of the structure

Modify existent DataSource

Objective: modify existent DataSource

- Modify DataSources/SimpleUDPSender/SimpleUDPSender.cpp
- Implement the GetBrokerName and the Synchronise method
    - *Hint you need to call Write and Flush on the UDPSocket*
- Compile

```
cd ~/Projects/MARTe2-demos-padova/
export MARTe2_DIR=~/Projects/MARTe2-dev
export MARTe2_Components_DIR=~/Projects/MARTe2-components/
make -f Makefile.x86-linux
```

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-7.cfg -s State1
```

- On another console type

```
nc -luv 127.0.0.1 4444 | hexdump -v -e '/4 "%d\n"'
```

Success: application executes and console is regularly updated with timer value, on the second console the time is also correctly listed

Modify existent DataSource 2

Objective: send structured data using DataSource without recompiling

- Modify Configurations/RTApp-2-7.cfg
- Add the Model Signal to the UDPGAM

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-7.cfg -s State1
```

- On another console type

```
nc -luv 127.0.0.1 4444 | hexdump -v -e '3/4 "%d    " /1 "   %d    " /8 " %.4f
" /4 " %.4f    " /8 " %d\n"'
```

Success: application executes and console is regularly updated with timer value, on the second console the packet with the structure is correctly listed

Synchronising RealTime Threads

Objective: exchange data between threads (synchronously)

- Modify Configurations/RTApp-2-8.cfg
- Change **GAMT1T2Interface** and **GAMDisplayThread2** so that it runs at ¼ of the frequency

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-8.cfg -s State1
```

Success: application executes and console is regularly updated by both threads at the relevant frequencies

Learn more:

- Use the Ranges to *downsample* the data and only show 1 out of 4 points in the **GAMDisplayThread2**
- Add a third thread at an 1/8 of the frequency

Asynchronous data between RealTime Threads

Objective: exchange data between threads (asynchronously – get latest)

- Modify Configurations/RTApp-2-9.cfg
- Change **GAMTimerT2** so that it runs at ¼ of the frequency

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-9.cfg -s State1
```

Success: application executes and console is regularly updated by both threads at the relevant frequencies

StateMachine

Objective: use the StateMachine to centralize state management

- Modify Configurations/RTApp-2-10.cfg
- Change **+START** so that the application goes directly to STATE3 upon start
  - Note that **param1 = State1** must also be changed!

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-10.cfg -m StateMachine:START
```

Success: application executes and console is regularly updated with slowly changing sine wave

Learn more:

- Modify the StateMachine so that in STATE2 it can also go back to STATE1

StateMachine II

Objective: drive the state machine from an external component

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/Startup/
./Main.sh -l RealTimeLoader -f ../Configurations/RTApp-2-11.cfg -m StateMachine:START
```

- On another console type

```
echo -e "Destination=StateMachine\nFunction=GOTOSTATE2" | nc 127.0.0.1 24680
```

- Check that the state has changed
- On another console type

```
echo -e "Destination=StateMachine\nFunction=GOTOSTATE3" | nc 127.0.0.1 24680
```

- Check that the state has changed

Success: application executes and state is updated successfully when running nc

Learn more:

- Modify the StateMachine so that in STATE2 it can also go back to STATE1

Further examples

- Measure the performance of all the components and output to the console
- Measure the performance use the HistogramGAM to collect statistics
- Use the WaveformGAM the SSMGAM and the PIDGAM to implement a control loop

FUSION FOR ENERGY

Thank you for your attention

Follow us on:

www.f4e.europa.eu

www.twitter.com/fusionforenergy

www.youtube.com/fusionforenergy

www.linkedin.com/company/fusion-for-energy

www.flickr.com/photos/fusionforenergy