MARTe2

Introduction and Examples

Alessandro Tenaglia
alessandro.tenaglia@uniroma2.it

University of Rome "Tor Vergata"
Department of Civil Engineering and Computer Science Engineering
Intelligent Systems Lab

May 25, 2022

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Roadmap

Roadmap

Overview

Multi-threaded Application Real-Time executor

MARTe is a **C++ modular** and **multi-platform framework** for the development of **multi-threaded real-time control system applications**.

MARTe1

- **MARTe1** is the previous version of this framework;
- **MARTe1** was deployed in many fusion real-time control systems, i.e. JET tokamak;
- The use of **MARTe1** increased the number of supported environments and platforms;

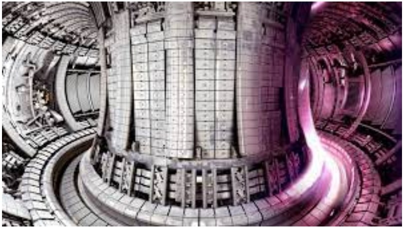

Figure 1: JET Tokamak

Roadmap

Code organisation

One of the main features of the MARTe2 architecture is the bold separation between:

- the platform **architecture**, i.e. x86, armv8;

Code organisation

One of the main features of the MARTe2 architecture is the bold separation between:

- the platform **architecture**, i.e. x86, armv8;
- the **environment** details, i.e. Linux, FreeRTOS, Windows;

Code organisation

One of the main features of the MARTe2 architecture is the bold separation between:

- the platform **architecture**, i.e. x86, armv8;
- the **environment** details, i.e. Linux, FreeRTOS, Windows;
- the **real-time algorithms**, i.e. the user code;

Code organisation

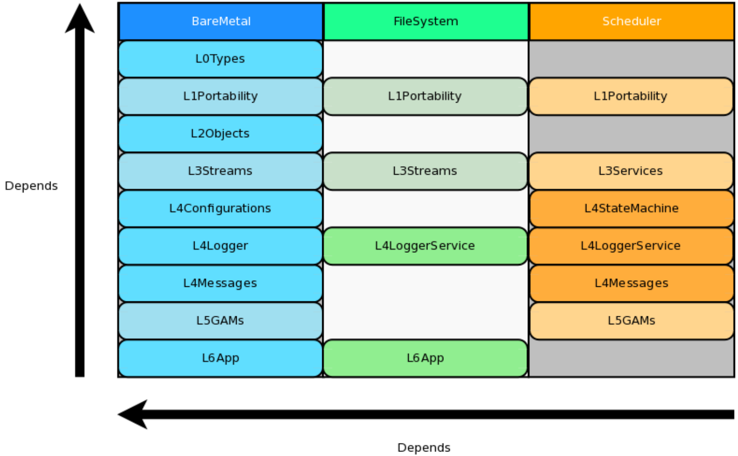

Figure 2: Code library

Makefile

The build of the core library (and all MARTe2 base projects) follows this structure:

- The **Makefile.os-arch** defines the **TARGET** operating system and architecture;

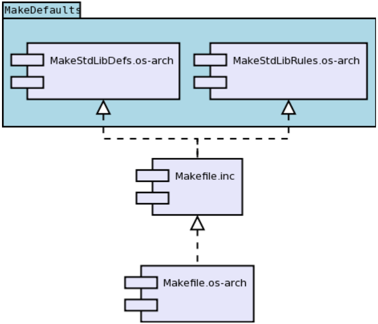

Figure 3: Makefile structure, from bottom to top

Makefile

The build of the core library (and all MARTe2 base projects) follows this structure:

- The **Makefile.os-arch** defines the **TARGET** operating system and architecture;
- The **Makefile.inc** defines all the common rules;

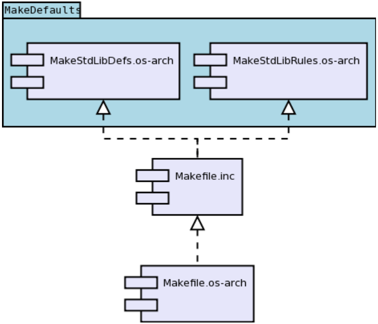

Figure 3: Makefile structure, from bottom to top

Makefile

The build of the core library (and all MARTe2 base projects) follows this structure:

- The **Makefile.os-arch** defines the **TARGET** operating system and architecture;
- The **Makefile.inc** defines all the common rules;
- The **MakeDefaults** defines the specific rules for the **TARGET**;

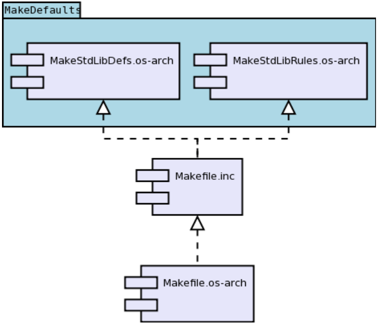

Figure 3: Makefile structure, from bottom to top

Roadmap

Real-Time Applications

- MARTe2 offers a generic base application, see **MARTeApp.cpp**;
- Real-Time Applications are built from the base one through a **Configuration Files** (.cfg);
- The **Configuration Files** defines the algorithms to be executed (**GAMs**) and the hardware or software involved (**Data Sources**).

Configuration file: RT App

```
$RTApp = {
  Class = RealTimeApplication
  +Functions = { // GAMs
    Class = ReferenceContainer
    ...
  }
  +Data = { // Data Sources
    Class = ReferenceContainer
    ...
  }
  +States = { // RT States
    Class = ReferenceContainer
    ...
  }
  +Scheduler = { // Scheduler
    ...
  }
}
```

GAMs

Generic Application Module

The **GAMs** are the components where user-algorithms are to be implemented.

GAMs

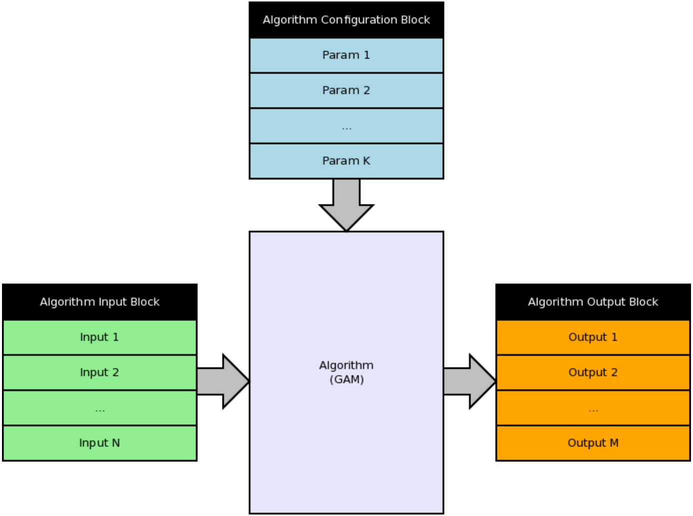

Figure 4: GAM

GAMs

Generic Application Module

The **GAMs** are the components where user-algorithms are to be implemented.

Warning

No interface with operating system (e.g. reading from files/sockets)

Configuration file: GAMs

```
+GAM1 = {
  Class = ExampleGAM
  InputSignals = {
    Input1 = {
      DataSource = DDB1
      Type = uint32
    }
  }
  OutputSignals = {
    Output1 = {
      DataSource = DDB1
      Type = uint32
    }
  }
  Parameters = {
    Param1 = (uint32) 1000
  }
}
```

Data Sources & Brokers

Data Sources

The **Data Sources** are the components that provide a interface for the interchange of input and output signals with the memory and the hardware.

Brokers

The **Brokers** are the components that provide the interface between the GAMs memory and the DataSource data.

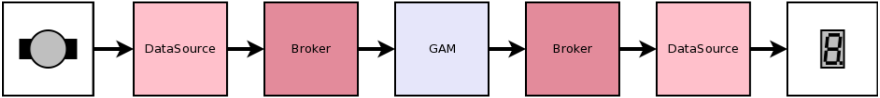

Figure 4: Data Sources & Brokers

Configuration file: Data Sources

```
+Data = { // Data Sources
  Class = ReferenceContainer
  +DDB1 = {
    Class = GAMDataSource
  }
  +Timer = {
    Class = LinuxTimer
    SleepNature = Default
    Signals = {
      Counter = {
      Type = uint32
    }
      Time = {
        Type = uint32
      }
    }
  ...
  }
}
```

States

- **GAMs** are grouped in real-time **threads** which are executed in the context of specific **states**.
- A Real-Time application shall be in one (**and only one**) state at a given time.

Configuration file: States

```
+States = { // RT States
  Class = ReferenceContainer
  +State1 = {
    Class = RealTimeState
    +Threads = {
      Class = ReferenceContainer
      +Thread1 = {
        Class = RealTimeThread
        CPUs = 0x8
        Functions = {GAMTimer}
      }
      ...
    }
  }
  ...
}
```

Configuration file: Scheduler

- A **real-time scheduler** handles thread execution;

```
+Scheduler = { // Scheduler
  Class = GAMScheduler
  TimingDataSource = Timings
}
```

Example: Real-Time Application

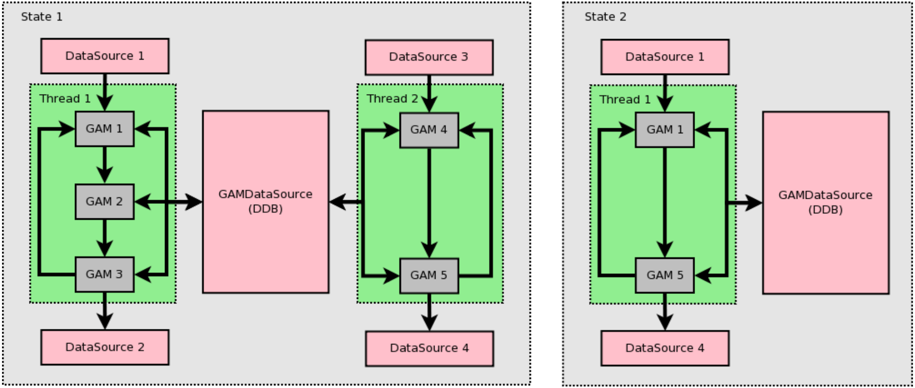

Figure 5: Example of a multi-state and multi-threaded real-time application

Roadmap

State Machine

State Machine

The **State Machine** is a component used to synchronise the application states against the external environment.

State Machine

State Machine

The **State Machine** is a component used to synchronise the application states against the external environment.

- The State Machine can be in **one and only one** state at a given time.
- The transitions between states are handled by **Events**;
- The State Machine components allows to associate the sending of **Messages** to **Events**;

State Machine

State Machine

The **State Machine** is a component used to synchronise the application states against the external environment.

- The State Machine can be in **one and only one** state at a given time.
- The transitions between states are handled by **Events**;
- The State Machine components allows to associate the sending of **Messages** to **Events**;

Warning

Be careful not to confuse the states of the **Real-Time application** with the states of the **State Machine**!

Example: State Machine

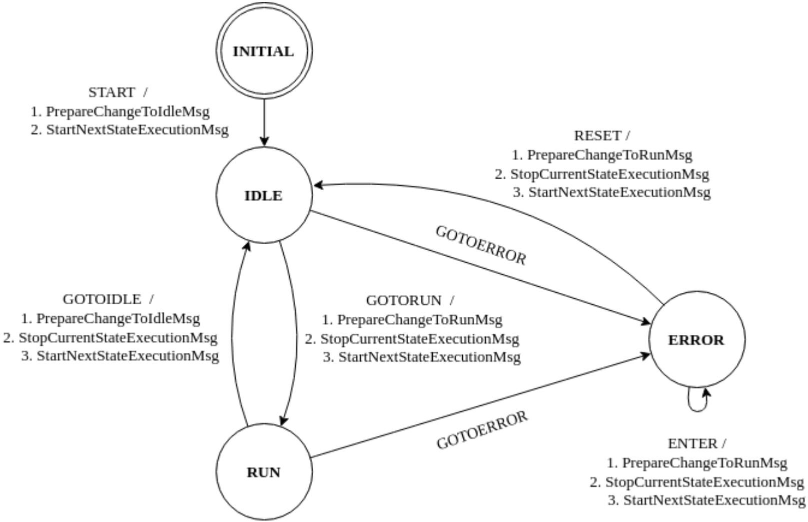

Figure 6: State Machine diagram

Configuration file: State Machine

```
+FSM = {
    Class = StateMachine
    +STATE1 = {
        Class = ReferenceContainer
        +GOTOSTATE2 = {
            Class = StateMachineEvent
            ...
        }
    }
    +STATE2 = {
        Class = ReferenceContainer
        +GOTOSTATE1 = {
            Class = StateMachineEvent
            ...
        }
    }
}
```

State Machine Event

A **StateMachineEvent** represents a transition and defines:

- **NextState**, the next state to pass through;
- **NextStateError**, the state to pass through on error;
- One or more **Messages** to send when is triggered:
  - PrepareNextState:
  - StopCurrentStateExecution:
  - StartNextStateExecution:

Configuration file: State Machine

```
+GOTOSTATE2 = {
    Class = StateMachineEvent
    NextState = STATE2
    NextStateError = ERROR
    +PrepareChangeToState2Msg = {
        Class = Message
        Function = PrepareNextState
    }
    +StopCurrentStateExecutionMsg = {
        Class = Message
        Function = StopCurrentStateExecution
    }
    +StartNextStateExecutionMsg = {
        Class = Message
        Function = StartNextStateExecution
    }
}
```

Roadmap

Model creation

- Create the model paying attention to data types;
- Define inputs as **Inport blocks** and name them;
- Define outputs as **Outport blocks** and name them;

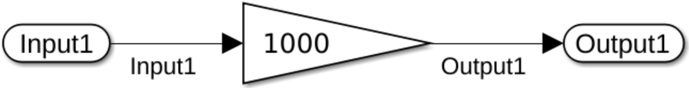

Figure 7: Base model

Param configuration

Paramters can be:

- **static**, so no longer modifiable after the code generation;
- **tunable**, so they can be modified at runtime;

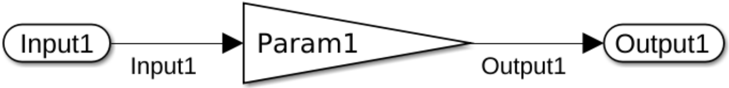

Figure 8: Base model with a tunable parameter

Code generation

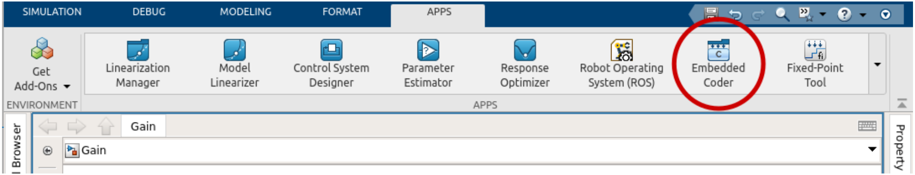

Figure 9: Embedded coder app

Code generation

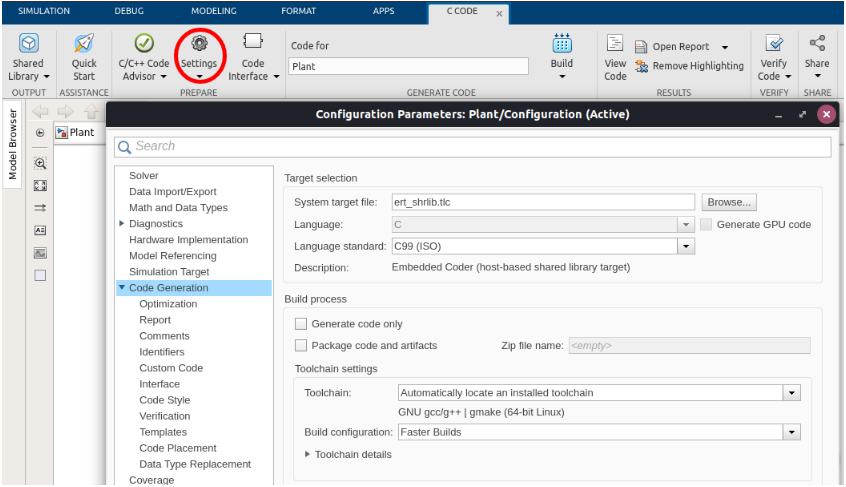

Figure 10: Code generation settings

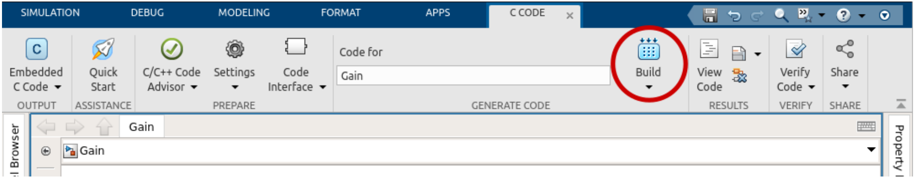

Figure 11: Base model with tunable parameter

Configuration file: SimulinkWrapperGAM

```
+GAMGain = {
    Class = SimulinkWrapperGAM
    Library = Gain.so // Library name
    SymbolPrefix = Gain // Model name
    InputSignals = {
        Input1 = {
            DataSource = DDB1
            Type = int32
        }
    }
    OutputSignals = {
        Output1 = {
            DataSource = DDB1
            Type = int32
        }
    }
    Parameters = {
        Param1 = (int32) 1000
    }
}
```

Example: Control System

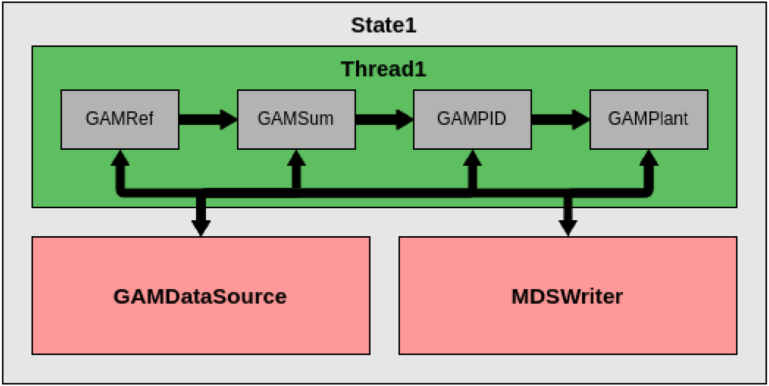

Figure 12: Control system app scheme

Roadmap

MDSplus

- **MDSplus** is a tool for data acquisition and storage;
- **MDSplus** stores data in a user-defined hierarchical structure, namely a tree;
- A tree is formed by nodes, each of which represents a data field;
- Experiments of the same type have the same tree structure and an incremental pulse number;

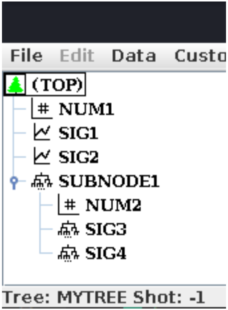

Figure 13: MDSplus tree