MARTe2

A real-time control framework for nuclear fusion

Roberto Masocco
roberto.masocco@uniroma2.it

University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering

June 12, 2024

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, *e.g.*, at the JET tokamak.

The use of **MARTe1** increased the number of supported environments and platforms.

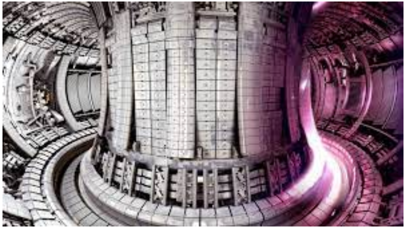

Figure 1: JET tokamak.

Roadmap

Code organisation

One of the main features of the MARTe2 architecture is the **decoupling** of:

- 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

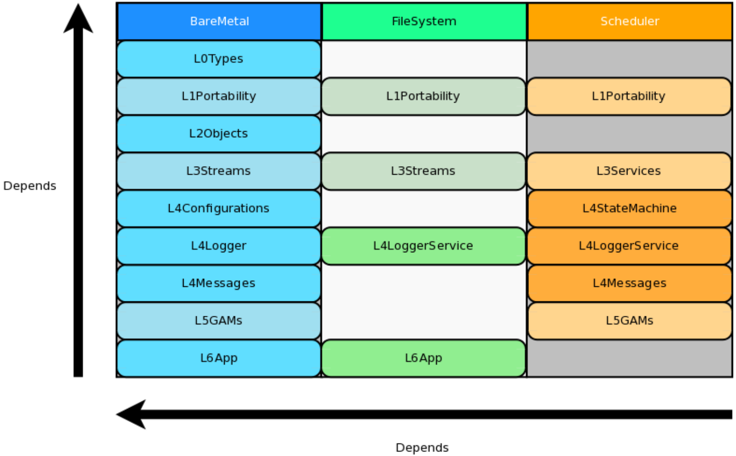

Figure 2: Core libraries organization.

Makefile

The build of the core library, as well as any MARTe2 project, follows this structure:

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

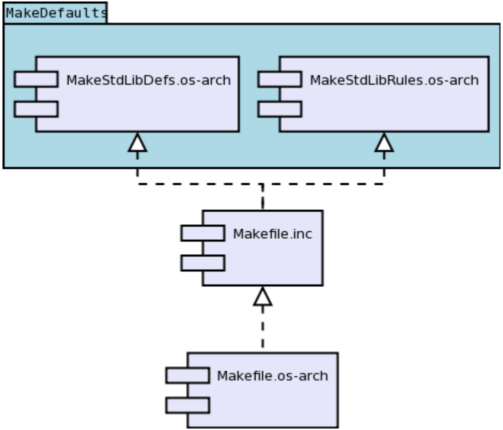

Figure 3: Makefile structure (bottom to top).

Roadmap

Real-Time Applications

MARTe2 offers a generic base application, see `MARTeApp.cpp`.

Real-Time Applications are built (more like "bootstrapped" or "put together") from the base one through **configuration files** (`.cfg`).

The **configuration files** define all the software components of a RTApp and their configuration properties, *e.g.*, the algorithms to be executed (**GAMs**) and the hardware or software modules involved (**Data Sources**).

Configuration file: RTApp

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

Listing 1: RTApp high-level configuration structure.

GAMs

Generic Application Module

The **GAMs** are the software components where **real-time user algorithms** must be implemented.

GAMs

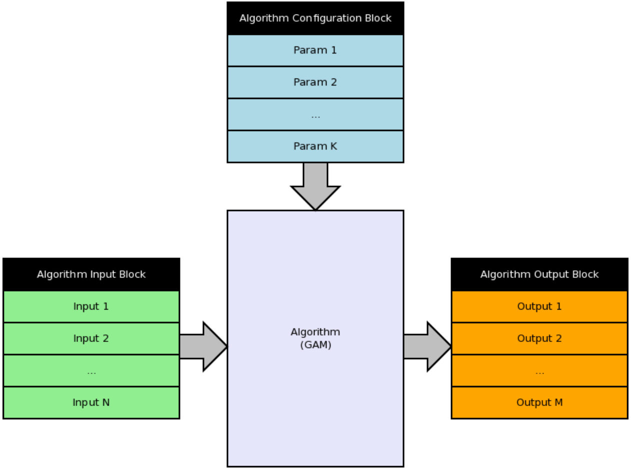

Figure 4: GAM operational scheme.

GAMs

Generic Application Module

The **GAMs** are the software components where **real-time user algorithms** must be implemented.

They should not perform neither data I/O operations nor OS calls (*e.g.*, accessing files, network sockets, devices...).

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

Listing 2: GAM configuration structure.

DataSources & Brokers

DataSources

The **DataSources** are the software components that provide an interface for the **exchange of input and output signals data** with the memory and the hardware.

Brokers

The **Brokers** are the software components that provide the **interface between the GAMs and the DataSources memory areas**, exchanging data between the two.

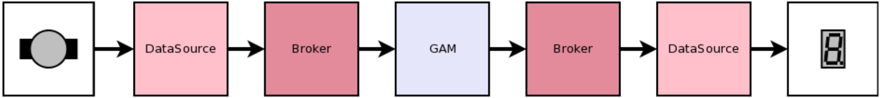

Figure 4: DataSources and Brokers operational scheme.

Configuration file: DataSource

```
+Data = { // Identifies DataSources section in the cfg
  Class = ReferenceContainer
  +DDB1 = {
    Class = GAMDataSource
  }
  +Timer = {
    Class = LinuxTimer
    SleepNature = Default
    Signals = {
      Counter = {
      Type = uint32
    }
      Time = {
        Type = uint32
      }
    }
  }
}
```

Listing 3: DataSource configuration structure.

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 = { // Identifies the States section
  Class = ReferenceContainer
  +State1 = { // For every state, multiple threads
    Class = RealTimeState
    +Threads = {
      Class = ReferenceContainer
      +Thread1 = {
        Class = RealTimeThread
        CPUs = 0x8 // CPU affinity
        Functions = {GAMTimer, ...} // Multiple GAMs
      }
      ...
    }
  }
  ...
}
```

Listing 4: States section of a configuration file.

Configuration file: Scheduler

A **real-time, OS-abstracting scheduler** handles the execution of real-time threads.

```
+Scheduler = { // Identifies the Scheduler section
  Class = GAMScheduler
  TimingDataSource = Timings
}
```

Listing 5: Scheduler section of a configuration file.

Example: Real-Time Application

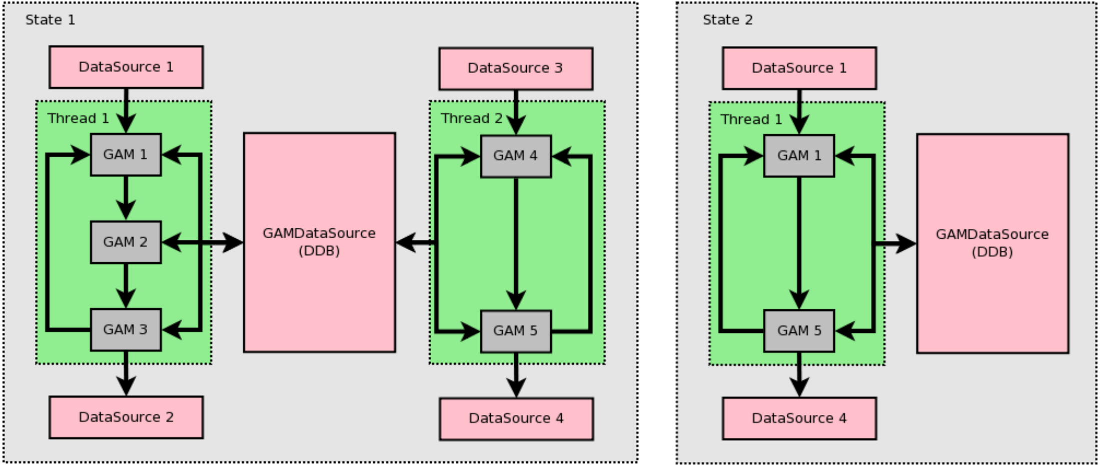

Figure 5: Example of a multi-state and multi-threaded Real-Time Application.

Roadmap

StateMachine

The **StateMachine** is a software component used to **synchronize the application states** with the external environment:

- it can be in **one and only one** state at a given time;
- transitions between states are handled by **Events**;
- it allows to associate the sending of **Messages** to **Events**, *i.e.*, to **trigger transitions externally**.

Warning

Be careful not to confuse the states of the **Real-Time Application** with the states of the **StateMachine**! The latter is just another software component you can use to implement FSMs.

Example: StateMachine

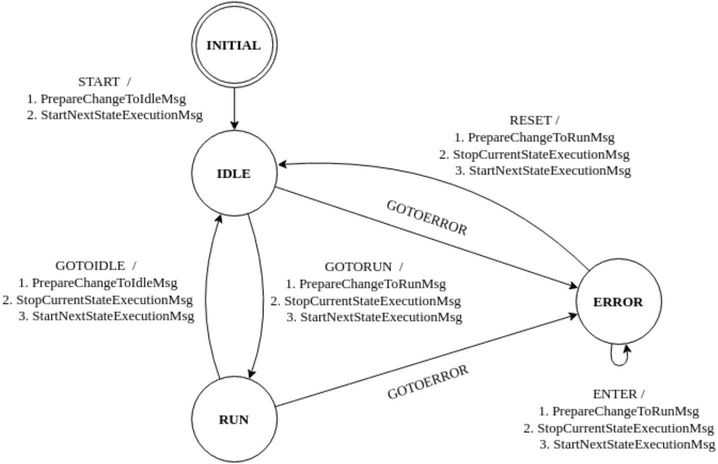

Figure 6: Example of StateMachine diagram.

Configuration file: StateMachine

```
+FSM = { // Declares a StateMachine named FSM
    Class = StateMachine
    +STATE1 = {
        Class = ReferenceContainer
        +GOTOSTATE2 = {
            Class = StateMachineEvent
            ...
        }}
    +STATE2 = {
        Class = ReferenceContainer
        +GOTOSTATE1 = {
            Class = StateMachineEvent
            ...
        }}}
```

Listing 6: StateMachine high-level configuration structure (brackets due to space constraints).

StateMachineEvent

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

- **NextState**, the next state to go to;
- **NextStateError**, the state to go to on error;
- one or more **Messages** to send when each of the following is executed:
  - `PrepareNextState`
  - `StopCurrentStateExecution`
  - `StartNextStateExecution`

Configuration file: StateMachineEvent

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

Listing 7: StateMachineEvent configuration structure (brackets due to space constraints).

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.

Compiled **shared libraries** can then be loaded by a **SimulinkWrapperGAM**.

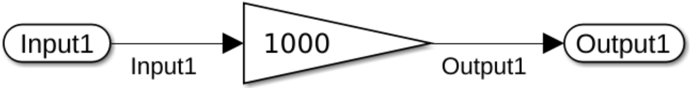

Figure 7: Basic Simulink model example.

Parameters

Paramters can be:

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

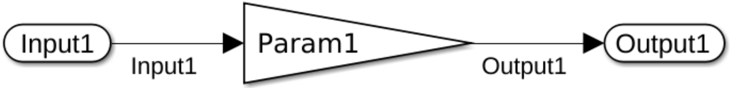

Figure 8: Basic Simulink model with a tunable parameter.

Code generation

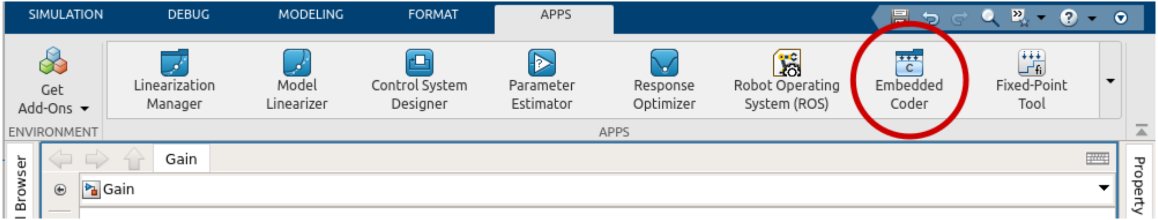

Figure 9: Embedded Coder app location.

Code generation

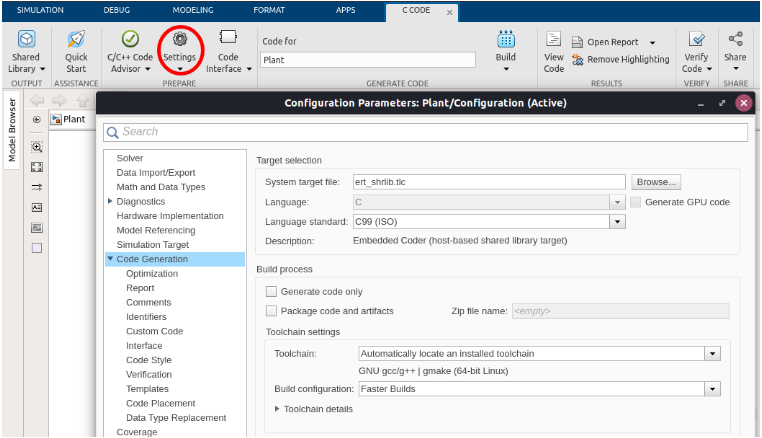

Figure 10: Code generation settings.

Code generation

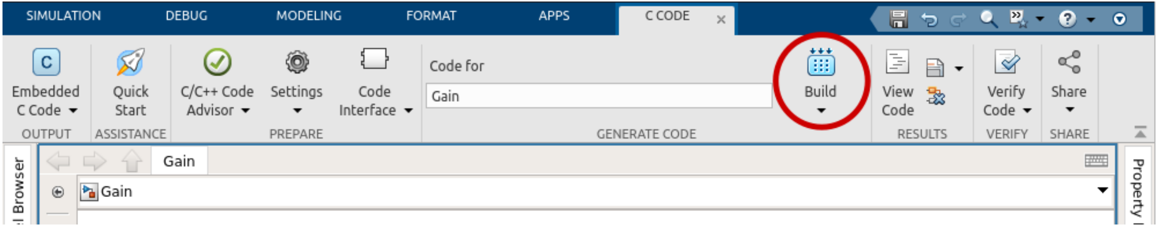

Figure 11: Build button location.

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}
}
```

Listing 8: SimulinkWrapperGAM configuration structure (brackets due to space constraints).

Example: Control system

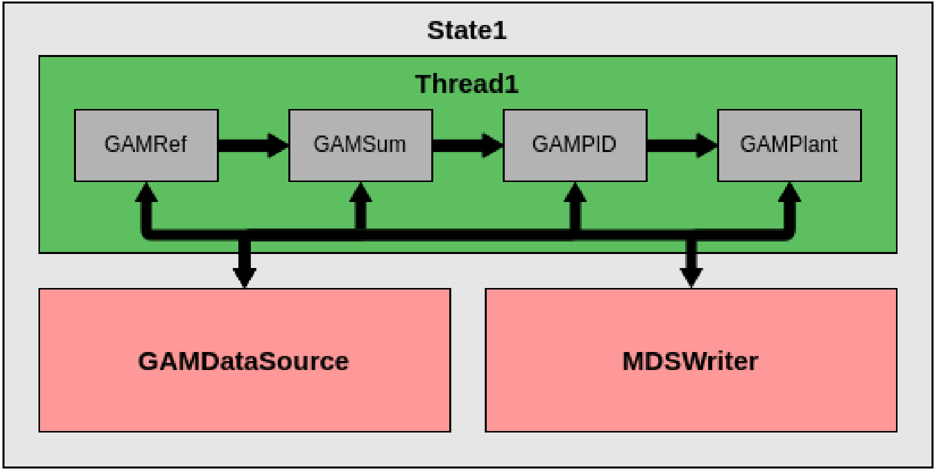

Figure 12: Control system MARTe app operational scheme.

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**.

Trees contents can be inspected with:

- **jScope** to plot signals;
- **jTraverser** to navigate the tree, inspect nodes and their values;
- **MDSReader**, **MDSWriter** DataSources.

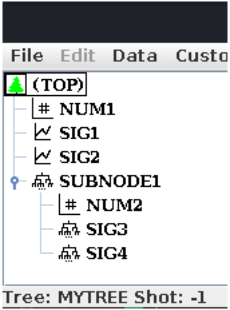

Figure 13: Example of MDSplus tree visualized with jTraverser.