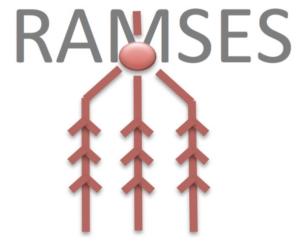
$\mathcal{R}AMSES$ 2 User Manual

Etienne Borde and Dominique Blouin

July 2020



Contents

1 Introduction **4**

2 Before starting **6**
2.1 Supported host platforms 6
2.2 Installation 6
2.3 Supported target platforms 6
2.4 Supported programming language 6

3 *RAMSES* 2, code synthesis process overview **7**

4 Configuration and launch **8**
4.1 *RAMSES* 2 preference page 8
4.2 *RAMSES* 2 property page 8
4.3 *RAMSES* 2 launch procedure 8

5 Illustrative examples **10**

6 User code integration **11**
6.1 User code: legacy integration 11
6.2 Writing AADL specific code 13

7 Input model requirements **16**
7.1 General requirements 16
7.2 Target specific requirements 17
7.2.1 Linux 17
7.2.2 nxtOSEK 18
7.2.3 POK 18
7.3 Requirements on local communications 18
7.4 Requirements on remote communications 19
7.4.1 Among processes bound to the same processor 19
7.4.2 Among processes bound to the different processors 19
7.4.3 Among processes and virtual busses 20
7.5 Requirements on modes specification 21

Add examples in RAMSES and pointers to these examples for each part of the documentation

1 Introduction

RAMSES 2 is a model refinement framework, based on the Architecture Analysis and Design Language (AADL), aiming the synthesis of embedded systems source code. *RAMSES* 2 is a plugin of the Open Source AADL Toolsuite Environment (OSATE). Figure 1 provides an overview of RAMSES functionalities: AADL model transformations, analysis, and code generation.

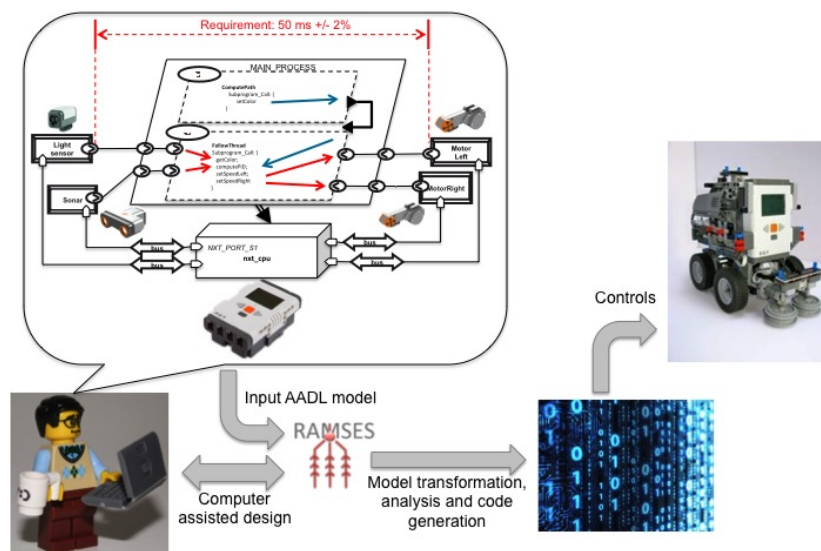


Figure 1: *RAMSES* 2, a Framework for Computer Assisted Software Design

This document is the user manual of *RAMSES* 2. To better understand this manual, readers are expected to have basic knowledge about:

- the C programming language;
- the AADL modeling language;
- the Eclipse integrated development environment and its OSATE perspective.

This user manual is structured as follows:

section 2. Installation procedure; list of supported target platforms;

section 3. Overview of the code synthesis process; introductions of terminology.

section 4. Configuration and launch procedure.

section 6 How to write source code in order to integrate it with the source code synthesized by $\mathcal{R}AMSES$ 2.

section 7 Specification of requirements for an input AADL model to be legit for $\mathcal{R}AMSES$ 2. Includes general requirements, to be met by any model, platform specific requirements, to be met by a model used to synthesize code for a target platform, etc.

2 Before starting

2.1 Supported host platforms

Being mostly implemented in Java, *RAMSES* 2 can run on any host platform on which a Java virtual machine (with a version number ≥ 1.8) is running.

Be aware that target platforms supported by *RAMSES* 2 might not be usable on every host. Refer to subsection 2.3 for the list of supported target platforms and a link to their documentation.

This user manual only describes how to configure the supported target platforms on Linux, which **we recommend to use as the host**.

2.2 Installation

Before installing *RAMSES* 2, you must install the **latest version** of OSATE. See the installation procedure of OSATE here.

The install procedure of *RAMSES* 2 is documented on this webpage.

2.3 Supported target platforms

RAMSES 2 supports three types of target platform:

1. POSIX compliant operating systems;
2. OSEK compliant operating systems;
3. ARINC653 compliant operating systems.

For each of these types, *RAMSES* 2 has been tested on a specific implementation:

1. Linux for POSIX-compliant operating systems;
2. nxtOSEK (`http://lejos-osek.sourceforge.net/`) for OSEK compliant operating systems;
3. POK (`https://pok-kernel.github.io/`) for ARINC653 compliant operating systems.

We describe how to install and configure nxtOSEK and POK on a Linux host on this webpage.

2.4 Supported programming language

RAMSES 2 can be used to integrate user code written using the C programming language. The way to integrate user code is described in section 6.

3 $\mathcal{R}AMSES$ 2, code synthesis process overview

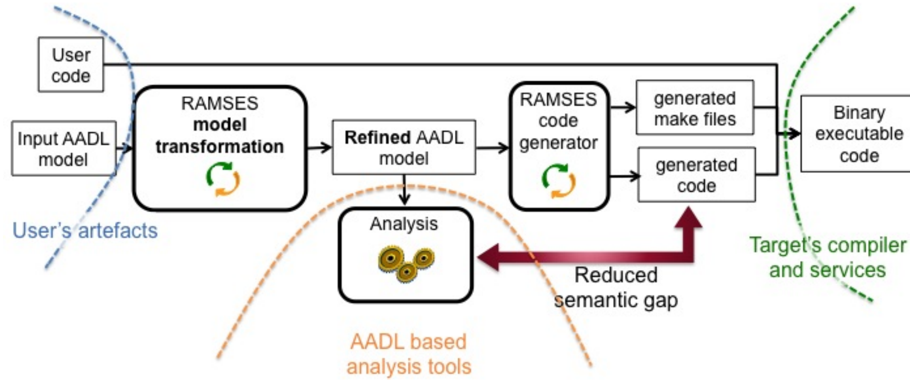


Figure 2: $\mathcal{R}AMSES$ 2, Model Transformation, Code Generation, and User Code Integration

Figure 2 illustrates the main components of $\mathcal{R}AMSES$ 2, as well as the artefacts they use and produce. On the left part of the figure, user's artefacts are listed: the input AADL model, and the associated user code. In section 6, we explain how to write the user code whereas the section 7 specifies what the input AADL model should contain. $\mathcal{R}AMSES$ 2 transforms the input AADL model into a (set of) refined AADL model(s), which can be analysed by AADL analysis tools. By producing and analysing a (set of) refined AADL model(s), the objective is to include the generated code, data, and associated runtime services in the analysis. This reduces the gap between the analysis results and generated code. Interested users can read our publication at RSP 2014 [Bor+14] for more information on this topic. The refined model is then used by RAMSES to generate source code and associated make files. The generated code is then compiled with the user code to produce the executable code.

4 Configuration and launch

4.1 $\mathcal{RAMSES}$ 2 preference page

The preference page of $\mathcal{RAMSES}$ 2 should be used to define the install directory of the targets you plan to use. To open the preference page of $\mathcal{RAMSES}$ 2, click on "Window → Preferences". The preference page of RAMSES is part of the listed preference pages on the left hand side. The preference page can also be opened from the property page, as explained in the next subsection.

4.2 $\mathcal{RAMSES}$ 2 property page

To open the configuration page of $\mathcal{RAMSES}$ 2, right click on your AADL project and select "Properties" (or select the project on select in the menu bar: "Project → Properties"). On the left side of the project property page, click on RAMSES.

The RAMSES property page appears on the right side. It contains a link to the preference page defined above. It also contains different fields to fill in:

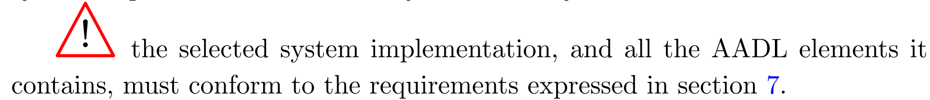
- output directory: this is the location where $\mathcal{RAMSES}$ 2 will store refined models, generated source code, build files, and binary code.
- target selection: chose the target patform you want to synthesize code for.
- target installation directory: if using Linux or All as a target, you can skip this. Otherwise, select the root directory of the target operating system (pok or nxtOSEK). For pok, this directory should contain the configuration file in `misc/config.mk`; for nxtOSEK, the root directory should contain the `ecrobot` subdirectory.
- properties table: when selecting a target, additional configuration properties may become available. If this is the case, the corresponding properties are listed in the property table, along with their description. Users can then define property values as strings.
- expand runtime share resources: check this box if you want to proceed to a timing analysis of the refined model and take into account the time spent in critical sections of the AADL runtime services.

The configuration is checked when saved and an error message appears if the configuration is incorrect.

4.3 $\mathcal{RAMSES}$ 2 launch procedure

$\mathcal{RAMSES}$ 2 can be launched either from the outline view of OSATE, or from an instance model. Both procedures are described bellow.

Launch from the outline view. By default, the outline view of OSATE appears on the right side. It represents the content of an AADL package as an unfold-able tree of AADL elements. In the outline view, select the AADL system implementation for which you want to synthesize code.


the selected system implementation, and all the AADL elements it contains, must conform to the requirements expressed in section 7.

Right click on the system implementation in the outline view, and click on “Launch RAMSES”. If the AADL project containing the selected system implementation is not configured properly, the project’s property page is automatically open. See section 4.2 for more information about the different fields of the property page.

Launch from an instance model. From the outline view, you can instantiate a system implementation: right-click on the system implementation of your choice and click on “Instantiate”. As a result, a directory named `instances` is created in your AADL project with an instance model in it (the file name of the instance model ends with an `aaxl2` extension).

Select the instance model file, then you can:

- select the “RAMSES 2” menu in the menu bar (top of the Eclipse window by default) and click on “Launch RAMSES”.

OR

- click on the RAMSES icon, present among the set of icons bellow the menu bar.

5 Illustrative examples

Once installed, *RAMSES* 2 comes with a set of illustrative examples. They are accessible by clicking on "File → New → Examples...". Then look for the RAMSES category. A step-by-step tutorial to execute one of these examples is available here. Provided examples are listed bellow.

For Linux:

- RAMSES Sampled Communications Example for Linux
- RAMSES Ping Sockets Example for Linux
- RAMSES Ping MQTT Example for Linux
- RAMSES Multi-mode threads Example for Linux
- RAMSES mine pump Example for Linux

For nxtOSEK:

- RAMSES Sampled Communications Example for nxtOSEK
- RAMSES Line Follower Example for NxtOSEK

For POK:

- RAMSES Sampled Communications Example for POK

For Ev3Dev:

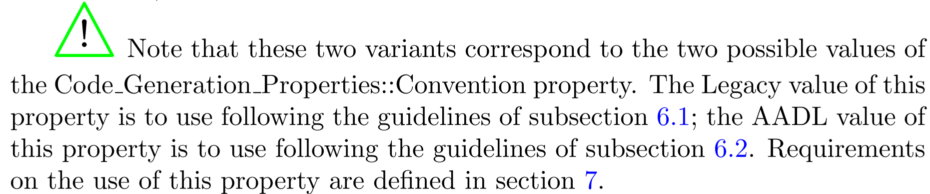
- RAMSES Sampled Communications Example for Ev3dev

For multi-targets:

- RAMSES multi-target Example

6 User code integration

In this section, we show how to write source code and AADL components in order to integrate user code in the code synthesized by $\mathcal{RAMSES}$ 2. The section is divided into two subsection: fisrt, how to integrate legacy code; second, how to write user code containing interations with AADL components features (ports in particular).

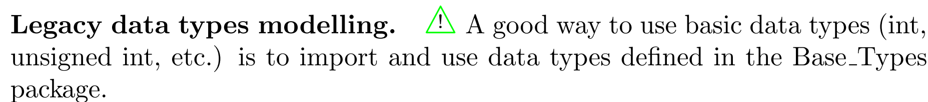


Note that these two variants correspond to the two possible values of the Code_Generation_Properties::Convention property. The Legacy value of this property is to use following the guidelines of subsection 6.1; the AADL value of this property is to use following the guidelines of subsection 6.2. Requirements on the use of this property are defined in section 7.

6.1 User code: legacy integration

In this subsection, we focus on the integration of legacy code. The motivation is to model existing code, written independently of the input AADL model, and to represent this code into an AADL model. This model can then be composed with other AADL models to define the AADL system implementation from which $\mathcal{RAMSES}$ 2 will generate code.

Modelling legacy code also requires to model legacy data types. The former relies on properties of the core AADL standard and the standard code generation annex; the latter relies on the standard data modeling annex.



Legacy data types modelling. A good way to use basic data types (int, unsigned int, etc.) is to import and use data types defined in the Base_Types package.

The set of useful properties for modeling data types are listed bellow. They are to be attached to AADL data components.

From the AADL core standard:

- Source_Text: list of files necessary to include when compiling the legacy code that uses the existing data type. The path provided for a file must either be an absolute path, a relative path from the AADL model, or a relative path from one of the included directory (see generated make files for more details on this last point). The parent directory of the file(s) with a .h extension is (are) added to the list of included directories.
- Source_Name (optional): name of the data type in the source files. If not provided, the name of the aadl data component (as defined in the code generation annex) is used.

From the standard data modeling annex, all the properties are relevant to model data types. We invite interested readers to read the data modeling annex for more detailed information.

```
1      data Int
2        properties
3          Data_Model::Data_Representation => integer;
4      end Int;
5
6      data Color
7        properties
8          Data_Model::Data_Representation => Enum;
9          Data_Model::Enumerators => ("NXT_COLOR_Black",
10                                     "NXT_COLOR_RED",
11                                     "NXT_COLOR_BLUE");
12         Source_Name => "COLOR_TYPE";
13         Source_Text => ("ecrobot_interface.h");
14     end Color;
15
```

Figure 3: AADL model of existing data types used in legacy source code

Figure 3 illustrate the use of these different properties to model an integer data type, and an enumeration. Both will be used in the example of legacy source code integration bellow. This model means that Int is an integer and Color is an enumeration defined as COLOR_TYPE in ecrobot_interface.h which is located in one of the directories of the nxtOSEK target platform.

Legacy source code modelling. The set of useful properties for legacy code modeling are listed bellow. They are to be attached to AADL subprograms modelling functions in the user code.

From the AADL core standard:

- Source_Language (optional): list of programming languages used to implement the function. Only supported value for $\mathcal{RAMSES}$ 2 is (C).
- Source_Text: list of files necessary to include/compile when integrating the legacy code into the synthesized code. The path provided for a file must either be an absolute path, a relative path from the AADL model, or a relative path from one of the included directory (see generated make files for more details on this last point). The extension of the file(s) is (are) used to decide if the source file(s) with a .c extension is (are) compiled or if the parent directory of the file(s) with a .h extension is (are) added to the list of included directories when compiling source files.
- Source_Name (optional): name of the function in the source code. If not provided, the name of the aadl subprogram (as defined in the code generation annex) is used.

From the standard code generation annex:

- Code_Generation_Properties::Return_Parameter; by default, the value of this property is false; set to true in case you need to represent a return

parameter of your function. It can only be associated to one of the parameters and this parameter must be an out parameter.

- Code_Generation_Properties::Parameter_Usage; this allows to specify if the data passed as parameter of the function is the data value or a reference to the data.

```
1    subprogram compute_pid
2      features
3        currentColor: in parameter Color;
4        colorToFollow: in parameter Color;
5        angle: out parameter Int;
6      properties
7        Source_Language => (C);
8        Source_Text => ("robot.h", "robot.c");
9        Source_Name => "computePID";
10     end compute_pid;
11
```

Figure 4: AADL model of an existing function in legacy source code

Figure 6 illustrates the use of these properties to model an existing C function called computePID in a "robot.c" source file located next to the AADL model. The signature of the function is :

```
1  void computePID(Color currentColor,
2                  Color colorToFollow,
3                  int * angle);
```

6.2 Writing AADL specific code

The main use-case for writing AADL specific code is to be able to interact with AADL components features in source code; for instance to manage the access to data queued in the input event data ports, or to interact with ports (*e.g.* send outputs) only under specific conditions detected at runtime. *RAMSES* 2 will apply the code integration method defined in the code generation annex if (i) the AADL value is associated to the Convention property, or if (ii) the subprogram has event or event data ports. Otherwise, the default semantics of AADL applies: read inputs, execute, write outputs.

Here is the list of signatures of functions (defined in `aadl_runtime_services.h`) user code can use:

- `error_code_t Put_Value(port_reference_t * port, void * value);`
- `error_code_t Send_Output(port_reference_t * port);`

- `error_code_t Next_Value(port_reference_t * port, void * dst);`
- `error_code_t Get_Value(port_reference_t * port, void * dst);`
- `error_code_t Receive_Input(port_reference_t * port);`
- `error_code_t Get_Count(port_reference_t * port, uint16_t * count_res);`
- `error_code_t Updated(port_reference_t * port, uint8_t * fresh_flag);`
- `error_code_t Await_Mode(thread_config_t * config);`
- `error_code_t Await_Dispatch(thread_config_t * global_q);`

Type `error_code_t` is defined as an enumeration, admiting the following values: `RUNTIME_OK, RUNTIME_EMPTY_QUEUE, RUNTIME_FULL_QUEUE, RUNTIME_LOCK_ERROR, RUNTIME_OUT_OF_BOUND, RUNTIME_INVALID_PARAMETER, RUNTIME_INVALID_SERVICE_CALL, RUNTIME_MISSING_INPUT, RUNTIME_INIT_HYPERPERIOD`.

We provide an example of AADL model in figure 5, which is rather simple; figure 6 provides source code with an example of interaction with a subprogram port. As you can see, because the AADL subprogram has an event port, the AADL convention mapping (defined in the code generation annex) is applied: the `C` function has a unique parameter, which is of type `__check_obstacle_context`. This context is actually a generated data structure, generated by $\mathcal{RAMSES}$ 2 in a file name `gtypes.h`. This data structure is composed of fields that matches the set of features of the AADL subprogram: `portId`, of type corresponding to `nxt_sensor_port` (using the same type mapping as for the legacy code) and `obstacle_detection` of type `port_reference_t`. In its code, a user can then use the runtime services defined in the C code of $\mathcal{RAMSES}$ 2. In this example, `__aadl_send_output` is used to send an event (the second parameter value can be ignored since we are just sending an event).

```
1    subprogram Check_Obstacle
2      features
3        portId: in parameter nxt_sensor_port;
4        obstacle_detected: out event port;
5      properties
6        Source_Language => (C);
7        Source_Text => ("source.h", "source.c");
8    end Check_Obstacle;
9
```

Figure 5: AADL model of a subprogram with an output event port

```c
#define MINDISTANCE 20
char obstacle_detected = 0;

void check_pbstacle(__check_obstacle_context * ctx) {
  uint32_t dist = ecrobot_get_sonar_sensor(ctx->portId);
  obstacle_detected = (dist>0 && dist <= MIN_DISTANCE);
  if(obstacle_detected) {
    // Put_Value on out event port
    Put_Value(ctx->obstacle_detected, NULL);
    // Send_Output
    Send_Output(ctx->obstacle_detected);
  }
}
```

Figure 6: User code with a runtime service call to send output in case of close obstacle

7 Input model requirements

In this section, we describe what an input AADL model must contain (or not) in order to be usable with RAMSES. Obviously, users should also make sure their models respect the naming, legality, and consistency rules as defined in the AADL standard.

7.1 General requirements

Root system instance

- From the root system, at least one process and one processor component should be defined, possibly traversing subcomponents of the root system. In other words, there must exist at least one process component instance and one processor component instance in the instance model.

Processor and Virtual Processor component instance

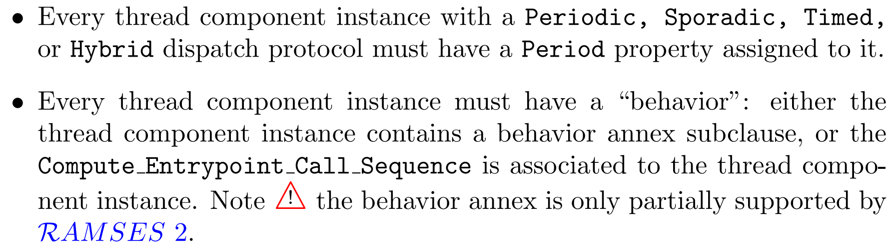
- The `RAMSES_Properties::Target` property must be associated to every Processor component instance. It is a string that identifies the target platform of the AADL processor ("Linux", "nxtOSEK", or "POK").
- The `Scheduling_Protocol` property must be associated to every Processor or Virtual Processor component instance.
- Every Virtual Processor component instance must be either (i) a subcomponent of a processor instance, or (ii) a subcomponent of a virtual processor component instance satisfying the current requirement, or (iii) bound (using the `Actual_Processor_Binding` property) to a processor component instance, or (iv) bound (using the `Actual_Processor_Binding` property) to a virtual processor component instance satisfying the current requirement.

Process component instance

- Every process component instance must be bound (using the `Actual_Processor_Binding` property) to exactly one processor or virtual processor component instance.
- Every process component instance must have at least one thread subcomponent, possibly traversing subcomponents (*thread groups*).

Thread component instance

- Every thread component instance must have a dispatch protocol: make sure the `Dispatch_Protocol` property is associated to every thread component instance.

- Every thread component instance with a `Periodic, Sporadic, Timed,` or `Hybrid` dispatch protocol must have a `Period` property assigned to it.
- Every thread component instance must have a "behavior": either the thread component instance contains a behavior annex subclause, or the `Compute_Entrypoint_Call_Sequence` is associated to the thread component instance. Note  the behavior annex is only partially supported by *RAMSES* 2.

Feature instances

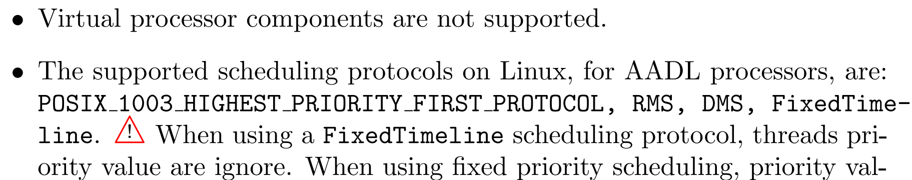
- Every feature instance must be part of a connection instance and/or used of at least one mode transition condition.
- Every feature instance of kind "feature group" must contain at least one feature instance.
- Every feature instance of kind "data port" or "event data port" must have a data classifier, and the data classifier must be attached a programming language data type (using the data modeling annex or the `Base_Types` package, see section 3 for more information on how to do this).

Subprogram components and calls

- Every subprogram referenced in a subprogram call must either have a behavior annex subclause, or a `Source_Text` property attached to it. See section 6 for more information about the meaning of this property.
- Every feature of the called subprogram must be connected to another feature.
- The default value for the Code_Generation_Properties::Convention property is Legacy. When the Legacy value is assigned to the subprogram, it can only have parameters (no port or data access). It cannot extend subprograms (or extend subprograms that extend subprograms...) having parameters because the ordering of parameters in the C code becomes more difficult to determine for users. If the AADL value is used, the `Parameter_Usage` value attached to parameters is ignored.

7.2 Target specific requirements

7.2.1 Linux

- Virtual processor components are not supported.
- The supported scheduling protocols on Linux, for AADL processors, are: `POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL, RMS, DMS, FixedTimeline`.  When using a `FixedTimeline` scheduling protocol, threads priority value are ignore. When using fixed priority scheduling, priority val-

ues are used as a reference. This means users must pre-compute priority values even when using a `RMS` or `DMS` scheduling.

7.2.2 nxtOSEK

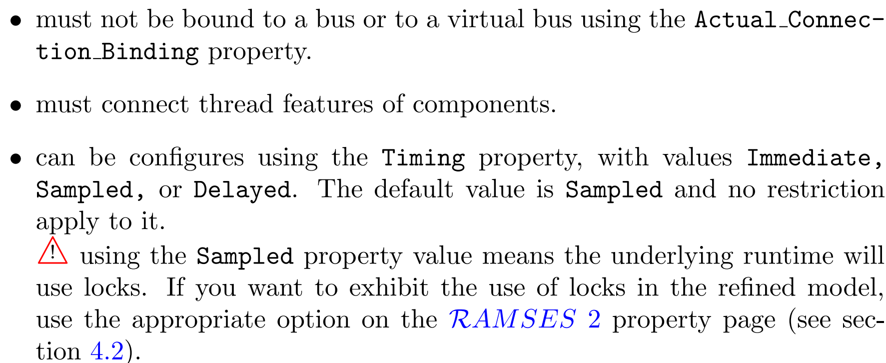
- Virtual processor components are not supported.
- The supported scheduling protocols on nxtOSEK, for AADL processors, are: `POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL, RMS,` and `DMS.`
- Every processor component instance must have the following properties : `OSEK::SystemCounter_MaxAllowedValue`, `OSEK::SystemCounter_TicksPerBase`, and `OSEK::SystemCounter_MinCycle`. These are properties for the configuration of the operating system; see the OSEK standard for the definition of these configuration parameters.

7.2.3 POK

- Virtual processor components are use to represent the module partitions environment (see the standard ARINC653 annex).
- The supported scheduling protocol on POK, for AADL processors, is: `ARINC653`.
- The supported scheduling protocols on POK, for AADL virtual processors, are: `RMS`, `POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL`, `Round_Robin_Protocol`, and `FixedTimeline`.
- Every processor component instance must have the following properties : `POK::Architecture`, `POK::BSP`, `ARINC653::Module_Major_Frame`, `ARINC653::Module_Schedule`. See the documentation of POK and the AADL ARINC653 annex for the definition of these configuration parameters.

7.3 Requirements on local communications

local communications:



- must not be bound to a bus or to a virtual bus using the `Actual_Connection_Binding` property.
- must connect thread features of components.
- can be configures using the `Timing` property, with values `Immediate, Sampled,` or `Delayed`. The default value is `Sampled` and no restriction apply to it.
  ⚠ using the `Sampled` property value means the underlying runtime will use locks. If you want to exhibit the use of locks in the refined model, use the appropriate option on the *RAMSES* 2 property page (see section 4.2).

⚠ When using the `Immediate` property value, you should make sure the scheduling configuration you use is compatible with this communication type: the thread with the output port of the connection should have a higher (or equal) frequency than the thread with the input port. The thread with the output port should also have a strictly higher priority (thus a smaller deadline or period in case of RMS and DMS respectively); except if you use a FixedTimeLine scheduling protocol (priorities are ignored in this case and the task set can be transformed into a simple directed acyclic graph of tasks with precedency constraints).
⚠Using the `Delayed` property value, a deadline can be provided for threads to compute the date of visibility of data on the output port(s). If not provided, *RAMSES* 2 considers implicit deadlines (deadline equals period).

7.4 Requirements on remote communications

remote communications may be connections (i) among processes bound to the same processor, (ii) among processes bound to different processors, or (iii) among processes and virtual busses.

7.4.1 Among processes bound to the same processor

This is only supported on POK, using the sampling and queueing services ot ARINC653; use the localhost and sockets to implement this communication type on Linux.

7.4.2 Among processes bound to the different processors

RAMSES 2 supports two types of communication through message queues. Message queue communications are modelled with a connection among ports of processes bound to different processors. Such a connection must:

- eventually connect features of thread components, or be involved in a mode transition in the destination process.
- connect ports for which the exchanged data size is known (using the `Data_Size` property or the data modeling annex).
- be bound to exactly one bus which connects the source and destination processors, or to a virtual bus bound to a bus that connects the source and target processor, or to a virtual bus bound to a virtual bus bound to a bus that connects the source and target processor, etc.
- use the default Timing property (Sampling).

The bus or virtual bus a remote connection is bound to must identify a protocol (e.g. sockets) and should be configured accordingly. Message queues

can be configured using sockets (tcp or udp) for the Linux target and pok-qemu-remote for the pok target.

The AADL property used to identify the communication protocol is `RAMSES_properties::Communication_Protocol`. It applies to bus or virtual bus components. It is an enum, and its potential values are `SOCKETS_TCP`, `SOCKETS_UDP`, and `POK_QEMU` for message queues.

When using Sockets (`SOCKET_TCP` or `SOCKET_UDP`) the bus accesses connected to the bus (or virtual bus) supporting the remote connection must have an address, and a port. The AADL property used to identify the address is `RAMSES_properties::Communication_Address`. The AADL property used to identify the port is `RAMSES_properties::Communication_Port`. Both are of type string.

Code generation for remote communications on POK-QEMU are only supported in a private plugin of RAMSES; contact the authors of this manual if you are interested.

7.4.3 Among processes and virtual busses

Publish subscribe communications are modelled with a connection among ports of processes and ports of a bus or virtual bus.

Such a connection must:

- on the process side, be connected to features of thread components, or be involved in a mode transition in the destination process.
- connect ports for which the exchanged data size is known (using the `Data_Size` property or the data modeling annex).
- exchanged data type should be associated with a topic identifier (the `RAMSES::Communication_Topic` property).
- use the default Timing property (Sampling).
- for a given virtual bus connected to a recipient process, there can only be one input port per topic. This is not really a restriction as the input port can then be connected to different threads of the recipient process. Given a recipient process, if the same topic is provided through N different MQTT networks, messages received on one port will be transmitted to the N input ports of the recipient process (they all subscribed to the same topic).

The virtual busses of such a remote connections must identify a protocol (e.g. MQTT) and should be configured accordingly. Publish/subscribe can be configured using MQTT for the Linux target.

The AADL property used to identify the communication protocol is `RAMSES_properties::Communication_Protocol`. It applies to bus or virtual bus components. It is an enum, and its values must be `MQTT`, for publish subscribe.

When using MQTT protocol, the connected bus should provide a host name and a port. If none is given, default values are used: localhost, 1883. Host name and port can be provided using the `RAMSES_properties::Communication_Host` and `RAMSES_properties::Communication_Port` properties respectively.

7.5 Requirements on modes specification

$\mathcal{RAMSES}$ 2 supports modes and mode switches to represent:

- activation/deactivation of thread components;
- mode-specific ports and data access connections;

subject to the following constraints:

- modes state machine can only be defined in process implementation components.
- the source of a mode change is an input event port connected to an output event port of a thread or to one of the predefined processor (or system representing a processor) ports (timing_failure_event, hyperperiod) as defined for the modeling of mixed-criticality directed acyclic graphs of tasks.

References

[Bor+14] E. Borde et al. "Architecture models refinement for fine grain timing analysis of embedded systems". In: *25th IEEE International Symposium on Rapid System Prototyping (RSP), 2014*. Oct. 2014, pp. 44–50.