ARTS Overview

Department of Informatics and Mathematical Modelling (IMM)
Denmark Technical University (DTU)

1 Contents

2 Introduction

This document tends to give an overview of the ARTS Framework as well as an introduction to the different main modules, used for constructing a framework. Low-level (source code) implementation details will not be presented. It is recommended to use the included examples as templates, when creating new frameworks.

2.1 Overview

The ARTS Framework is a simulation tool for user-driven abstract MPSoC design explorations. The framework allows for modelling of process elements (PE), consisting of an abstract application model (RTOS and tasks) and a core interface (IO device driver and IO device) for inter-processor communication. Further a SoC communication platform model is available for modelling different communication topologies, such as bus and NoC. Currently supported communication protocols are OCP 2.0 at TL1 and TL0. Figure 1 shows the main block diagram of the ARTS framework.

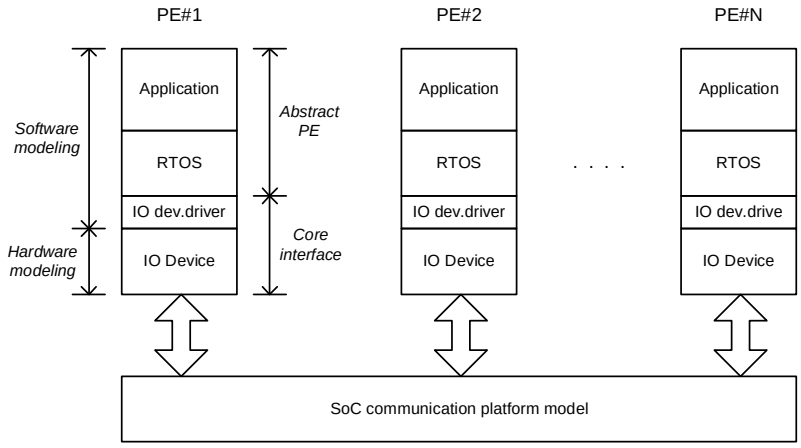

Figure 1 | ARTS framework block diagram.

The framework features flexible and easy configuration with respect to selection of task partitioning/mapping, RTOS protocols, communication topology etc. This is done using a simple script language, developed especially for the ARTS Framework. Further, the framework supports configuration, based on TGFF files (describing resources, application and data communication).

2.2 Implementation approach

2.2.1 Platform

The framework is based on SystemC 2.0.1 and has been implemented in an object oriented manner, making it easy to implement new modules (e.g. different RTOS policies, task types etc.). All module type implementations inherit from an associated base class, defining the API to the particular module.

2.2.2 Module communication

Communication between different modules (e.g. in a PE) is based on *messages*, where a message is a `struct`. Passing a message from one module to another is done using calls to a (API) method, defined by the base class of the target module. Argument to this method is the message. In

conjunction to this, the different modules are “connected” via object pointers. (i.e. if module A communicates to module B, a module B object pointer must be passed onto module A, before the simulation starts).

3 System requirements

In order to use the ARTS framework, the following requirements must be met:

- C++ compiler (GNU g++/gcc, Microsoft VC 6.0)
- SystemC 2.0.1 (available from www.systemc.org)
- OCP Transaction Level Library (can be requested from www.ocpip.org)

NOTE: The OCP Transaction Level Library for SystemC is only required, when using OCP TL1 as communication protocol. In addition to this, it is highly recommended to install the OCP Monitor package (only available for members of OCP), in order to monitoring the OCP channel.

4 Directory structure

The ARTS Framework directory structure is shown in Figure 2. The contents of the different folders are briefly described in Table 1.

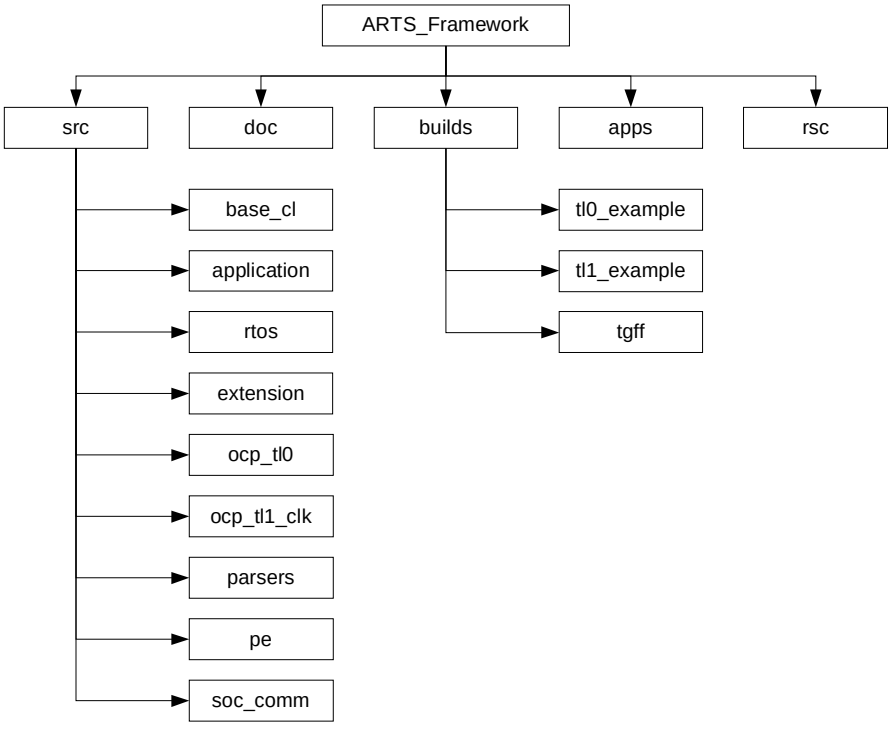

Figure 2 | ARTS Framework directory structure.

Directory	Contents description
ARTS_Framework	Top-level directory. Contains a README file and the Makefile.defs, used when building a framework
./src	Folders with *source code* for the different implementation modules.

./src/base_cl	Base classes, defining the API to the different implementation modules.
./src/application	Implementation of different application models, consisting of periodic task model and the IO device driver (IO task). Also contains an application module, used for managing the different tasks (not IO task) assigned to a PE.
./src/rtos	Implementation of different RTOS protocols for synchronization, resource allocation and scheduling.
./src/extension	Different top-level extension modules. This includes a macro class (arts_macro) containing handy methods; for example for framework configuration. Also contains a dependency controller (global task database module) and a performance monitor module, for monitoring different performance parameters.
./src/ocp_tl0	Implementation of an OCP2.0 TL0 (RTL) IO device model, consisting of a master and slave.
./src/ocp_tl1_clk	Implementation of an OCP2.0 TL1 IO device model, consisting of a master and slave.
./src/parsers	ARTS script language parser (Parser) and different TGFF parsers for application (scanAPP.cpp), resource (scanRSC) and data communication (scanCMM) files. These are essential for the dynamic framework configuration.
./src/pe	Different Process Element (PE) implementations; one with a OCP2.0 TL0 core interface (PE_TL0) and another with an OCP2.0 TL1 core interface (PE_TL1).
./src/soc_comm	Implementation modules for the SoC communication platform model.
./builds	Folders with different *framework* implementation examples.
./builds/tl0_example	A framework instantiating a user defined no. of OCP2.0 TL0 PE's, connected to a communication platform, with user defined topology (bus or NoC). Using an ARTS script, defining the applications, RTOS policies etc. ***** NOT WORKING *****
./builds/tl1_example	A framework instantiating a user defined no. of OCP2.0 TL1 PE's, connected to a communication platform, with user defined topology (bus or NoC). Using an ARTS script, defining the applications, RTOS policies etc.
./builds/tgff	A more complex framework instantiating a user defined no. of OCP2.0 TL1 PE's, connected to a communication platform, with user defined topology (bus or NoC). This framework uses TGFF files for defining applications and processor types and an ARTS script for defining RTOS policies and initial task partitioning/mapping. This example uses iteration-based simulations.
./app	Different application files, profiled using TGFF.
./rsc	Different resource and data communication files in TGFF format.
./doc	ARTS framework documentation

Table 1 | ARTS Framework directory description.

5 The base classes

The base classes found in ./src/base_cl defines the API to the different modules (pure virtual methods), used in the ARTS framework. Usage of a module API ensures a well defined interface between the different modules and further allows for module exchange at runtime (e.g. changing RTOS scheduling policy or task mapping).

There exist different base classes for the modules used in the Process Element (PE) model (described next) and the SoC communication platform model (described later). Implementation of a new module type requires inheritance of the associated base module. Examples of different module implementations can be found in ./src/rtos (for RTOS module implementations, used in the PE model) and ./src/soc_comm (for SoC communication platform modules implementations). Use these as a reference, when creating new module implementations. See also the separate document, API_base_classes.doc, describing the different base classes.

6 Application model

The application model is based on static dataflow/task graphs, where the exact functionality of a task is abstracted away and expressed using a set of timing constrains (execution time, deadline and offset. There is a periodic task model implementation (PerTask) available, which can be found in ./src/application. This model supports pre-emption.

6.1 Object constructor

The PerTask object constructor requires the following arguments, except otherwise specified:

Type	**Description**
sc_module_name	SystemC module name
Uint	Thread ID
Uint	Task ID
Uint	Application ID (The ID of the application, to which the task belongs to).
Uint	Execution period/frequency, expressed in no. of clock cycles.
Uint	Deadline, expressed in no. of clock cycles
Uint	Offset, expressed in no. of clock cycles (an offset time, relative to zero-time, when the task is released)
ofstream*	Ofstream pointer to PE logfile for logging to file, when the task misses deadline. Not a mandatory argument. If left out, no file will be created.
performance_monitor*	Performance monitor object pointer, if the performance monitor are to monitoring the PE. Not a mandatory argument.

6.2 Public method description

Below follows a brief description of the different public method in the PerTask module.

Name : command
Arguments : msg (message_type*)
Return value : None
Description : API method called from the RTOS, when sending commands to the task (e.g. start

execution, redemption/resume etc.)

Name : set_pe
Arguments : id (uint)
Return value : None
Description : Sets the PE ID, to which the task must be assigned.

Name : get_pe
Arguments : None
Return value : uint
Description : Returns the PE ID, to which the task is assigned.

Name : get_taskID
Arguments : None
Return value : uint
Description : Returns the ID of the task.

Name : get_appID
Arguments : None
Return value : Uint
Description : Returns the ID of the application, which the task belongs to.

Name : get_pincode
Arguments : None
Return value : uint
Description : Returns the pincode of task, which is an encoded id, containing the task ID and the application ID. Bit [0:N-1] = application ID and bit[32:N] = task ID. The value N is equal to the define-statement, _PINCODE_BIT_SPLIT declared in Parameter.h in ./src/rtos.

Name : get_task_name
Arguments : None
Return value : sc_module_name
Description : Returns the sc_module_name of the task.

Name : Initialize
Arguments : None
Return value : None
Description : Initializes the state machine of the task.

Name : set_execution_time
Arguments : BCET (uint), WCET (uint)
Return value : None
Description : Sets the best-case execution time (BCET) and worse-case execution time (WCET), expressed in no. of clock cycles.

Name : get_execution_time
Arguments : BCET (&uint), WCET (&uint)

Return value : See arguments
Description : Returns the best-case execution time (BCET) and worse-case execution time (WCET), expressed in no. of clock cycles.

Name : get_prg_memory
Arguments : Size (uint)
Return value : None
Description : Sets the program memory size characteristic of the task.

Name : update_tx_datamem
Arguments : Size (uint)
Return value : None
Description : Updates the amount of data memory to *reserve*, when task execution starts. Each time this method is called the data memory requirement will increase, corresponding to the value of the argument

Name : get_tx_datamem
Arguments : None
Return value : uint
Description : Returns the amount of data memory to *reserve* (when task execution starts).

Name : update_rx_datamem
Arguments : Size (uint)
Return value : None
Description : Updates the amount of data memory to *release*, when task execution completes. Each time this method is called the data memory requirement will increase, corresponding to the value of the argument

Name : get_rx_datamem
Arguments : None
Return value : uint
Description : Returns the amount of data memory to *release* (when task execution completes).

Name : push_soc_comm_nfo
Arguments : Target task ID (uint),
Target application ID (uint),
Base address of target PE (uint),
Upper address range of target PE (uint)
Data transfer size (uint)
Transfer type; e.g. write/read (uint)
Return value : uint
Description : Pushes information about inter-task dependency into a database in the task. This information relates to preceding dependency to a task assigned to different PE and will cause the task to initiate a SoC transaction (inter-processor communication) when execution completes. For multiple inter-dependencies, this method just has to be called several times.

Name : init_soc_comm_nfo
Arguments : none
Return value : None
Description : Clears the inter-task dependency database.

Name : new_resource_requirement
Arguments : Resource ID (uint),
Resource request time (uint),
Critical section length (uint)
Return value : None
Description : Assigns a *PE local* resource requirement to a task, where the resource request time identifies the time, relative to start of execution, when the task should request for a resource, while critical section length identifies the amount of time the resource is occupied. The times are expressed in no. of clock cycles. For multiple resource requirements, this method just has to be called several times.

7 The Process element (PE)

The PE models the behaviour of an IP core; for example a CPU. It is characterized by supporting change of RTOS policies as well as task mapping during runtime. The model is shown in Figure 3 and with the different module briefly described next. There exists a PE implementation having a OCP2.0 TL0 and TL1 core interface respectively. They can be found in ./src/pe.

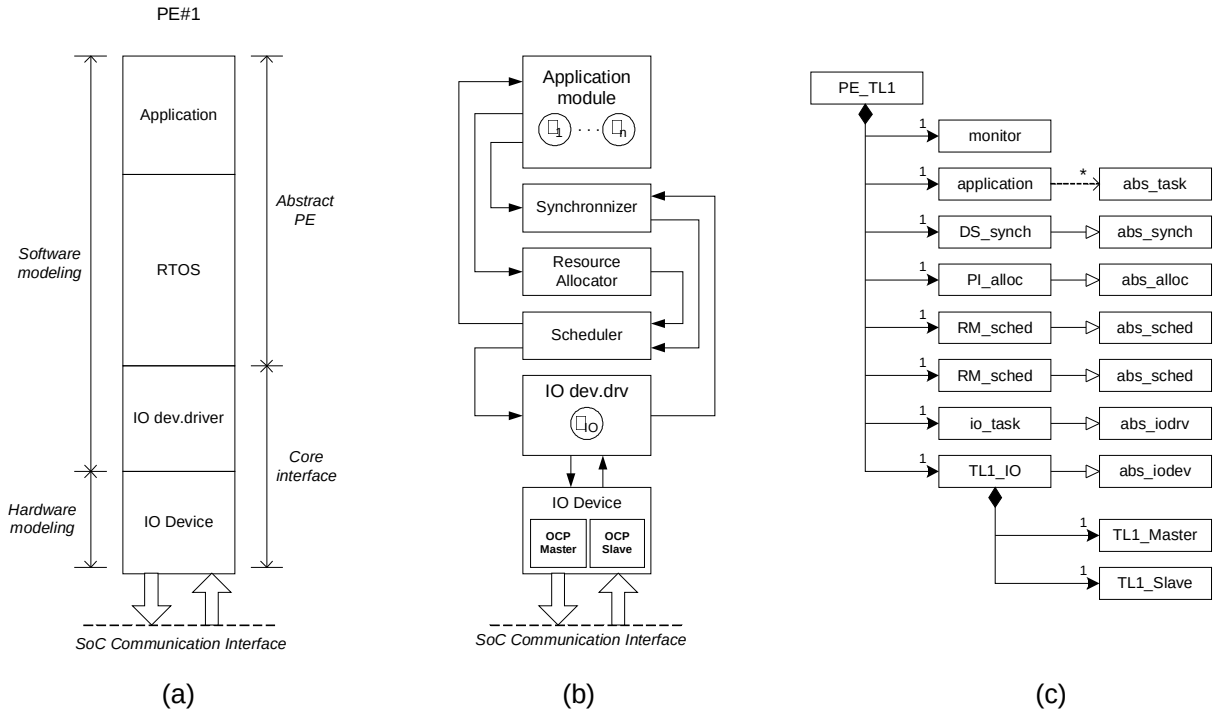

Figure 3 | (a) layer model (b) block diagram (c) simplified UML diagram (for PE_TL1).

7.1 Application

The application (module) holds pointers to task object assigned to the PE. It receives messages from the RTOS (scheduler) and forwards this to the target task. Further, it connects the assigned tasks to the RTOS, so they can send messages to the RTOS (synchronizer and resource allocator). The application module and the task module implementations are found in ./src/application.

7.2 RTOS

The Real-Time Operating System (RTOS) modes basic RTOS services, covering task synchronization, resource allocation and scheduling. It is composed of the synchronizer, resource allocator and scheduler modules. RTOS modules are found in ./src/rtos. Current supported protocols are listed in Table 2.

Module	Protocol
Synchronizer	Direct Synchronization (DS)
Resource Allocator	Basic Priority Inheritance (PI)
Scheduler	Rate-Monotonic (RM) Earliest-Deadline-First (EDF)

Table 2 | RTOS protocol implementations.

7.3 IO device driver

The IO device driver models an IO device driver application. It controls the IO device and encodes/decodes data to/from IO device (SoC communication interface), being synchronization messages between tasks with inter-dependencies.

For request (write or read transaction), the synchronization is based on the address encoding scheme, shown in Figure 4. For burst requests, the address encoding will be fixed. Tasks and application ID bit width can be configured using _TASK_ID_BW and _APP_ID_BW, specified in Parameter.h, located in ./src/rtos.

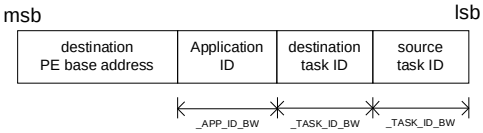

Figure 4 | Request (write|read) address decoding scheme.

For response (to a previous initiated read), the data will simply equals source task ID (issuing the response data).

The IO device driver module implementation is found in ./src/application.

7.4 IO device

The IO device models the physical hardware port, managing the communication protocol. Currently two IO device models are available for OCP 2.0 at TL0 and TL1 respectively. Both models have a fully multithreaded interface and can be configured, relative to the OCP channel. The PE implementation, PE_TL0 uses the TL0 model while PE_TL1 uses the TL1 model. The TL0 and TL1 IO device implementations can be found in ./src/ocp_tl0 and ./src/ocp_tl1_clk respectively.

7.5 Object constructor

The PE_TL0 and PE_TL1 object constructor requires the following arguments, except otherwise specified:

Type	Description
sc_module_name	SystemC module name
uint	PE ID
performance_monitor*	Performance monitor object pointer, if the performance monitor are to monitoring the PE. Not a mandatory argument.
dependency_control*	Dependency controller object pointer, required by the synchronizer module, in order to access the global synchronization database.
bool	Screen dump flag, enable RTOS status logging to screen (true=enable|false=disable)
ofstream*	Ofstream pointer to PE logfile for RTOS status logging to file. Not a mandatory argument. If left out, no file will be created.

7.6 Public method description

The following public method are common to PE_TL0 and PE_TL1, except otherwise specified.

Name : set_synchronizer
Arguments : type (uint)
Return value : None
Description : Selects the synchronizer to use. Not applicable at the moment, since only DS synchronization is implemented at the moment.

Name : set_allocator
Arguments : type (uint)
Return value : none
Description : Selects the resource allocator to use. Not applicable at the moment, since only Basic Priority Inheritance (PI) protocol is implemented at the moment.

Name : set_scheduler
Arguments : type (uint)
Return value : none
Description : Selects the scheduling policy to use. Applicable arguments (type) are:
0 = Rate Monotonic (RM) scheduling.
1 = Earliest Deadline First (EDF) scheduling.

Name : connect_OCP_Master
Arguments : *pOCP (OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >)
Return value : None
Description : Connects OCP Master in the PE_TL1 implementation to an OCP channel.
NOTE: For PE_TL0, see which SystemC signals are required/a part of the OCP channel in the header file (PE_TL0.h).

Name : connect_OCP_Slave

Arguments : *pOCP (OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >)
Return value : None
Description : Connects OCP Slave in the PE_TL1 implementation to an OCP channel. NOTE: For PE_TL0, see which SystemC signals are required/a part of the OCP channel in the header file (PE_TL0.h).

Name : set_master_buffer
Arguments : size (uint)
Return value : None
Description : Sets the response data buffer size in the OCP Master.

Name : set_slave_buffer
Arguments : size (uint)
Return value : None
Description : Sets the request data (write data) buffer size in the OCP Slave.

Name : set_processor
Arguments : type (uint)
Return value : None
Description : Sets the processor type ID for the PE. Not used for anything inside the PE.

Name : get_processor
Arguments : None
Return value : type (uint)
Description : Return the processor type ID for the PE. Use set_processor for specifying the processor type ID.

Name : set_address
Arguments : lo (uint), hi (uint)
Return value : None
Description : Assign an address space to the PE, used by other PE's when they are to transmit inter-dependency synchronization messages to this PE.

Name : get_address
Arguments : &lo (uint), &hi (uint)
Return value : Lower and upper address boundary; see Arguments.
Description : Returns the assigned address space to the PE.

Name : set_offset_time
Arguments : offset_time (sc_time)
Return value : None
Description : Set the offset time, when a simulation is restarted. Only applicable when doing iteration-based simulation (see the example in ./builds/tgff)

Name : map_tasks
Arguments : *obj (deque<abs_task*>)
Return value : None
Description : Used for assigning tasks to a PE. Argument is a pointer to a task pool. The

application module will scan the task module and connect all tasks, assigned to this PE.

Name	:	initialize
Arguments	:	None
Return value	:	None
Description	:	Initializes the PE. This consisting of disconnecting any assigned task and initializing the RTOS and IO device driver and IO device. NOTE: should only be used for iteration-based simulation (see the example in ./builds/tgff).

Name	:	flush_mode
Arguments	:	None
Return value	:	None
Description	:	Set the PE in flush mode; that is disconnecting any assigned task and initializing the RTOS. NOTE: flush mode MUST be used in iteration-based simulation (see the example in ./builds/tgff), BEFORE starting a new simulation.

Name	:	flush_done
Arguments	:	None
Return value	:	None
Description	:	Sets the PE out of flush mode. Must be called after flushing. NOTE: flush mode MUST be used in iteration-based simulation (see the example in ./builds/tgff), BEFORE starting a new simulation.

Name	:	get_task_count
Arguments	:	None
Return value	:	uint
Description	:	Returns the number of tasks assigned to the PE.

8 SoC communication platform model

The SoC communication platform model is used for modelling different communication topologies. Currently available topologies are a single shared bus and a simplified 1D/2D mesh Network-On-Chip (NoC) with minimal path routing and store-and-forward transmission approach. The SoC communication platform model is characterized by having an abstract description of the topology while being able to support transmission of real data very low abstraction level (e.g. at RTL). Figure 5 shows a block diagram of the model as well as the corresponding simplified UML diagram. There exists SoC communication platform models implementation, supporting OPC2.0 TL0 and TL1 protocol. They can be found in ./src/soc_comm.

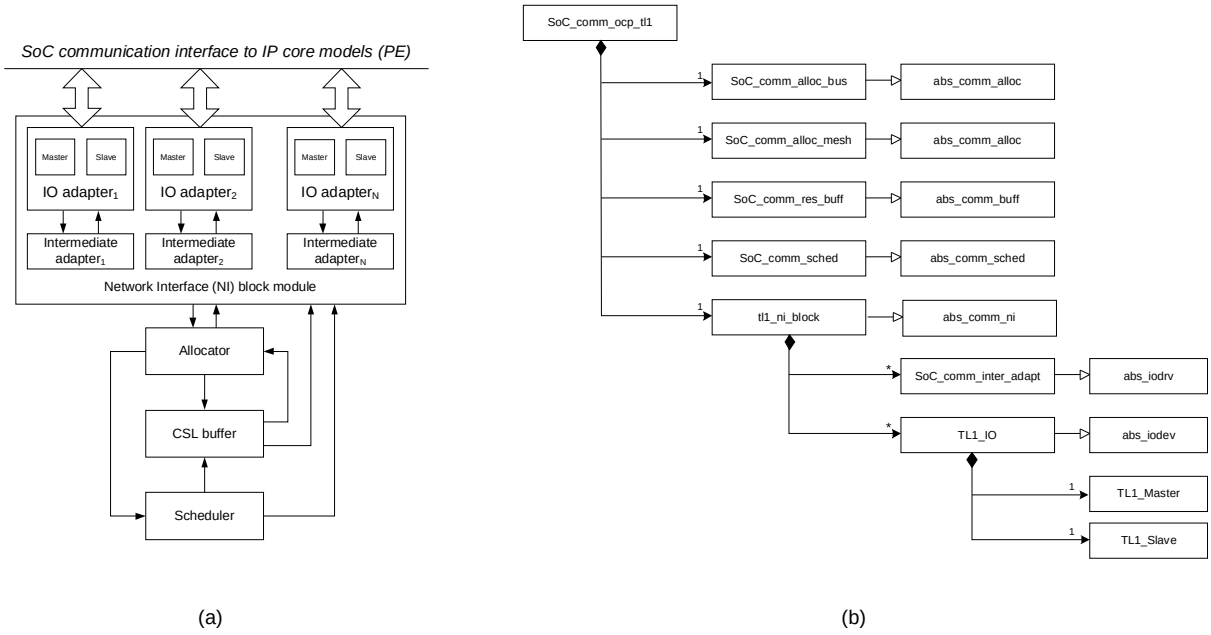

Figure 5 | (a) block diagram (b) simplified UML diagram.

8.1 Transport messages

The message communication in the SoC communication platform model is based on transport messages (noc_message_type) or data packages, containing a payload and a header, used for routing modelling. A transport message always originates from a network adapter when data is being received from an IP core model (PE). In the current implementation *only one* for transport message will be issued for a request/response transaction.

8.2 Network interface (NI) block

The NI block holds a configurable numbers of network adaptor models. It serves to route messages from the SoC communication layer (allocator, CSL buffer or Scheduler) to the correct network adapters.

A network adapter model is composed of an IO device model, handling the SoC communication protocol. This is the same module type used as IO device in the PE model. Further the network adapter model consists of an intermediate adapter, controlling the IO device and managing the encoding/decoding of data between IO device model and the SoC communication layer (allocator, CSL buffer and Scheduler). This module is somewhat equivalent to the functionality of IO device driver model in PE model, except that the behaviour is quit different.

There exists a NI block for OCP2.0 TL0 (tl0_ni_block) and TL1 (tl1_ni_block) respectively. They can be found in ./src/soc_comm.

8.3 Allocator

The allocator implements the actual topology modelling and manages allocation of shared communication resources. Transport messages received by the allocator always indicate release of a shared resource and requesting for a new one. Which new resource to assign to the transport message is determined by the allocator, and reflects the topology. If a resource is occupied, the transport message gets forwarded to the scheduler. Otherwise it is granted the resource, and the transport message gets forwarded to the CSL buffer.

There exists an allocator model for a single shared bus (SoC_comm_alloc_bus) and simplified 1D/2D mesh with minimal path routing (SoC_comm_alloc_mesh). They can be found in ./src/soc_comm.

8.4 CSL buffer

The CSL buffer models the mechanism of using a shared communication resource, by buffering a transport message during CSL. Relative to the data size, the allocator will have assigned a CSL to a transport message, equal to the amount of time the resource will be occupied. When CSL expires for transport message, it gets forwarded back to the allocator again. Thus the interaction between the allocator and CSL buffer actually models a chain of communication tasks (depending upon the topology modelling). The CSL buffer also manages the forwarding of a transport message, when it is ready for being released to the destination network adapter.

The implementation of the CSL buffer can be found in ./src/soc_comm.

8.5 Scheduler

The scheduler manages the scheduling of transport messages, in case of communication resource contention. The current scheduling policy is based on the first-come-first-served principle. When a resource becomes available, and there is a transport message waiting for this resource to become free, the scheduler will receive a message from the allocator. This causes the scheduler to release the transport message to the CSL buffer.

The implementation of the scheduler can be found in ./src/soc_comm.

8.6 Object constructor

The constructor for SoC communication platform models, supporting OCP2.0 TL0 and TL1 (SoC_comm_ocp20_tl0 and SoC_comm_ocp20_tl1) requires the following arguments, except otherwise specified:

Type	Description
sc_module_name	SystemC module name
uint	No. of PE's assigned to the framework
uint	No. of threads supported
bool	Screen dump flag. True=the state of the SoC communication model will be logged to screen, during simulation. False=no screen dumping.
ofstream*	Ofstream pointer to logfile, where to the state of the SoC communication model, during simulation. Not a mandatory argument. If left out, no file will be created.
ofstream*	Ofstream pointer to logfile, where to log communication contention count versus time. Not a mandatory argument. If left out, no file will be created.

8.7 Public method description

The following public method are common to SoC_comm_ocp20_tl0 and SoC_comm_ocp_tl1, except otherwise specified.

Name : set_addr_map
Arguments : nodeID (uint), addr_lo (uint), addr_hi (uint)
Return value : None
Description : Sets the address range (addr_lo to addr_hi) associated with a certain network adapter, identified by the node ID (nodeID). This information is forwarded to all network adapters in the NI block, and stored in a look-up table, used in conjunction with transport message routing management (that is identifying the target node ID for a request, for an example).

Name : get_refuse_count
Arguments : None
Return value : Uint
Description : Returns the number of contentions. The method is normally called after a simulation has completed.

Name : connect_OCP_Master
Arguments : *pOCP (OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >)
nodeID (uint)
Return value : None
Description : Connects OCP Master in network adapter (nodeID) to an OCP channel.
NOTE: Not implemented in SoC_comm_ocp20_tl0. See associated header file for the required SystemC signals, defining of the OCP channel.

Name : connect_OCP_Slave
Arguments : *pOCP (OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >)
nodeID (uint)
Return value : None
Description : Connects OCP Slave in network adapter (nodeID) to an OCP channel.
NOTE: Not implemented in SoC_comm_ocp20_tl0. See associated header file for the required SystemC signals, defining of the OCP channel.

Name : set_master_buffer_size
Arguments : size (uint)
Return value : None
Description : Sets the response data buffer size in the OCP Master. Common for all network adapters.

Name : set_slave_buffer_size
Arguments : size (uint)
Return value : None
Description : Sets the request data (write data) buffer size in the OCP Slave. Common for all network adapters.

Name : initialize
Arguments : None
Return value : None
Description : Initializes the SoC communication model.
NOTE: should only be used for iteration-based simulation (see the example in ./builds/tgff).

Name : flush_mode
Arguments : None
Return value : None
Description : Sets the SoC communication in flush mode; that is bypassing transport messages to/from the NI block to the SoC communication layer.
NOTE: flush mode MUST be used in iteration-based simulation (see the example in ./builds/tgff), BEFORE starting a new simulation.

Name : flush_done
Arguments : None
Return value : None
Description : Sets the PE out of flush mode; that is removing the bypassing of transport messages. Must be called after flushing.
NOTE: flush mode MUST be used in iteration-based simulation (see the example in ./builds/tgff), BEFORE starting a new simulation.

Name : set_offset_time
Arguments : offset_time (sc_time)
Return value : None
Description : Set the offset time, when a simulation is restarted. Only applicable when doing iteration-based simulation (see the example in ./builds/tgff)

Name : set_allocator
Arguments : type (uint) [span (uint)]
Return value : None
Description : Sets allocator type, defining the topology. Valid arguments are:
0 = bus model (SoC_comm_alloc_bus)

1 = 1D/2D mesh NoC (SoC_comm_alloc_mesh). 2nd argument defines the mesh-span, yielding a symmetrical mesh (i.e. span = 3 -> 3x3mesh).

Name : set_resource
Arguments : type (uint)
Return value : None
Description : Selects the CSL buffer type to use. Currently not applicable, since only one CSL buffer type is implemented.

Name : set_scheduler
Arguments : type (uint)
Return value : None
Description : Selects the scheduling policy. Currently not applicable, since only one CSL buffer type is implemented.

9 Top-level modules

There exists two other top-level modules (beside the parsers), connecting to the framework. These are the dependency controller and performance monitor. The implementations can be found in ./src/extensions.

9.1 Dependency controller

The dependency controller (dependency_controller) module is a global dependency database, managing the dependencies among the different (application) tasks assigned to the framework. Dependency database information (dm_type) is provided to the module constructor during object creation. A macro method (copy_db), found in the macros class (arts_macro located in ./src/extensions) has been implemented for fetching this information from the configuration file parser or TGFF application parser.

Public methods in the module are accessed by the synchronizers in the different PE, whenever information is provided to/from the database (e.g. when a task finished or checking when if dependencies have been resolved). In addition to this, a pointer to the dependency controller object must be provided to the PE_TL1/PE_TL0 constructor.

Beside this, the dependency controller holds a database with pointers to the different task objects. This database is used when blocking/unblocking a task (that is a task gets blocked after execution has completed, and unblocked again when the entire application has completed. This feature is implemented for synchronization reasons). Thus whenever a task object is created, a pointer to the object must be forwarded to the dependency control (done by calling push_task_ptr with object pointer as argument. See examples in ./builds).

9.1.1 Object constructor

The object constructor requires the following arguments, except otherwise specified:

Type	Description
dm_type*	Pointer to dependency data base, describing the dependencies among tasks in the different applications.

uint	No. of applications assigned to the framework
bool	Screen dump flag. True = a notification will be prompted to the screen, when an application completes, false = no screen dump.
ofstream*	Ofstream pointer to logfile, where to log application completion information. Not a mandatory argument. If left out, no file will be created.

9.2 Performance monitor

The performance monitor module is used for monitoring performance parameters of the different PE's, assigned to a framework. This includes parameters such as utilization and program/data memory usage. During simulation, public methods in the module are accessed by the RTOS and task modules for reporting different states. The performance monitor module is not mandatory and can be left out.

9.2.1 Object constructor

The object constructor requires the following arguments, except otherwise specified:

Type	**Description**
sc_module_name	SystemC module name
uint	No. of PE's assigned to the framework
uint	No. of applications assigned to the framework
ofstream*	Ofstream pointer to logfile, where to log PE utilization. Not a mandatory argument. If left out, no file will be created.
ofstream*	Ofstream pointer to logfile, where to log dynamic memory usage in the PE's (a file displaying the memory usage vs. time). Not a mandatory argument. If left out, no file will be created.

10 Framework construction steps

Different steps are required when constructing a framework. These steps can be seen in the top-level modules in the examples (./builds/tl1_example and/or ./builds/tgff). It is recommended to use the examples found in ./builds as templates when designing a new framework. However the *main* steps are briefly summarized in Table 3 to give an overview.

Step	**Description**
Parse files	Parse configuration file and TGFF files, if used for framework construction.
Extract configuration file scalar declarations	Get screen dump flags and filename declarations for configuration file and create ofstream objects for result logging.
Copy dependency database information	Extract the task dependency information from the configuration file or TGFF application parser and store this in a dependency database (dm_type). Use the macro, copy_db to do this.
Create dependency controller	Create the dependency controller object. A pointer to the newly created dependency database must be

	provided to the constructor.
Create performance monitor	Create performance monitor module, if performance monitor parameters (PE utilization/memory usage) are to be logged/monitored.
Create PE's and connect to OCP channels	Based upon the no. of PE assigned to the framework a corresponding no. of OCP channels must be created. PE characteristics are fetched from the configuration file parser by calling the macro method, get_pe_data.
Create SoC communication platform	Create SoC communication platform model, set topology, set address range for the different network adapters (equal to the PE address ranges) and connect to ocp channels.
Create tasks based on declarations (TGFF or configuration file)	Task information is fetched from the configuration file/TGFF application parser and used for task object creation. A pointer to the task object MUST be forwarded to the dependency controller (push_task_ptr) and store in a *task pool* (deque<abs_task*>), used in conjunction with PE task mapping.
Configure task (inter-dependency configuration)	Task having preceding inter-dependencies must be configured to issue an inter-processor communication, when execution completes. This is done by calling the macro method, task_configuration.
Map tasks to PE	After tasks have been created and configured, they are to be mapped to the different PE's. Mapping tasks to a PE is done by calling the PE method, map_task with a pointer to the task pool. Any tasks in the task pool, assigned to the PE will be connected to the PE.
Create and dump memory map	Create a task memory map for each PE, showing the address location, where non-local tasks write to/reads from, in case of inter-dependency. The memory map also shows the actual partitioning/task mapping. Memory map is created by calling the macro method, create_memory_map
Create VCD files	If VCD file dumping has been created, VCD file objects are created; one for task state vs. time and another for task execution on a PE vs. time.
Start execution	A this step, the simulation can be started since the framework has been constructed and connected.
Close result files	After simulation has completed, the different log files must be closed.
Delete objects (memory clean-up)	Finally, the different objects (i.e. PE's, tasks, SoC communication model etc.) must be deleted, to ensure memory clean-up.

Table 3 | Framework construction step overview.

11 Framework flushing and initialization

Iteration-based simulations required flushing and initializaion, when a series of simulation sessions are performed in a chain, like batch simulation (i.e. changing task partitioning/mapping and doing the same simulation again). An example of an iteration-bases simulation framework can be found in ./builds/tgff, where the processor type is changed from iteration to iteration. Flushing and initialization MUST be done when a simulation finished, but before a new one is started.

11.1 Flushing

Flushing is required to "flush out" any ongoing SoC transaction, occurring at the time when the simulation stopped. Without flushing, the SoC transaction will resume at the start of the next simulation. During flushing, the CI in the PE's and the network adapters in the SoC communication model will be disconnect and a number of simulation cycles are executed, to flush out any ongoing SoC transactions. Disconnection ensures that the RTOS in the PE and the network layer in the SoC communication model cannot issues new SoC transactions.

11.2 Initialization

After flushing the framework must be initialized. This includes resetting tasks, RTOS, the dependency database etc. to ensure any information/states related to the previous simulation is removed. It would be possible to avoid flushing, but this would introduce high-complexity in the initialization of the IO device drivers and IO device models.

The steps associated with flushing and initialization can be seen in the example in ./builds/tgff.

12 Framework examples

There exists different examples of framework implementations in ./builds. Please consult the README file in /ARTS_Framework for how to execute these examples.