Integration of heterogeneous components for co-simulation

Jawher Jerray, Rabéa Ameur-Boulifa, Ludovic Apvrille

Integration of heterogeneous components for co-simulation*

Jawher Jerray[1], Rabea Ameur-Boulifa[1], and Ludovic Apvrille[1]
[1]LTCI, Télécom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France

Abstract

Because of their complexity, embedded systems are designed with sub-systems or components taken in charge by different development teams or entities and with different modeling frameworks and simulation tools, depending on the characteristics of each component. Unfortunately, this diversity of tools and semantics makes the integration of these heterogeneous components difficult. Thus, to evaluate their integration before their hardware or software is available, one solution would be to merge them into a common modeling framework. Yet, such a holistic environment supporting many computation and computation semantics seems hard to settle. Another solution we investigate in this paper is to generically link their respective simulation environments in order to keep the strength and semantics of each component environment. The paper presents a method to simulate heterogeneous components of embedded systems in real-time. These components can be described at any abstraction level. Our main contribution is a generic glue that can analyze in real-time the state of different simulation environments and accordingly enforce the correct communication semantics between components. Once presented in a generic way, our glue is illustrated with Apache Kafka as the communication facility between simulation engines. It is then applied to two model and simulation frameworks: TTool and SystemC. Finally, Zigbee serves as a case study to illustrate the strengths of our approach.

***Keywords*—** Heterogeneous models, Simulation, Formal verification, Integration, System design

1 Introduction

Complex embedded systems are commonly designed using several modeling approaches and tools, because of the different nature of sub-systems, and because of the use of tiers to provide equipments. Integrating such heterogeneous is known as complex because of the diversity of models. Yet, ideally, this integration stage should be done as early as possible in the development process of these systems to verify, e.g., that the interfaces and main data exchanged are as expected, and can provide the expected overall functions.

Since forcing all suppliers to use the same modeling languages or simulation techniques is a too hard constraint, integrating together components designed in different formalisms requires either to somehow connect these formalisms. A first way to do would be to glue the different meta-models of the components in order to build a unique model from which verifications can be performed. This has already been shown in the scope of components for which their models of computation are quite similar [ZAM20]. Yet, when they are too different, a second approach must be used: connecting these components at simulation level. This is what the paper proposes to do, with a focus on the data exchanges between these components to ensure that the correct communication semantics are enforced.

In this paper, we define a method and techniques to allow to integrate a set of models designed with different frameworks and simulated using their own simulator, with no modification on their simulation

*This is the author version of the manuscript of the same name published in the proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023).

engine. To maintain the co-simulation in real-time of these different simulators, we propose a generic "simulation glue" based on a distributed event streaming platform to join heterogeneous simulators together. This work offers an alternative to the socket-based approaches such as the one proposed in [HEF19]. After having defined this glue, the paper illustrates in a more concrete way how SystemC and TTool [SAV21] components can be co-simulated, using Apache Kafka as the distributed event streaming platform. A case study with Zigbee shows the efficiency and limits of our contribution.

2 Related Work

In the area of heterogeneous distributed systems analysis, e.g., [BBS06; LD20], most of the works are based on co-simulation, but not many of them can support both simulation and formal validation in the same framework.

For coupling two or more simulation tools in a co-simulation environment, most of the approaches (e.g., [Nee+14; Tav+16; Mug+19]) rely on the Functional Mockup Interface (FMI) Standard [Blo+11] to bundle, in a single black-box, the internal computations and the interface descriptions of the simulation units. One of the most difficult challenges these approaches face is dealing with the gap between the different semantics [Tri15] of simulation units such as the various semantics of the coordination, which can refer to continuous time or discrete events. In [LD20], the authors have defined a language for describing model coordination interfaces. The interface is dedicated to share the elements necessary to coordinate the execution and communication among the simulation units.

There also exist dedicated platforms for the modelling and the simulation of heterogeneous component-based systems, examples include [Bal+03; Eke+03; BBS06]. These platforms support several modeling languages with a variety of component semantics. However, they offer a general and unified framework for the design and simulation, even for hybrid systems.

Among existing co-simulation solutions for integrating complex systems, the model transformation from a high-level language to SystemC is proposed in the scope of UML or SysML [RS07; VV09; RS12]. In [Ati+08], the authors propose a tool to transform a high level MARTE description into an executable platform via a chain of model transformations. These solutions are relatively tedious since the transformation of high-level languages into traditional simulation or verification must preserve the semantics, which is commonly tedious for supporting correctly communication and computation semantics. Moreover, all modeling environments must propose a model transformation to the same target language (e.g., SystemC) and the simulation specifications obtained from high-level languages must still be integrated together.

3 Contribution: a framework for simulating heterogeneous models via a distributed event streaming platform

This section presents our contribution: a new framework for the integration of heterogeneous components using a simulation integration platform. This integration platform is agnostic to the simulation technology, as long as it (partially) supports the communication semantics described in this section.

3.1 Component model

A model is components interconnected with ports. A model supports 4 types of ports: internal input ports I_i, external input ports I'_j, internal output ports O_k and external output ports O'_l. A model M has m internal input ports, n external input ports, p internal output ports and q external output ports as shown in Fig. 1. We assume that the objective is to co-simulate a set of components $\mathcal{C}$. We assume that the simulation engine of each $c \in \mathcal{C}$ can already handle communications internal to c. Thus, in this paper, we focus on external input ports I'_j and external output ports O'_l.

In this paper, we consider two modeling frameworks and their corresponding meta-models: TTool [23] and SystemC.

TTool models TTool can be used to capture SysML components that can be simulated with the internal simulator [KAP09] of TTool to prove safety and performance properties.

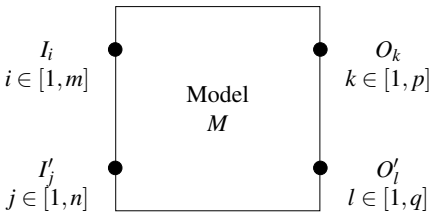

Figure 1: A model

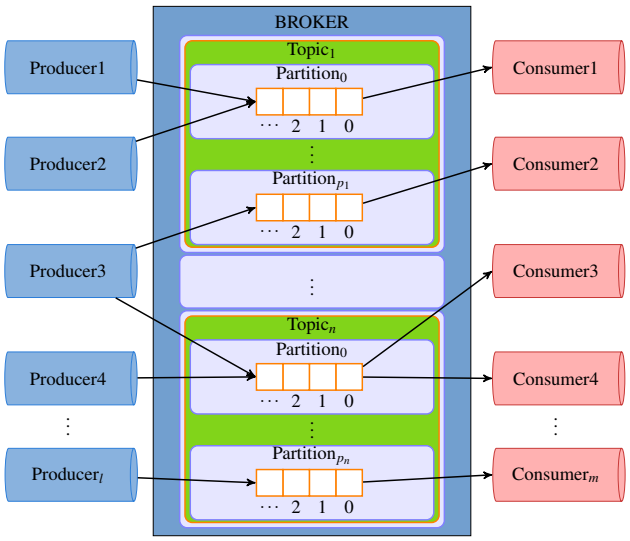

Figure 2: Structure of Kafka Server

SystemC models SystemC components, written in C++, can be simulated with the SystemC simulation engine.

3.2 Distributed event streaming platform

We assume that a facility, part of our platform, can stream events to distributed senders/receivers so as to ensure data can be sent and received, in real-time, by the simulation engines of components. There are many event streaming platforms, we cite for example: Apache Spark, Apache Kafka, Apache Flink and Spring Cloud Data Flow.

Apache Kafka. We decided to rely on the Apache Kafka to forward messages between simulation engines. We used Kafka because it supports many computing platforms and can also handle distributed and event-based communications. Yet, Kafka cannot natively support the communication semantics usually found in modeling frameworks for embedded systems, for instance the exchange of values via FIFO. Yet, Kafka supports the notion of *broker*: a broker contains a set of topics, and each topic has a set of partitions that can be considered as an infinite FIFO buffer, as shown in Fig. 2.

To send messages to a partition, we rely on *producers*. A producer can send messages to different partitions in different topics. *Consumers* can receive messages from partitions configured at co-simulation setup.

3.3 Communication

In Fig. 3, we give an example of the simulation integration between 3 models via a distributed event streaming platform, where:

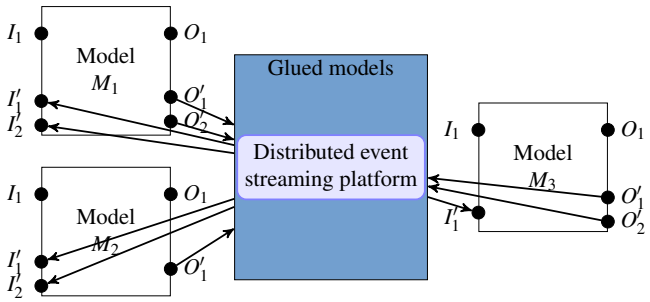

Figure 3: Communication between models and the distributed event streaming platform

- Model M_1 has 2 external input ports I'_1 and I'_2 and 2 external output ports O'_1 and O'_2.
- Model M_2 has 2 external input ports I'_1 and I'_2 and 1 external output port O'_1.
- Model M_3 has 1 external input port I'_1 and 2 external output ports O'_1 and O'_2.

The distributed event streaming platform must ensure communications from / to:

- External output O'_1 of M_1 to external input I'_1 of M_2.
- External output O'_2 of M_1 to external input I'_1 of M_3.
- External output O'_1 of M_2 to external input I'_1 of M_1.
- External output O'_1 of M_3 to external input I'_2 of M_1.
- External output O'_2 of M_3 to external input I'_2 of M_2.

Fig. 4 depicts the general approach of our contribution. First, we assume that a user wants to co-simulate at least two models having possibly different meta-models. From those models, our contribution automatically updates these models to allow them to interact with our co-simulation framework. Then, using these Co-simulation models, we start the corresponding simulation of each model at the same time. We assume we can access to the simulation trace of each simulation when they are running: what is of interest for us is obviously to identify all the potential read or write transactions on external ports. More precisely, the co-simulator needs to know about the kind of data being sent or received on each port and at which simulation step they are to be sent or received. When data is ready to be sent on an external output port, our co-simulator ensures data are forwarded to the corresponding external input port while enforcing the communication semantics, e.g., finite or infinite FIFO, exchange of values or exchange of a quantity of information.

Fig. 5 zooms on the co-simulation box shown in Fig. 4. As we can note all the co-simulation is based on streaming platform. For each model, the co-simulator manages the sending and receiving of data from the event streaming platform depending on the port type and the current status of ports.

3.4 Co-simulation models

The purpose of modifying the original models is to facilitate communication between the simulation of a model and the co-simulation. The Co-simulation model will add new tasks in the functional view, also it adds new CPUs and memories in the architecture. The Co-simulation models are automatically generated from a given model. We give below the algorithm that allows to obtain the "Co-simulation model" box shown in Fig. 4 for a given TTool model.

Algorithm. A TTool model consists of two views: a functional view and a platform view. In the functional view, SysML blocks are used to describe the functions and their communication. The behaviour of each function is given with an Activity Diagram. Functions can communicate together using two different facilities: event and data channels. Events can be used to exchange control values. Data channels are used to capture the exchange of a quantity of information. In the platform view, functional blocks are allocated to blocks representing hardware components: processors, buses and memories. Functions are to be allocated to execution components (e.g., processors), and data communication are mapped to buses

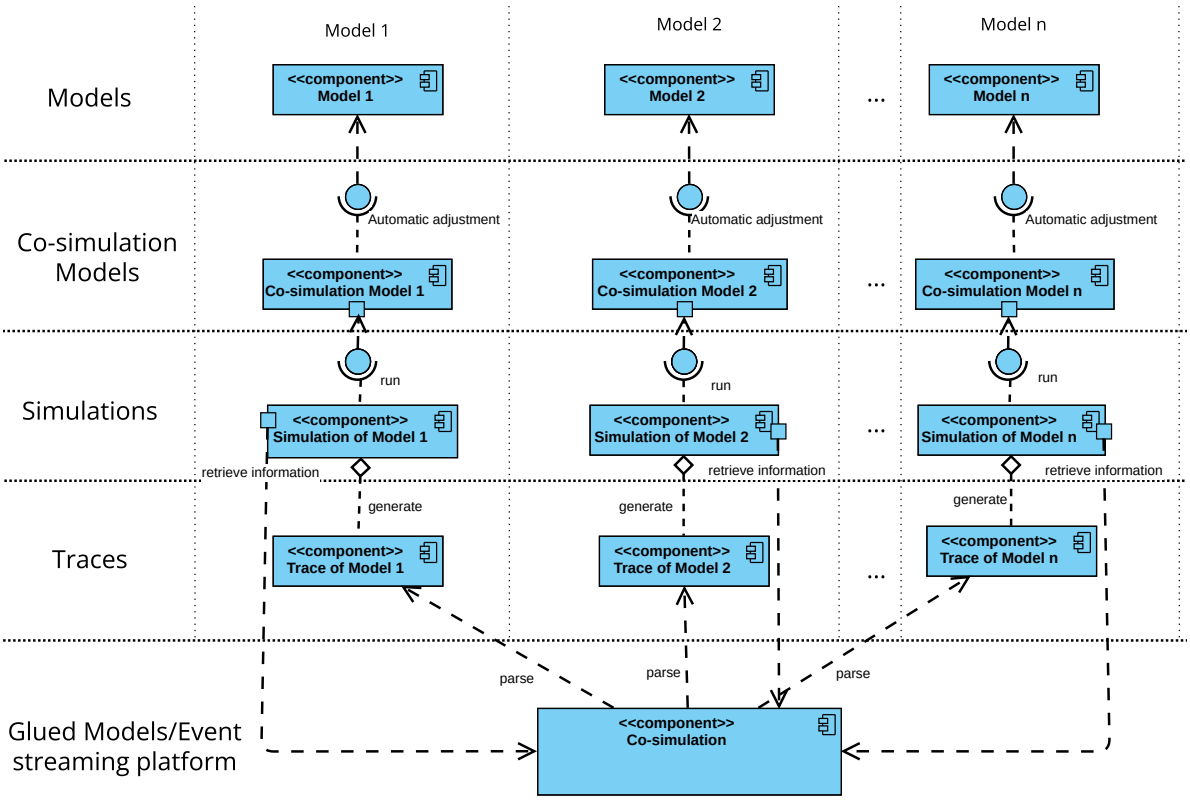

Figure 4: An overview of the proposed approach

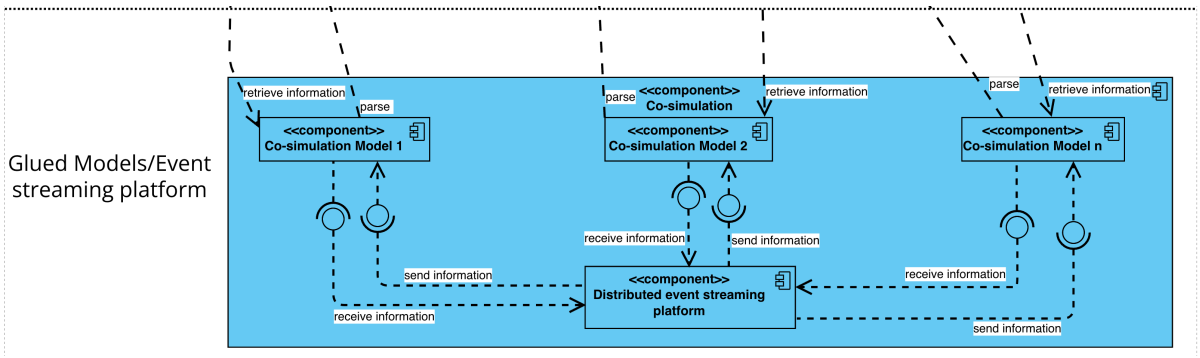

Figure 5: Zoom in the ”Co-simulation” box

and memories. Events are not mapped since the traffic induced by control signals is usually considered as negligible. Events and data channels support different semantics, such as finite FIFO, infinite FIFO, etc.

Handling a communication (events, data) with a model outside of TTool means that the hardware platform modeled in TTool must contain an input/output device connected to a bus from which all system components can be reached. This is how this is achieved in embedded systems: a communication interface must be used for input and output operations.

As a consequence, sending information to a component external to the TTool's model means that transactions on memories and on buses leading to the communication interface device must be taken into account when simulating the TTool model. Thus, when connecting TTool's simulator with another simulator, TTool's model must first be patched to add one communication device per output or input channel. Each communication component c_e is composed of a function f_{c_e} mapped to a dedicated processor p_{c_e}. Just like for other function block, f_{c_e} is captured with a SysML block. The activity diagram of f_{c_e} features a main loop, which is infinite. In the case of an input device, the loop contains a wait event operator that makes it possible to receive the number of samples to be read (i.e., the quantity of information). The wait event is followed by a read channel operator. In the case of an output device, the infinite loop had a send event to send to the destination the number of samples to be read, and then a write operator. Algorithm 1 formalizes the different steps necessary to prepare a TTool model for external co-simulation: this includes creating the new tasks f_{c_e} and their behavior, creating the processor p_{c_e}, connecting it to the corresponding bus, and connecting the new ports (ports of f_{c_e} to the sending ou receiving function).

4 Implementation of our method for sysML and SystemC models based on Kafka

4.1 An example of Generating automatically an Co-simulation TTool model

Let's consider an example with 3 external input ports ("evt_1", "comm_1" and "comm_2") and 2 external output ports ("evt_2" and "comm_3"). The top task diagram in Fig. 6 and the top architecture in Fig. 9 depict functional components and the allocation of components to the related architecture, respectively.

The application of Algorithm 1 to this model automatically transforms the component view to the bottom task diagram given in Fig. 6. Similarly, the activity diagrams of the new tasks of the external channels "comm_1" and "comm_3" are shown in Figs. 7 and 8, respectively. Besides, the activity diagram of the task of the external channel "comm_2" is similar to that of the channel "comm_1". Also, we give in the bottom of Fig. 9 the new hardware architecture.

4.2 Handling automatically the communication semantics via Kafka

The general idea behind our contribution is to create automatically the necessary partition to handle the communication semantics, e.g., to manage the number of events or data samples in transit between two external ports. At each port, i.e. for each model, we also create *consumers* and *producers*. A consumer intends to collect values from partitions, while a producer add information to partitions. Finally, our co-simulation framework is based on a set of partitions handled by Kafka and a set of producers and consumers.

We now review how the different communication semantics of data channels and events can be handled by the co-simulation framework ("Co-simulation" box in Fig. 4). The paper focuses on two communication semantics: Blocking Read Blocking Write for data channels and Block Read Non Blocking Write with finite FIFO for events.

4.2.1 Blocking Read, Blocking Write Channel

Let's consider a model built upon two components c_1 and c_2 exchanging data with an external communication. c_1 has an "external output" port *comm* of type "blocking write channel", c_2 has an "external input" port *comm* of type "blocking read channel". Because they are external, these two ports must exchange data via our platform to provide a blocking-read blocking-write channel communication. The platform is configured as follows.

Algorithm 1: Algorithm of the creation of the Co-simulation TTool model

```
for each i_input_channel in list_input_channels do
    disconnect_port_from_origin_side (i_input_channel); /* Disconnect the
      external port from its origin.                                    */
    create_new_task(); /* Create new task for the external input channel.
      */
    connect_port_to_new_task(i_input_channel); /* Connect the external port to
      the new task.                                                     */
    add_event(i_input_channel); /* Add an event to the new task, this event
      will retrieve data from the co-simulator using the avs command
      that allows to add signals to a given event.                      */
    associate_activity_diagram_to_task (i_input_channel); /* Create the activity
      diagram of the new task that contains an infinite loop with a
      read event to get the data sent by the co-simulator followed by
      a write channel to insert the data to the model.                  */
    add_CPU_in_architecture(i_input_channel); /* Add a new CPU in the
      architecture and associate the created task to it.                */
    link_CPU_to_bus(i_input_channel); /* Link the new CPU to the main Bus.
      */
end
for each i_output_channel in list_output_channels do
    disconnect_port_from_destination_side (i_output_channel); /* Disconnect the
      external port from its destination.                               */
    create_new_task(); /* Create new task for the external input channel
      */
    connect_port_to_new_task(i_output_channel); /* Connect the external port
      to the new task                                                   */
    add_event(i_output_channel); /* Add an event to the new task, this event
      will retrieve data from the co-simulator using the avs command
      that allows to add signals to a given event.                      */
    associate_activity_diagram_to_task (i_output_channel); /* Create the activity
      diagram of the new task that contains an infinite loop with a
      read event, followed by a read channel to remove samples from
      the channel.                                                      */
    add_CPU_in_architecture(i_output_channel); /* Add a new CPU in the
      architecture and associate the created task to it.                */
    link_CPU_to_bus(i_output_channel); /* Link the new CPU to the main Bus.
      */
end
```

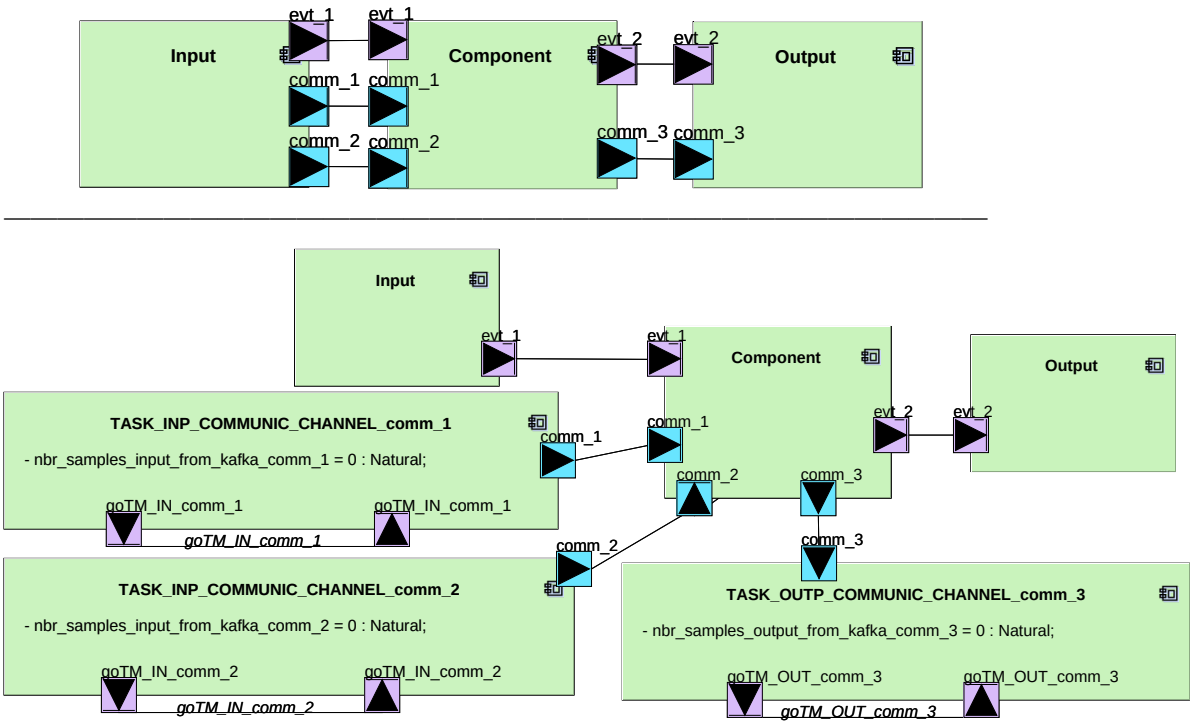

Figure 6: Top: The original Task diagram of the initial model. Bottom: The auto-generated task diagram of the Co-simulation model.

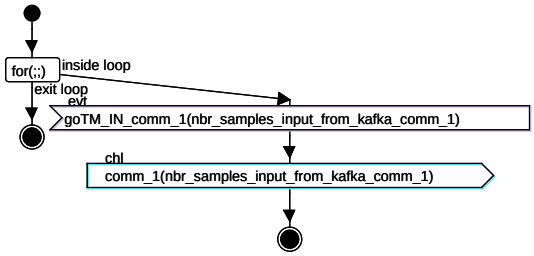

Figure 7: The auto-generated activity diagram for Channel $comm_1$

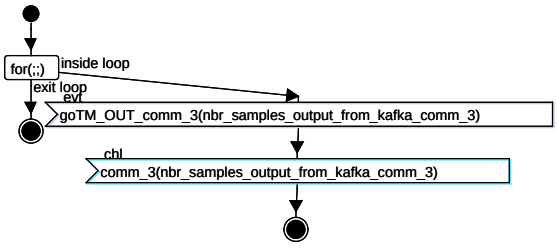

Figure 8: Generated activity diagram for Channel $comm_3$

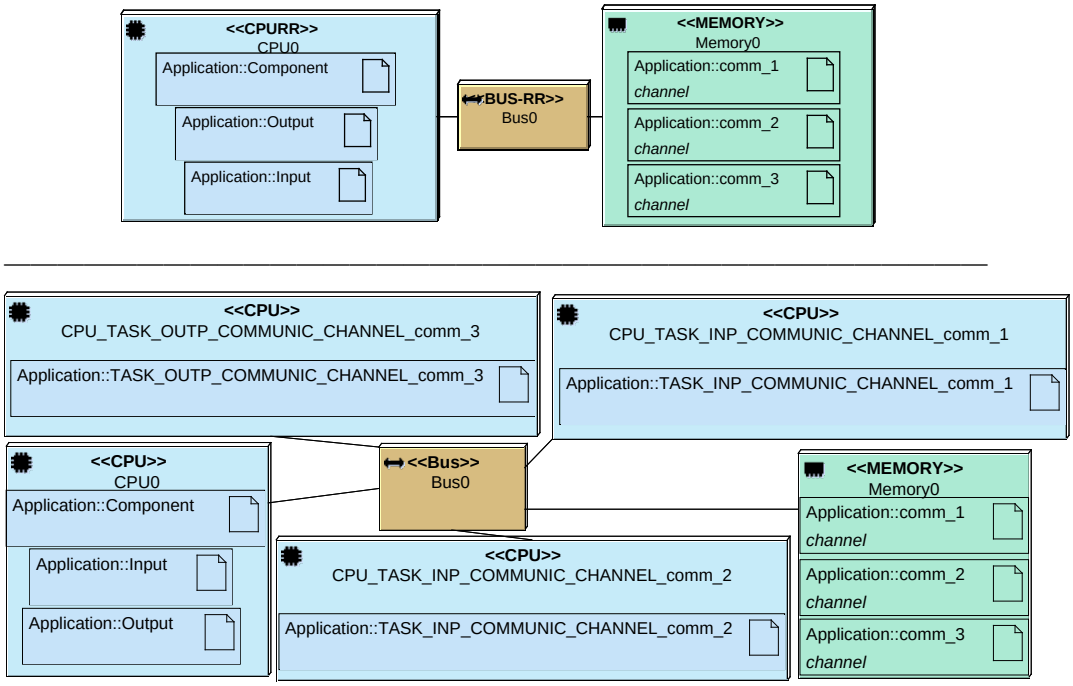

Figure 9: Top: The original architecture of the initial model linked to the top task diagram in Fig. 6. Bottom: The auto-generated architecture of the Co-simulation model linked to the bottom task diagram in Fig. 6.

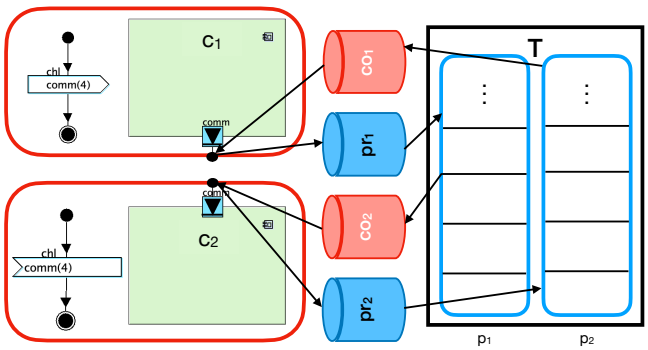

Figure 10: External communication between c_1 and c_2 via a BRBW channel

First of all, a topic denoted T with two partitions p_1 and p_2 is created to apply the correct communication semantics between c_1 and c_2. p_1 indicates the current number of samples that can be consumed by c_2 while p_2 contains the number of samples that has been already read by c_2.

For c_1, we create a producer pr_1 that puts in p_1 the number of samples that have been written. The producer pr_1 detects the written samples by analyzing in real-time the simulation trace of c_1. c_1 relies on p_2 via consumer co_1 to know how many samples have been already read by c_2 to figure out how many samples can be transmitted to p_1.

For c_2, we create a consumer co_2 to get from p_1 the number of samples that has been written by pr_1, and we use a producer pr_2 to send in p_2 the number of samples that have been read based on the simulation trace of c_2. So, the consumer co_2 is blocked until a new element is added to p_1 or a read is performed by c_2.

Fig. 10 depicts the communication between c_1 and c_2 and presents the different producers and consumers that are used to maintain correct communication semantics.

Algorithm 2 features how the channels at output side (component c_1) are handled for a BRBW channel and Algorithm 3 shows how to handle channels at input side (component c_2) for a BRBW channel.

Algorithm 2: Algorithm of the output side (component c_1) for a BRBW channel

```
search ← search_write_trans_in_trace(channel_name); /* Check if there is a write
    transaction of this output channel in the trace.                          */
if search is True then
    nb_samples_written ← get_from_trace_number_of_samples_written (channel_name);
        /* Parse, from the trace, the number of samples of the external output channel that has
        been written.                                                         */
    send_message(pr1, T, 0, nb_samples_written); /* Send the number of samples in p1 of T
        created for this channel using its producer pr1.                      */
end
nb_samples_read_by_c2 ← read_first_message_no_blocking (co1); /* Check if there is a
    new message in p2 since the last consumption and return the value of the first new message
    (element).                                                                */
if nb_samples_read_by_c2 != "" then
    exec_command("avs "+ name_of_event_created_for_channel_output + " 1 " +
        nb_samples_read_by_c2); /* Add the number of samples obtained from p2 to the event of
        the new task that was created for the output external channel. Thanks to the avs command
        of the simulator that allows to add virtual signals for a given event. By adding the
        number of the samples read by c2 to the event a read transition will be succeeded to
        remove the obtained number of samples from the output channel buffer. */
end
```

Algorithm 3: Algorithm of the input side (component c_2) for a BRBW channel

```
1  check_new_data ← check_new_data_in_topic(channel_name); /* Check if there is new
       data since the last consumption in p1 of the topic T that was created for this channel.  */
2  if check_new_data is True then
3      nb_samples ← read_first_message_blocking(co2); /* Consume the first new message
           since the last consumption and return the number of samples found in p1 using the consumer
           co2 created for this input channel.                                                  */
4      exec_command("avs "+ name_of_event_created_for_channel_input + " 1 " +
           nb_samples); /* Add the number of the new samples obtained from p1 to the event of the
           new task that was created for the input external channel (for example the event
           goTM_OUT_comm_1 in Fig. 7). By adding the number of the samples written by c1 to the
           event, a write transition will be succeeded to add the obtained number of samples to the
           input channel buffer.                                                                */
5  end
6  search ← search_read_trans_in_trace(channel_name); /* Check if there is a read
       transaction of this input channel in the trace.                                          */
7  if search is True then
8      nb_samples_read ← get_from_trace_number_of_samples_read (channel_name);
           /* Parse, from the trace, the number of samples of the external input channel that has
           been read.                                                                           */
9      send_message(pr2, T, 1, nb_samples_read); /* Send the number of read samples in p2 of
           the topic T using the producer pr2.                                                  */
10 end
```

4.2.2 Blocking Read, No Blocking Write with finite FIFO Event

Let's consider two components c_1 and c_2. c_1 has an external output port of type "no blocking write event with finite FIFO", and c_2 has an external input port of type "blocking read event". Because they are external, these two ports must exchange events via our platform. The platform is configured as follows.

First of all, a topic T with two partitions p_1 and p_2 is created. p_1 indicates the number of events that can be received while p_2 is used to store the parameters of each sent event.

At sending side (component c_1), we create two consumers co_{1_1} and co_{1_2}. co_{1_1} is responsible to get the current number of events to be read from partition p_1 and Consumer co_{1_2} handles parameters values stored in p_2. Thus, if the FIFO is not full and the simulation trace has a sending event transition, then p_1 adds the corresponding event to p_1 and the parameters of this event to p_2. For instance, if the event is $evt(1,true)$, and p_1 contains the value 2, then value value in p_1 is now set to 3 (one more event), and 1 and $true$ are added to p_2. We also create a producer pr_1 to update the new current number of samples in the first partition p_1 each time a new send command is successfully performed by c_1. When this is the case, the producer pr_1 puts the corresponding parameters of the event in p_2.

At receiving side (component c_2), we create also two consumers co_{2_1} and co_{2_2}. co_{2_1} is responsible for reading parameters values from p_2 while consumer co_{2_2} gets from the first partition p_1 the number of currently events to be read. Also, we create a producer pr_2 that sends in p_1 the new current number of samples when an event is read by c_2. Thus, co_{2_1} blocks until p_1 contains a value greater than 0. When this is the case the second consumer co_{2_1} reads parameters from p_2 and the event is forwarded to component c_2.

Fig. 11 summarizes the communication between c_1 and c_2 and presents the different elements used to maintain correct communication semantics.

Algorithm 4 gives the algorithm to handle events at output side for BRNBW events with finite FIFO.

Algorithm 5 gives the algorithm used at input event side for BRNBW events with finite FIFO.

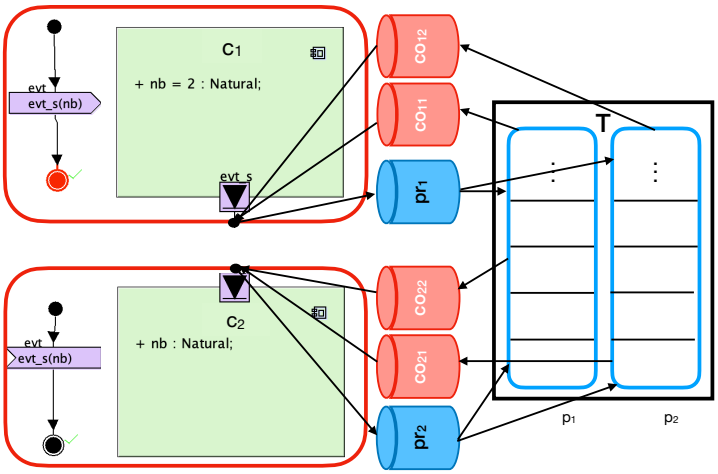

Figure 11: External communication between c_1 and c_2 via a BRNBW event with finite FIFO

Algorithm 4: Algorithm of the output side (component c_1) for a BRNBW event with finite FIFO

```
1  search ← search_send_trans_in_trace(event_name); /* Check if there is a send transaction
       of this output event in the trace.                                          */
2  if search is True then
3      parameters_values ← get_from_trace_parameters_values (event_name); /* Parse,
           from the trace, the values of the parameters of the external output event that has been
           send.                                                                   */
4      send_message(pr1, T, 1, parameters_values); /* Send the parameters values in p2 of the
           topic T created for this event using its producer pr1.                  */
5      last_nb_samples, list_indexes_to_ignore ← read_last_message_no_blocking_no_offset
           (co1_1); /* Get the last current number of samples and the current list of the indexes of
           parameters values to ignore from p1.                                    */
6      if last_nb_samples < FIFO_size then
7          send_message(pr1, T, 0, str(last_nb_samples+1)+list_indexes_to_ignore); /* Send
               the new current number of samples with the list of indexes of parameters values to
               ignore in p1.                                                       */
8      else
9          index_parameters_to_ignore = offset_last_message_no_blocking_no_offset (co1_2);
               /* Get the index of parameters to ignore according to p2.          */
10         send_message(pr1, T, 0, str(last_nb_samples+1) + list_indexes_to_ignore +
               index_parameters_to_ignore); /* Send the new current number of samples with the
               updated list of indexes of parameters values to ignore in p1.       */
11     end
12 end
```

Algorithm 5: Algorithm of the input side (component c_2) for a BRNBW event with finite FIFO

```
1  cur_nb_samples, indexes_to_ignore ← read_last_message_no_blocking_no_offset (co_{2_2});
       /* Get the current number of samples and the indexes to ignore from p_1.        */
2  if cur_nb_samples > 0 then
3      offset_message, parameters_values = read_offset_first_message_blocking (co_{2_1});
           /* Get the index of the current message and its value in p_2.               */
4      while offset_message ∈ indexes_to_ignore do
5          offset_message, parameters_values = read_offset_first_message_blocking (co_{2_1});
               /* Get the index of the current message and its value in p_2.           */
6      end
7      exec_command("avs "+ name_of_external_input_event + number_of_parameters +
           parameters_values); /* Add to the external output event the new sample with the values
           of its parameters that has been send by c_1 using avs command.              */
8  end
9  search ← search_wait_trans_in_trace(event_name); /* Check if there is a wait transaction
       of this input event in the trace.                                               */
10 if search is True then
11     cur_nb_samples, indexes_to_ignore ← read_last_message_no_blocking_no_offset
           (co_{2_2}); /* Get the current number of samples and the indexes to ignore from p_1.  */
12     send_message(pr_2, T, 0, str(cur_nb_samples-1)+indexes_to_ignore); /* Update the
           current number of samples in p_1.                                           */
13 end
```

5 Co-simulation of SysML and SystemC models

In this section, we show how our platform can be used to co-simulate components designed in TTool (SysML) and others designed in SystemC. A ZigBee decoder serves as case study: we use the version described in [EAP17]. ZigBee is a wireless communication scheme adapted to low-power devices.

In this example, we have divided the ZigBee decoder into 5 components, as shown Fig. 12:

- Source is modeled and simulated by TTool.
- symbol2ChipSeq is modeled and simulated using SystemC. This component gets data from Source by the channel *Symbol2ChipSeqChIn* and the event *Symbol2ChipSeqEvtIn*.
- Chip2Octet is modeled and simulated using TTool. This component gets data from symbol2ChipSeq by the channel *Chip2OctetChIn* and the event *Chip2OctetEvtIn*.
- CW is modeled and simulated using SystemC, it contains 3 components CWL, CWP_I and CWP_Q. This sub-system gets data from Chip2Octet by the channel *CWLChIn* and the event *CWLEvtIn*.
- Sink is modeled and simulated using TTool. This component gets data from CW by the channels *CWP_IChOut*, *CWP_QChOut* and the events *CWP_IEvtOut* and *CWP_QEvtOut*.

All the external channels, in this example, are of type blocking read blocking write with size 2 and the external events are of type blocking read, no blocking write with infinite FIFO. Also, we set the control parameter *size* to 13.

After having modeled all the above-mentioned sub-systems, we first apply our model update approach. Then, our approach starts all five simulation engines (one for each component) and our platform, thus including all the necessary Kafka consumers and producers. Fig. 13 shows the resulting simulation traces for the five components, where

- "W" denotes a write in a channel.
- "R" denotes a read operation from a channel.
- "S" denotes a send event.
- "G" denotes a get event.

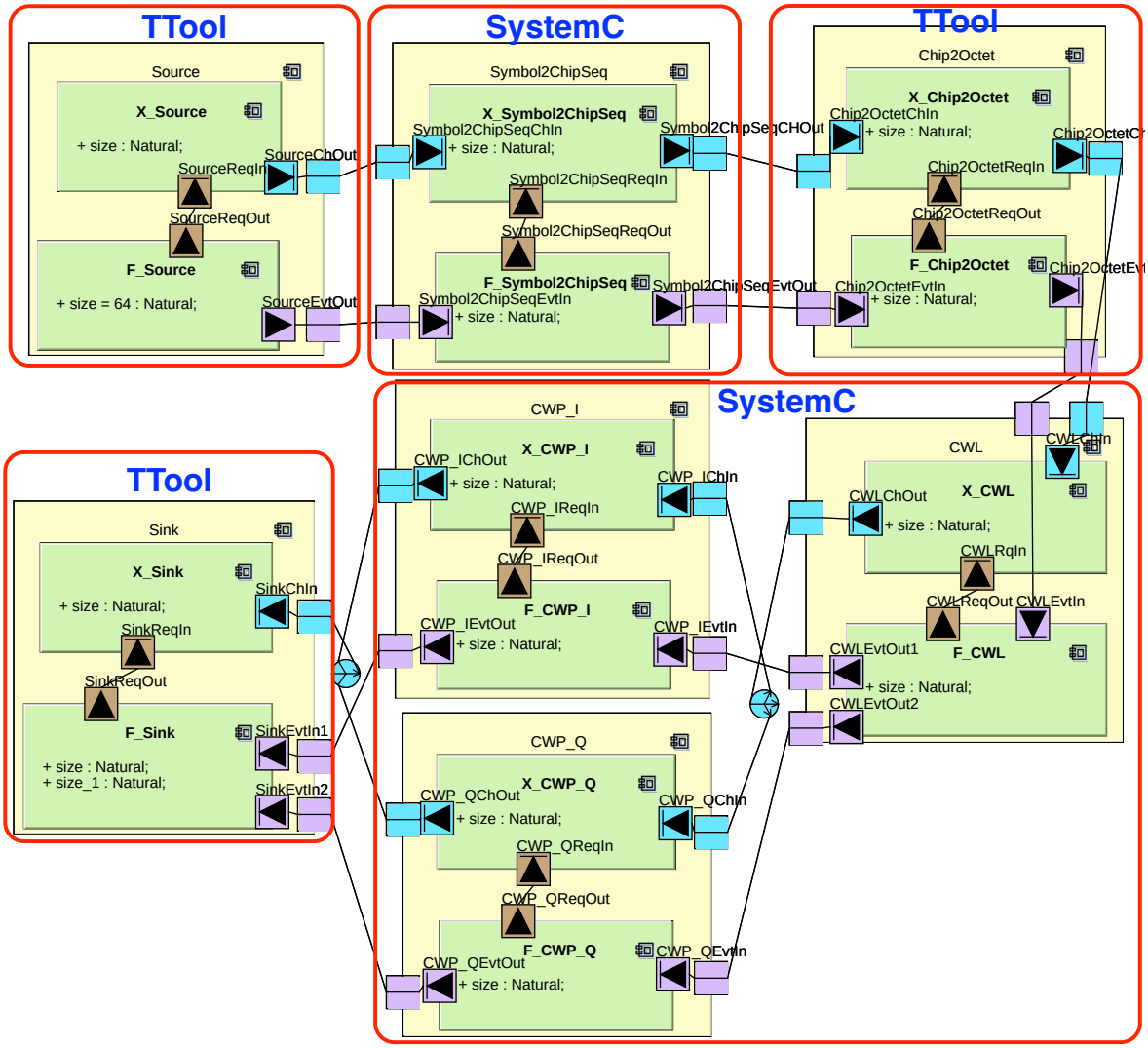

Figure 12: Overall view of Zigbee application

- The red arrows show the communication and the data exchanged between components from the first one (Source) until the last component (Sink) which proves that our platform was able to ensure the communication between the different simulation engines.

By analyzing the simulation trace, we can easily verify that our platform respects the simulation semantics. For instance, the channel *SourceChOut* of the component "X_Source" sends 2 samples to "X_Symbol2chipSeq" on every write transaction which matches with the size of the external channels, only in the last transaction where 1 sample was sent (at $t \in [40,41]$). Starting with the second write transaction in "X_Source", these write transactions are performed after read operators in "X_Symbol2chipSeq". A total of 13 samples were sent to "X_Symbol2chipSeq". In this use case, we showed an example of communication between components modeled and simulated by different platforms. The overall execution time of the Zigbee simulation is equal to 23 seconds, this time is calculated between the first sending by the component Source and the last signal received by the component Sink. The high value of the execution time is due to the latency of Kafka broker. Other alternatives can be used to improve the performance of the approach, for example by using Sockets instead of Kafka.

6 Conclusion

In this paper, we highlighted the need to integrate components together without common modeling languages nor model transformations. Thus, we presented a method that allows to co-simulate, in real-time, embedded systems with heterogeneous components while respecting usual communication semantics between the components to be integrated. Our approach is based on simple model updates, on Kafka, and on the use of consumers and producers.

We have shown that our method applies to mid-size systems such as Zigbee.

Having a distributed co-simulation has a cost in term of simulation time: we intend to lower the extra latency as much as possible: an option is to experiment with other brokers, even if Kafka has the advantage to be a recognized platform for distributed data exchange and is platform agnostic. We also intend to experiment with more modeling and simulation environments like AADL.

References

[23] *TTool*. https://ttool.telecom-paris.fr. [Online]. 2023 (cit. on p. 2).

[Ati+08] Rabie Ben Atitallah, Philippe Marquet, Éric Piel, Samy Meftali, Smaïl Niar, Anne Etien, Jean-Luc Dekeyser, and Pierre Boulet. "Gaspard2: from MARTE to SystemC Simulation". In: *Workshop on Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML*. 2008 (cit. on p. 2).

[Bal+03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. "Metropolis: an integrated electronic system design environment". In: *Computer* 36.4 (2003), pp. 45–52. DOI: 10.1109/MC.2003.1193228 (cit. on p. 2).

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: *Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2006, pp. 3–12 (cit. on p. 2).

[Blo+11] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. V. Peetz, S. Wolf, and C. Claub. "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". In: *Proceedings of the 8th International Modelica Conference*. 2011, pp. 105–114. DOI: 10.3384/ecp11063105 (cit. on p. 2).

[EAP17] Andrea Enrici, Ludovic Apvrille, and Renaud Pacalet. "A Model-Driven Engineering Methodology to Design Parallel and Distributed Embedded Systems". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.2 (Jan. 2017), 34:1–34:25. ISSN: 1084-4309. DOI: 10.1145/2999537 (cit. on p. 13).

[Eke+03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. "Taming heterogeneity – the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (2003), pp. 127–144. DOI: 10.1109/JPROC.2002.805829 (cit. on p. 2).

[HEF19] Eman Hammad, Mellitus Ezeme, and Abdallah Farraj. "Implementation and development of an offline co-simulation testbed for studies of power systems cyber security and control verification". In: *International Journal of Electrical Power & Energy Systems* 104 (2019), pp. 817–826. ISSN: 0142-0615. DOI: https://doi.org/10.1016/j.ijepes.2018.07.058 (cit. on p. 2).

[KAP09] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. "Fast Simulation Techniques for Design Space Exploration". In: *Objects, Components, Models and Patterns: 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*. Ed. by Manuel Oriol and Bertrand Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–327. ISBN: 978-3-642-02571-6. DOI: 10.1007/978-3-642-02571-6_18 (cit. on p. 2).

[LD20] Giovanni Liboni and Julien Deantoni. "CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations". In: *ICISE 2020 - 5th International Conference on Information Systems Engineering*. Manchester/Virtual, United Kingdom, Nov. 2020 (cit. on p. 2).

[Mug+19] Chuma Francis Mugombozi, Rawad Zgheib, Thierry Roudier, Anthony Kemmeugne, Dmitry Rimorov, and Innocent Kamwa. "Collaborative Simulation of Heterogeneous Components as a Means Toward a More Comprehensive Analysis of Smart Grids". In: *2019 7th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. 2019, pp. 1–6. DOI: 10.1109/MSCPES.2019.8738794 (cit. on p. 2).

[Nee+14] Himanshu Neema, Jesse Gohl, Zsolt Lattmann, Janos Sztipanovits, Gabor Karsai, Sandeep Neema, Ted Bapty, John Batteh, and Hubertus Tummescheit. "Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems". In: *Lund University*. 2014, pp. 235–245 (cit. on p. 2).

[RS07] Waseem Raslan and Ahmed Sameh. "Mapping SysML to SystemC". In: *Forum on specification & Design Languages FDL*. ECSI, 2007, pp. 225–230 (cit. on p. 2).

[RS12] Elvinia Riccobene and Patrizia Scandurra. “Integrating the SysML and the SystemC-UML profiles in a model-driven embedded system design flow”. In: *Design Automation for Embedded Systems* 16 (2012), pp. 53–91 (cit. on p. 2).

[SAV21] Pierre de Saqui-Sannes, Ludovic Apvrille, and Rob Vingerhoeds. “Checking SysML Models Against Safety and Security Properties”. In: *Journal of Aerospace Information Systems* (Nov. 2021), pp. 1–13. DOI: 10.2514/1.i010950 (cit. on p. 2).

[Tav+16] Jean-Philippe Tavella, Mathieu Caujolle, Charles Tan, Gilles Plessis, Mathieu Schumann, Stéphane Vialle, Cherifa Dad, Arnaud Cuccuru, and Sébastien Revol. *Toward an Hybrid Co-simulation with the FMI-CS Standard*. Research Report. Apr. 2016 (cit. on p. 2).

[Tri15] Stavros Tripakis. “Bridging the semantic gap between heterogeneous modeling formalisms and FMI”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation SAMOS*. IEEE, 2015, pp. 60–69 (cit. on p. 2).

[VV09] Roberto Varona-Gómez and Eugenio Villar. “AADL Simulation and Performance Analysis in SystemC”. In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*. 2009, pp. 323–328. DOI: 10.1109/ICECCS.2009.11 (cit. on p. 2).

[ZAM20] Hui Zhao, Ludovic Apvrille, and Frédéric Mallet. “A Model-Based Combination Language for Scheduling Verification”. In: *Model-Driven Engineering and Software Development*. Ed. by Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić. Cham: Springer International Publishing, 2020, pp. 27–49. ISBN: 978-3-030-37873-8 (cit. on p. 1).

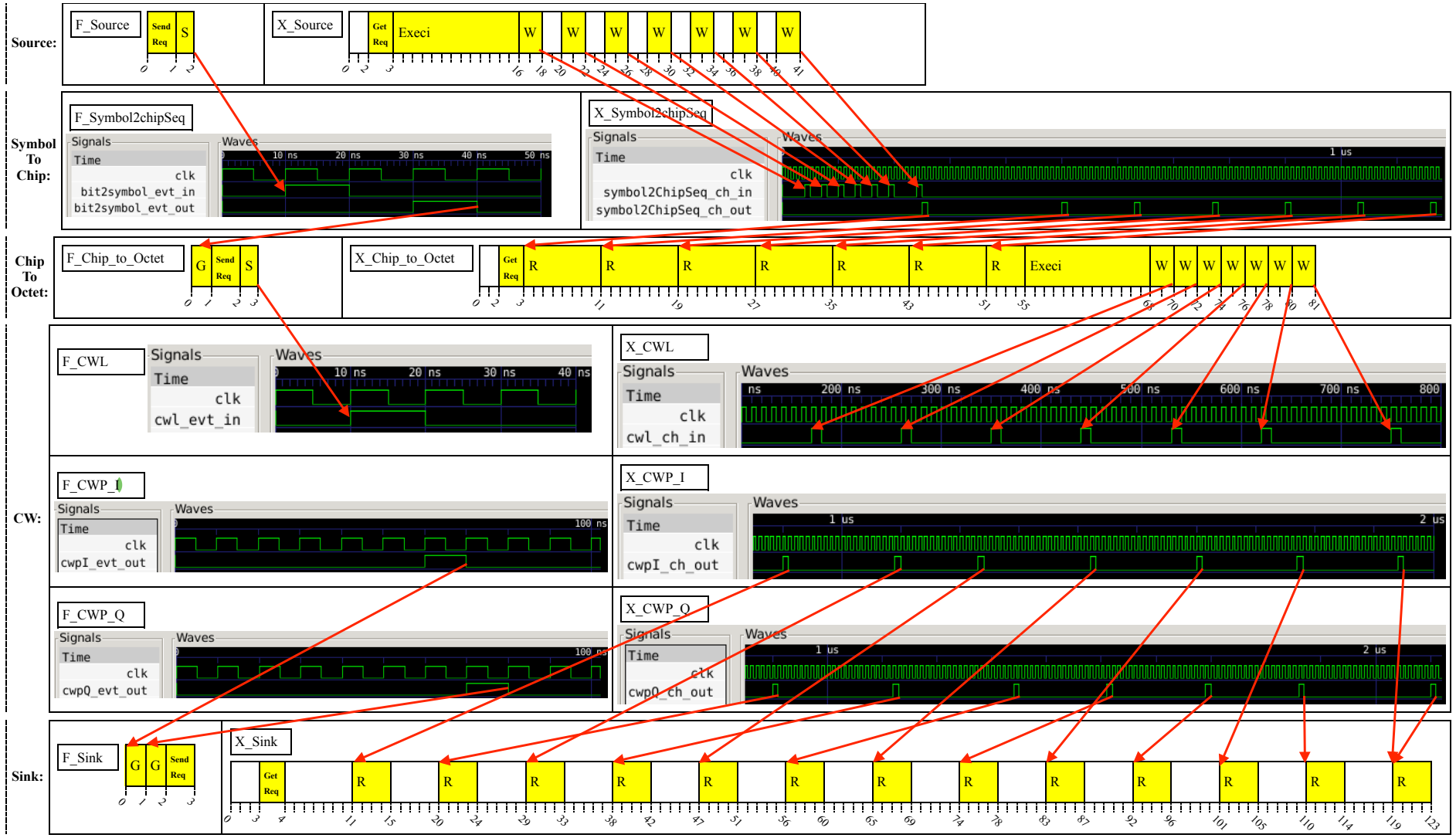

Figure 13: Zigbee: communication between the components