FORMALLY VERIFIED FMI ENABLED EXTERNAL DATA BROKER: RABBITMQ FMU

Casper Thule
Cláudio Gomes
Kenneth G. Lausdahl

Centre for Digital Twins
DIGIT, Aarhus University
Finlandsgade 22
8200 Aarhus N
Denmark
{casper.thule,claudio.gomes}@eng.au.dk and kenneth@lausdahl.com

ABSTRACT

Automated monitoring of deployed Cyber-physical Systems while they are operating can assist in detecting misbehavior. One way to achieve such monitoring is via co-simulation of a modelled system that corresponds to the deployed system. A necessary element in the realization of such a concept is data brokering between the systems. Our work addresses the topic of brokering external data into an Functional Mock-up Interface (FMI) 2.0 enabled co-simulation via a Functional Mock-up Unit (a simulation unit) called RabbitMQ FMU that outputs messages from an external message queue into the FMI enabled co-simulation.

Our contribution includes a formal specification that captures the semantics of RabbitMQ FMU and ensures a maximum delay on the messages relayed to the co-simulation even through an unreliable network. This formal specification is then used to generate test cases that exercise all states and transitions of the formal specification. The implementation is then subjected to these test cases to verify that it is a refinement of the specification. The verification procedure is ongoing and currently the implementation has successfully been subjected to 1,539,246 tests. The RabbitMQ FMU provides a general and self-contained solution to brokering external data into and FMI-enabled co-simulation with guaranteed delays.

Keywords: Digital Twins, Self-Adaptive systems, Co-Simulation, Functional Mock-up Interface, Formal Verification

1 INTRODUCTION

Cyber-Physical Systems (CPS) that integrate physical, software and network aspects are emerging, presenting challenges regarding not just their design, but also operation and maintenance. These systems are typically developed in a distributed fashion, which leads to partial solutions that have to be integrated (Van der Auweraer et al. 2013). Modeling and simulation can assist in the development of each partial solution and co-simulation can assist in their integration (Blochwitz et al. 2012, Gomes et al. 2018).

While co-simulation frameworks such as Maestro (Thule et al. 2019) or DACCOSIM (Galtier et al. 2015) support the engineering activities at design time, they are not enough to support the monitoring and maintenance activities that occur during system operation. Advanced monitoring techniques, supported by co-

SummerSim, July 20-22, 2020, Madrid, Spain;

simulations, are one of the pillars towards systematic production of self-adaptive CPS and digital twins (Weyns 2019, Zhou et al. 2019, Kritzinger et al. 2018).

The architecture of such monitoring systems combines both co-simulation and real-time data from the system operation, and is illustrated in Figure 1. The goal is to detect misbehavior of the deployed system based on simulations of the corresponding modelled system, calibrated using data from the deployed system. For example to detect performance degradation due to tear and wear. Developing such monitors is outside the scope of this publication.

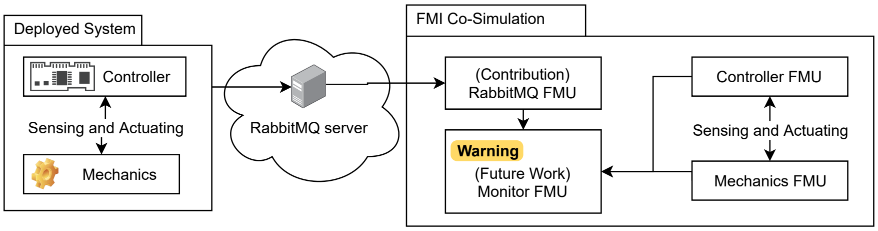

Figure 1: Example of a Co-simulation based monitoring system. The deployed system is fitted with sensors that upload data to a rabbitMQ queue through an unreliable network. Based on the co-simulation needs, our data broker (RabbitMQ FMU) decides when, and how much data, to fetch from the queue. It performs the correct extrapolation on the data received so that the other FMUs have a consistent view of the system.

In this paper we aim at enabling the development of advanced monitors by focusing on the problem of brokering external data from a deployed system so that it can be used in a co-simulation. Concretely, the approach to be presented concerns brokering messages from a message queue (RabbitMQ (Toshev 2015)) into an Functional Mock-up Interface (FMI) standard, version 2 (FMI v. 2.0 2014) enabled co-simulation. Moreover, we highlight the challenge of feeding signals to the co-simulation that are delayed by no more than a user-set tolerance Δ seconds by building a formal specification of the data broker. The data broker specification is then proven correct using model checking. Since the actual data broker implementation is made in C++ (for its performance), we used the data broker specification to generate tests that check whether the implementation is a refinement of the specification. This way we build confidence that the data broker implemented will either provide signals that are at most Δ seconds delayed, or will fail and not allow the co-simulation to progress. The reason why the data broker will fail is because it makes no assumptions on the network used to transport the signals. If the network has a guaranteed latency smaller than Δ, then the data broker will never fail. This system is a small step towards guaranteed monitoring and predictive maintenance of a CPS.

We show that the formal reasonancy allowed the quicker detection of nuanced problems, such as how to relate the initial co-simulation time with the timestamps of the messages coming from the system. The contributions of this paper are summarized as follows[1]:

- Generalized approach and tool for brokering data into an FMI-enabled co-simulation;
- Formal specification of the data broker and model-checking using TLA+ (Lamport 2002);
- Extensive testing of the implementation with tests derived from the formal specification.

The structure of the paper is the following: Section 2 presents background information; afterwards, Section 3 presents the formal specification and properties of the approach; next, we describe the implementation and the strategy for verifying that the implementation adheres to the specification in Section 4; and, finally, the paper concludes in Section 5.

---

[1] Data Broker Specification: https://gitlab.au.dk/into-cps/rabbitmq-fmu-tla/-/releases/1.1.0, Accessed July 3, 2020. - RabbitMQ FMU: https://github.com/INTO-CPS-Association/fmu-rabbitmq/,. Test Case Generator: https://gitlab.au.dk/into-cps/rabbitmq-fmu-tla/-/releases/tests-2020-july-1. All accessed July 3, 2020.

2 BACKGROUND

In this section, we give a brief overview of the conceps that are important for the reader to understand our contribution.

2.1 Co-simulation

Co-simulation consists of the theory and techniques to enable global simulation of a coupled system via the composition of simulators (Gomes et al. 2018). The *Functional Mock-up Interface* (FMI) 2.0 is a standard to enable co-simulation (Blochwitz et al. 2012). It describes how to develop a simulation unit, such that it can participate in an FMI enabled co-simulation. A simulation unit that adheres to the standard is referred to as a *Functional Mock-up Unit* (FMU).

The FMI *master* is the entity responsible for conducting a co-simulation by moving the data between the individual FMUs and asking them to progress in simulated time. The FMI describes an interface that an FMU has to implement. Here we highlight the *DoStep(t,H)* operation, which is invoked by the master on an FMU and informs the FMU that it must progress its internal time $t + H$ and compute its new outputs at that time.

Inputs, outputs and parameters of an FMU are described in a static description file, called the *modeldescription.xml*, which is also standardized by the FMI.

A basic FMI master performs initialization of the participating FMUs and then enters the following simulation loop: prompt the simulation units to progress in time by invoking the *DoStep* operation, and perform the exchange of inputs and outputs.

2.2 Model Checking and TLA+

We give an brief introduction to model checking and TLA+, and refer the reader to (Lamport 2002) and (Wayne 2018) for more details.

Model checking is the act of exploring all possible behaviors of a model and checking that they satisfy a specification (Baier and Katoen 2008). In TLA+, the model is specified as a non-deterministic state transition system, and the specification can be encoded as a set of assertions, invariants, computation tree logic (CTL) formulas, and deadlocking statements.

A state transition system consists of a specification of a state space, an initial state, and a state relation that details how a state leads to a set of successor states. In the next section, we describe the specification of the data broker and its environment.

3 SPECIFICATION

We start by describing how we modeled the data broker environment, then we describe its possible states, the initial states, and finally the state transition function. In each item, to conserve space, we provide an informal description. The full specification is available online[1].

There are two key elements in the environment of the data broker: the RabbitMQ queue, denoted as rabbit queue, and the master algorithm. Recalling Figure 1, the data-broker collects messages from the rabbit queue, to which data messages are uploaded via an unreliable network from the system. The system is any entity that produces a stream of messages. Each message consists of a timestamp (the absolute time in which the message was produced in the system), a variable name, and value.

The system may contain multiple independent sensors that produce independent streams of messages. However, we do not lose generality by considering a single stream of messages, because we make no assumptions on the order in which messages arrive at the rabbit queue. RabbitMQ makes messages available to the data broker in the LIFO order of their arrival, which is not necessarily the same order in which they were produced (their timestamps). The messages that the data broker receives are used to determine its output signals.

The output signals are read by a master, who feeds these signals to the appropriate FMUs in the co-simulation. As the co-simulation progresses, the master informs the data broker of the co-simulation time advances by invoking the $DoStep$ operation. The co-simulation time determines the value provided at the outputs by the data broker based on the related timestamp of the value.

It is the data broker's responsibility to sort (in the timestamp order) the messages that it reads from the rabbit queue, store them internally, and use them to define the output signals provided to the master algorithm. The variable name, in each message, refers to an output of the same name. Prior to execution, the user informs the data broker of all variable names via the standardized modeldescription.xml file, so the data broker can initialize its internal data structures. Moreover, the data broker has to produce the right output values at the given co-simulation time, so that the latter is as close as possible to the timestamp of the former, without violating causality. Causality is important because the co-simulation may be slower than the real system, which means that the co-simulation time may refer to a wall-clock time in the past, and it is therefore important that the master collects outputs from the data broker that correspond to that past system view.

When the data broker collects messages from the rabbit queue, there are no guarantees that the former will read all messages produced by the system, and in the order in which they are produced, as some messages might still be in transit. This means that, as the co-simulation progresses, the data broker will repeatedly attempt to collect messages from the rabbit queue, when it needs to compute new outputs.

To simplify, we start with the assumption that the data broker only reads a single stream of messages, all corresponding to the same variable. Therefore we assume for now that the data broker has a single output. We later discuss how the specification was generalized to multiple variables/outputs in Section 3.5.

3.1 State

At any point in execution of the specification, the state is given by: the rabbit queue state, the data broker state, the user-set maximum tolerable delay (or age), and the communication step size used in the next invocation to the DoStep function.

The rabbit queue state represents the messages that are available for collection by the data broker. The data broker state represents the messages processed so far, the current co-simulation time, and the current output value. Finally, the maximum tolerable delay and the next communication step size are Naturals. It will later become clear how these affect the behavior of the data broker. We now describe, in a bottom up fashion, the different concepts that make up the state of the data broker.

A signal represents the complete stream of messages (corresponding to the same variable) to be read by the data broker. Without loss of generality, we assume that the value of each message is its timestamp, which is a Natural number. For instance, the signal $\langle 3, 1, 2, 4 \rangle$ represents four messages, to be collected by the data broker in the order from left to right (i.e., the first message to be read is 3, then 1, and so on). Because the data broker may not read all messages in a signal in one access, we need an extra Natural including zero ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) to represent how many messages can be read by the data broker on the next access.

For example, the following rabbit queue state means that, on the next access, the data broker will read the messages $\langle 3, 1\rangle$: $[signal \mapsto \langle 3, 1, 2, 4\rangle, visiblen \mapsto 2]$.

The data broker state introduced above is detailed as follows. It is a structure with a Natural including zero (the current co-simulation time), a sorted sequence of Naturals (the messages read so far), a Natural (the current output value), and a Boolean (whether the data broker is blocking the co-simulation). A blocked co-simulation means that the data broker is waiting to read more data from the rabbit queue. For instance, the following state represents the data broker at time 2, having read messages $\langle 3, 1\rangle$, with output value 1, and not waiting for more messages: $[time \mapsto 2, queues \mapsto \langle 1, 3\rangle, outputs \mapsto 1, blocked \mapsto \text{FALSE}]$.

The complete state is represented by a structure aggregating the states introduced above. Given a maximum number of messages M, the complete state has the following form:

$$\begin{aligned} fmu &\mapsto [time \mapsto t, queues \mapsto q, outputs \mapsto o, blocked \mapsto b] \\ env &\mapsto [signals \mapsto [signal \mapsto s, visiblen \mapsto n], maxDelay \mapsto d, H \mapsto h], \end{aligned} \tag{1}$$

where: fmu denotes the data broker state; env denotes the rabbit queue state, the maximum delay, and the communication step size; $t \in \mathbb{N}_0$; q is a finite sorted sequence of Naturals; $o \in \mathbb{N}_0$; b is a Boolean; s is a finite sequence of Naturals; $n \in \mathbb{N}_0$; $d \in \mathbb{N}_0$; and $h \in \mathbb{N}_0$. As an example, the following represents a possible state after the data broker read two messages from the rabbit queue: $t = 2$, $q = \langle 1, 3\rangle$, $o = 1$, $b = \text{FALSE}$, $s = \langle 2, 4\rangle$, $n = 1$, $d = 0$, $h = 1$.

3.2 Initial State

The behavior of the data broker is deterministic. However, its environment (rabbit queue, maximum delay, and communication step size) is non-deterministic. As such, to explain what the possible initial states are, we focus only on the possible initial environment states. Intuitively, the set of possible initial states has to cover all possible complete signals, all possible initial visible signals, and all possible maximum delay configurations. Given a maximum number of messages $M \in \mathbb{N}$, the initial environment is any state with the following form:

$$[signals \mapsto \langle [signal \mapsto s, visiblen \mapsto n]\rangle, maxDelay \mapsto d, H \mapsto 0]] \tag{2}$$

where s can be any sub-sequence of elements of the set $\{1, ..., M\}$ with no duplicate messages[2], n is any integer less than or equal to $|s|$ (the length of s), and $d \in \{1, ..., M\}$.

3.3 State Transition Relation

We explain the state transition relation by detailing how a single state leads a set of successor states. Recall that the initial state refers only to the environment state. As such, we will split the state transition function in two parts: the first part details how the successors of any initial state are computed; and the second part details how the successors of any non-initial state are computed.

3.3.1 Successors of Initial State

Given an initial state of the form of Equation (2), the set of successor states is computed by first computing the data broker state (a single state, as the data broker is deterministic), and then computing the set of all

---

[2] Duplicate messages can happen in an unreliable network. Our implementation can handle them by just dropping the duplicates.

successors from the resulting data broker state. The initial FMU state is computed roughly as follows. If $\mathit{visiblen} = 0$, then the data broker will block and the resulting data broker state is

$$[\mathit{time} \mapsto 0, \mathit{queues} \mapsto \langle\rangle, \mathit{outputs} \mapsto 0, \mathit{blocked} \mapsto \text{TRUE}]. \tag{3}$$

Otherwise: the data broker will read all $\mathit{visiblen}$ signals from the rabbit queue, sort the messages read according to their timestamp, and discard all but the last message m_0; and, the last message is the initial output of the data broker and its timestamp t_0 becomes the time in the resulting data broker state. The choice to use the last message received is a nuanced one and is revisited later in Section 3.5. The resulting state will therefore be:

$$[\mathit{time} \mapsto t_0, \mathit{queues} \mapsto \langle m_0\rangle, \mathit{outputs} \mapsto m_0, \mathit{blocked} \mapsto \text{FALSE}]. \tag{4}$$

Given the initial environment state in the form of Equation (2), and the initialized data broker state in the form of Equation (3) or Equation (4), the set of successor complete states is computed as follows. If the data broker is blocked, then there is a single successor state, which is the aggregation of the initial state, plus the fmu state, as illustrated in Equation (1). If the data broker is not blocked, then each next successor state will have the form of a complete state, where the fmu state is the same as the initialized data broker state, and the environment state is obtained from the initial state in Equation (2) as follows:

$$[\mathit{signals} \mapsto \langle[\mathit{signal} \mapsto s', \mathit{visiblen} \mapsto n']\rangle, \mathit{maxDelay} \mapsto d', H \mapsto h']] \tag{5}$$

where s' is the signal obtained by removing the messages read by the data broker from s in Equation (2), $n' \in \{0, ..., |s'|\}$, $d' = d$ (the maximum delay parameter remains the same throughout the execution), and $h' \in \{1, ..., M\}$.

In either case, the resulting complete state will therefore have the form introduced in Equation (1).

3.3.2 Successors of Non-Initial State

We detail the state transition relation between a predecessor state in the form:

$$\begin{aligned} \mathit{fmu} &\mapsto [\mathit{time} \mapsto t, \mathit{queues} \mapsto q, \mathit{outputs} \mapsto o, \mathit{blocked} \mapsto b] \\ \mathit{env} &\mapsto [\mathit{signals} \mapsto [\mathit{signal} \mapsto s, \mathit{visiblen} \mapsto n], \mathit{maxDelay} \mapsto d, H \mapsto h], \end{aligned} \tag{6}$$

and a successor state in the form:

$$\begin{aligned} \mathit{fmu} &\mapsto [\mathit{time} \mapsto t', \mathit{queues} \mapsto q', \mathit{outputs} \mapsto o', \mathit{blocked} \mapsto b'] \\ \mathit{env} &\mapsto [\mathit{signals} \mapsto [\mathit{signal} \mapsto s', \mathit{visiblen} \mapsto n'], \mathit{maxDelay} \mapsto d', H \mapsto h'], \end{aligned} \tag{7}$$

There are two main cases in this relation: A) there are not enough signals in s to have a non-trivial state relation; and B) otherwise. Case A is triggered when s is an empty sequence or $b = \text{TRUE}$. In that case, the resulting state is the same as the current state. That is, $t' = t$, $q' = q$, $o' = o$, $b' = b$, $s' = s$, $n' = n$, $d' = d$, and $h' = h$. In case B, first the fmu state will be updated by the DoStep operation, explained below. Then the set of new successor states is computed on the resulting fmu state and current environment state.

The DoStep operation is modelled by the following steps:

1. Check if the internal signals are enough to compute an output while respecting the maximum delay constraint.

2. If that is the case, then the data broker will not block (b' = FALSE), and will:
   (a) update the internal time: $t' = t + h$;
   (b) set the output to the value of the message whose timestamp is the closest (from the left) to t';
   (c) drop the internal messages whose timestamp is from the past, yielding q'.
3. Otherwise, the data broker will have to access the rabbit queue of the current environment state:
   (a) Read n messages from the rabbit queue s, merge them with the fmu queue q and sort the result.
   (b) Check if the resulting sequence can be used to produce an output at time $t + h$ that satisfies the delay constraints. If so, then $b' = \text{FALSE}$. Otherwise $b' = \text{TRUE}$.
   (c) If $b' = \text{TRUE}$ then $t' = t$, otherwise $t' = t + h$.
   (d) Set the output to the value of the message whose timestamp is the closest (from the left) to t'.
   (e) Drop the internal messages whose timestamp is from the past, yielding q'.

The DoStep operation yields the successor fmu state. From that state, one can compute the set of successor environment states as follows: the environment signal s' is given by s except for the messages read by the data broker; $n' \in \{0, ..., |s'|\}$, $d' = d$, and $h' \in \{1, ..., M\}$.

The successive applications of the relations described in Sections 3.3.1 and 3.3.2 to the initial states described in Section 3.2 yields the complete state space. The assumptions made in this specification are discussed in Section 3.6.

3.4 Correctness Properties

We present the main correctness criteria informally, and we refer the reader to the online specification[1] for all the properties used. Each property is an invariant, encoded as an assertion that is checked by the TLC model checker between a pair of states S, S' where S' is a successor state of S. If one of the properties does not hold, the model checker produces the state trace that leads to the property violation.

If S' is a successor to the initial state (as in Section 3.3.1), then the main property is: if the fmu did not block on S', then its time must be greater than 0, and its output must be equal to its timestamp. If S' is a successor to a non initial state (as in Section 3.3.2), then the main properties are: (i) if the fmu is blocked in S', then its time must not have changed; (ii) otherwise, its time must have progressed exactly by h units in S, its output must be smaller than or equal to its time, and the difference between its current time and its output must be smaller than or equal to the maximum tolerable delay.

3.5 Generalization to Multiple Signals

Until now we made the simplifying assumption that the data broker reads only one signal. In practice, we specified its behavior for a user configurable number of signals. This has the following important implications: (i) The possible initial state set covers all possible combinations of different signal sequences for each signal. (ii) The initial time is the maximum timestamp for all messages read. (iii) The data broker will block if at least one signal does not satisfy the maximum delay constraint. (iv) The data broker will access to rabbit queue if at least one signal does not satisfy the maximum delay constraint.

The initial time is the maximum of all messages (of all signals) read, for two reasons. First, the last message is the one that is closest to the actual wall-clock time. Moreover suppose that we choose the earliest message to be the co-simulation time. Then every other message from every other signal would be considered a future value for that signal. This would cause the data broker to block waiting for past values that satisfy the maximum delay, so as to preserve causality. This is because the data broker does not perform interpolation of future values, and it very important for the interpretability of the simulation results.

3.6 Discussion on Abstractions Made

To develop this specification, we made multiple abstractions that are addressed in the implementation (see Section 4.2). The most important is that there is finite number of messages and variables, dictated by user configurable constants. As a result, the state space is finite. Its size is affected by the maximum number of messages and maximum number of variables. We argue that this finite state space is representative of all states that the data broker FMU can face in practice. To see why, note that what matters is the relative ordering of the messages in a signal, and not their absolute timestamp. In other words, the behavior of the model for a stream of N messages of a signal with timestamps $t_0, t_1, \ldots, t_N$, is the same as for a stream of messages with timestamps $t_0 + C, t_1 + C, \ldots, t_N + C$, for any constant $C > 0$. As such, during the actual execution of the data broker implementation, the state of the rabbit queue repeats itself infinitely many times, even as the timestamps of the messages increase indefinitely.

In our experiments, based on coverage, there is evidence that the number of messages must be at least 4 and the number of variables/outputs must be at least 2. It is outside the scope of this paper to explain why this is the case in a formal fashion (ongoing work).

4 IMPLEMENTATION AND VALIDATION

The specification of the data broker and environment, introduced in Section 3, hereby referred to as the *specification*, was developed concurrently with the implementation, due to time pressure and developer specialization. In order to ensure that the implementation is a refinement of the specification, we employ a testing strategy that, given enough computation time, will test all behaviors of the specification and compare them with the implementation. Here, to keep the explanation simple, we will describe only the testing approach for the state transition relation between a non-initial state and its successor states, as described in Section 3.3.2. We refer the reader to the online code[1] for the complete description.

4.1 Testing Strategy

The testing framework works by exploring the state space of the specification, as generated by the model checker. The state space is a graph whose edges are pairs of states $S \to S'$, that have the form of Equation (6) and Equation (7). Each edge induces a refinement test as follows. The source state S is mapped to the implementation state. Then, the DoStep operation is invoked in the implementation. The resulting state of the implementation is compared with S'. If it is equivalent to S' then the test passes. Otherwise the test fails.

This kind of test is functional in nature, since it requires the possibility of setting the pre state and performing an action regardless of what happened before. As an example, it shall be possible to test a DoStep action independently of any initialize actions. This is quite different from an execution of the FMU according to the FMI standard, since such an execution will always have executed the initialize action before any DoStep actions. Thus, the testing strategy we employed places restrictions on the structure of the implementation, which is described in Section 4.2. The structure of the testing strategy is summarized in Figure 2.

4.2 Implementation

We give an overview of the implementation, with a focus on the main differences with respect to the specification. The reader is referred to the online code for a complete description[1]. In order to enable the testing

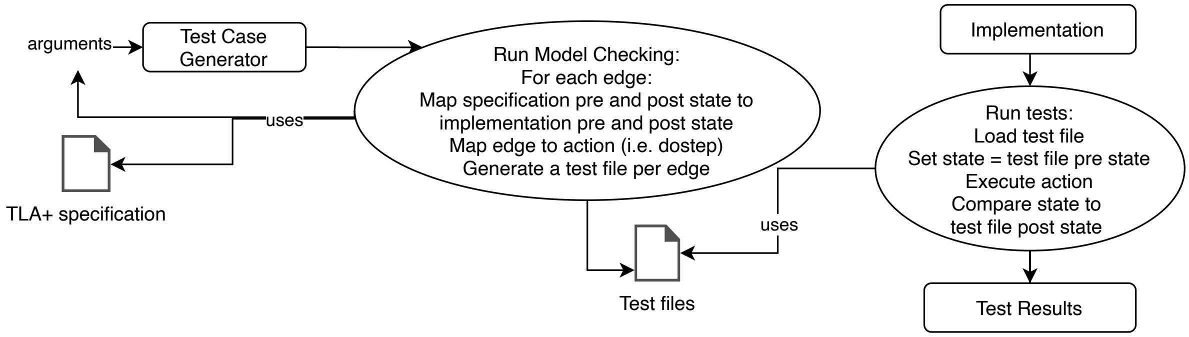
arguments
Test Case Generator
Run Model Checking:
For each edge:
Map specification pre and post state to implementation pre and post state
Map edge to action (i.e. dostep)
Generate a test file per edge
uses
TLA+ specification
Test files
uses
Implementation
Run tests:
Load test file
Set state = test file pre state
Execute action
Compare state to test file post state
Test Results

Figure 2: The testing strategy. The Test Case Generator extends the TLC Model Checker in order to receive state information. As the model checker executes, the test case generator maps the individual states and transitions to a structure expected by the implementation, and according to the test structure as described above. Finally, each test is serialized, such that the implementation can load and execute them.

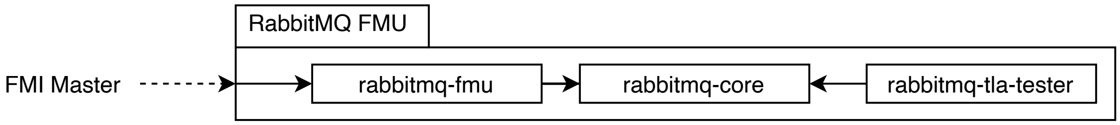

Figure 3: RabbitMQ FMU Internals. `rabbitmq-fmu` implements the FMI interface and connection to a RabbitMQ server. An `FMI Master` communicates with `rabbitmq-fmu`, which makes use of `rabbitmq-core` that corresponds to the implementation of the specification (Section 3). Thus, `rabbitmq-core` can operate in a functional fashion as required by Section 4.1. `rabbitmq-tla-tester` can then set the state of `rabbitmq-core`, query it to perform an action, and verify the resulting state.

strategy described in Section 4.1, and keep the implementation as close to as possible to the specification, we split the implementation into two parts, as described in Figure 3.

The most important differences between the implementation and the specification are:

- RabbitMQ FMU has to open a connection a RabbitMQ server, and therefore the connection details have to be described in its `modeldescription.xml` file.
- The timestamps and time used in the co-simulation are real numbers and represented in ISO8601. Therefore when checking for equality and computing time advances, the fmu needs to use a precision parameter. The specification has timestamps relative to time 0. The implementation therefore calculates a time offset based on the messages processed during initialization.
- When there are no signals in the rabbit queue, the implementation cannot literally block and wait for more. Instead, it stays responsive at all times by using a `communicationTimeOut` parameter.
- The implementation prioritizes memory and processing efficiency, above conceptual separation of the different stages of queue processing. The specification makes a clear separation, sacrificing efficiency.

The above differences make it impractical to automatically generate part of the implementation code. In the best case scenario that parts would be automatically generate, we would still have to prove that the implementation is a refinement of the specification.

The implementation was tested following the approach outlined in Section 4.1. At the end of each test, the pre and post implementation tests are used to evaluate the correctness properties described in Section 3.4. These properties were translated to C++ Boolean statements.

5 CONCLUDING REMARKS

We described the data broker specification, implementation, and our approach to ensure that the implementation is a refinement of the specification. The outcome is a formally verified data broker that ensures co-simulations will observe values from the deployed system that are at most Δ seconds delayed from the time of their production, and transmited over an unreliable network. We focus on remote monitoring because, for visualization and optimization, local monitoring would interfer with the resources of the system.

Related Work. To the best of our knowledge, there is no work focused on bridging network data to FMI based co-simulations with the guarantees described above. However, the techniques we used are not new: we were inspired by model based testing approaches (Li et al. 2018) and refinement checking (Gibson-Robinson et al. 2014, Leuschel and Butler 2005).

This work can have an impact on approaches that implement monitors of CPS. For instance, in (Zheng et al. 2019), the authors describe a remote monitoring system for a welding production line. The monitoring system receives the data from the machines (positions, current, etc.) and displays this data in a 3D model (i.e., the position data is translated into movement of the 3D objects, and other data is displayed in overlay screens). The monitoring system gets its data via the OPC UA specification (see, e.g., (Schwarz and Borcsok 2013) for an introduction). The authors show empirically that the data is delayed by about 1 second. With our work, which could be adapted to receive data from an OPCUA server, the authors would have a guarantee on the maximum delay of the data being used to inform the monitoring system or any simulation using the real-time data. Moreover, if those co-simulations comprise models that are sensitivity to delays (e.g., networked control systems (Hespanha et al. 2007)), the data broker may be used to prevent those instabilities by not allowing the simulation to progress when high delays occur (the maximum delay tolerated for a co-simulation can be approximated by stability analysis (Gomes et al. 2018, Appendix C)).

The concurrent development lead to the following insights regarding the general methodology of concurrent development and specification.

Levels of Abstraction. The specification and the implementation are written at different levels of abstraction. In the specification, the environment is explicitly represented, which is not the case for the implementation as it executes in a given environment. This means that the implementation has to follow an particular architecture in order to make it easier to test following the specification. Conversely, the specification should be adapted to facilitate the mapping between its states and the implementation states.

For example, consider an state space edge $S \rightarrow S'$. In the corresponding test, the `visiblen` property is important in S, as it describes how many messages are available in the environment queue. However, this property is not interesting in the post state, as what matters in the post state (for the C++ test) is the resulting fmu state in S'. This highlights the importance of clearly separating the different state components in the specification.

Implementation Details Matter. Thank to frequent communication between the implementation and specification teams, the specification was able to incorporate important performance enhancements that were not originally consider to be important. For example, the original specification of the data broker dictated that the data broker would always (whenever DoStep was invoked) read as many messages as possible from the rabbit queue. This behavior, as pointed out by the implementation team, was nefarious to the performance of the co-simulation, due to the many network accesses.

Specification Matters. Given the high level of abstraction of the specification, communication and validation of possible features was facilitated. Moreover, executing the tests revealed a bug in the implementation related to marking messages as in the future instead of current time. This was caused by a miss-typed `less than or equal to` that should have been a `less than`.

Limitations. At the time of writing, we were able to run the total state space of 1,880,865 tests of which 1,539,246 passed, 990 failed, and 340,710 was skipped as they were only relevant to the model checking of the model[3]. Currently the investigation of the failed tests are ongoing.

Working with the number of tests at this scale makes performance a priority. For this reason, the test case generator has a number of arguments to control the range of tests to generate, how much debug information to include in the generation of test cases and similar. This makes it possible to regenerate and serialize the 990 failed tests with more debug information, without regenerating and serializing the other test cases.

At the conceptual level, we assume that there is a data broker per rabbit queue source. It is the user that must set a maximum delay is appropriate to the application domain, and is supported by the network.

Future Work. It is our vision that RabbitMQ FMU will be used in both industry and research projects, which will further reveal more limitations of our approach. An already known limitation is that RabbitMQ FMU cannot send messages to a message queue, which is a planned upgrade to both the specification and implementation. This will enable *closing the loop* between the co-simulation and the real world, and allow for self-adaptive systems to be developed. Work has been initiated on developing concrete case studies relying on the RabbitMQ FMU to develop these systems.

REFERENCES

Baier, C., and J.-P. Katoen. 2008. *Principles of model checking*. MIT press.

Blochwitz, T., M. Otter, J. Åkesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012, 09. “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models”.

Galtier, V., S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui, and G. Plessis. 2015. “FMI-Based Distributed Multi-Simulation with DACCOSIM”. In *Spring Simulation Multi-Conference*, pp. 804–811. Alexandria, Virginia, USA, Society for Computer Simulation International San Diego, CA, USA.

Gibson-Robinson, T., P. Armstrong, A. Boulgakov, and A. W. Roscoe. 2014. “FDR3 — A Modern Refinement Checker for CSP”. In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, E. Ábrahám, and K. Havelund, Volume 8413, pp. 187–201. Berlin, Heidelberg, Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

Gomes, C., C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. 2018. “Co-simulation: a Survey”. *ACM Computing Surveys* vol. 51 (3), pp. Article 49.

Gomes, C., C. Thule, P. G. Larsen, J. Denil, and H. Vangheluwe. 2018. “Co-simulation of Continuous Systems: A Tutorial”. Technical Report arXiv:1809.08463, University of Antwerp, Belgium. arXiv: 1809.08463.

Hespanha, J., P. Naghshtabrizi, and Y. Xu. 2007. “A Survey of Recent Results in Networked Control Systems”. *Proceedings of the IEEE* vol. 95 (1), pp. 138–162. arXiv: 1011.1669v3 ISBN: 0018-9219 VO - 95.

Kritzinger, W., M. Karner, G. Traar, J. Henjes, and W. Sihn. 2018. “Digital Twin in manufacturing: A categorical literature review and classification”. *IFAC-PapersOnLine* vol. 51 (11), pp. 1016–1022.

Lamport, L. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc.

---

[3] The results are available at https://gitlab.au.dk/into-cps/rabbitmq-tla-test-results.

Leuschel, M., and M. Butler. 2005. "Automatic Refinement Checking for B". In *Formal Methods and Software Engineering*, edited by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, K.-K. Lau, and R. Banach, Volume 3785, pp. 345–359. Berlin, Heidelberg, Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

Li, W., F. Le Gall, and N. Spaseski. 2018. "A Survey on Model-Based Testing Tools for Test Case Generation". In *Tools and Methods of Program Analysis*, edited by V. Itsykson, A. Scedrov, and V. Zakharov, Volume 779, pp. 77–89. Cham, Springer International Publishing.

Schwarz, M. H., and J. Borcsok. 2013, October. "A survey on OPC and OPC-UA: About the standard, developments and investigations". In *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*, pp. 1–6. Sarajevo, Bosnia and Herzegovina, IEEE.

Thule, C., K. Lausdahl, C. Gomes, G. Meisl, and P. G. Larsen. 2019. "Maestro: The INTO-CPS Co-simulation Framework". *Simulation Modelling Practice and Theory* vol. 92 (April), pp. 45–61.

Toshev, M. 2015. *Learning RabbitMQ*. Packt Publishing Ltd. =.

FMI v. 2.0 2014. "Functional Mock-up Interface for Model Exchange and Co-Simulation".

Van der Auweraer, H., J. Anthonis, S. De Bruyne, and J. Leuridan. 2013. "Virtual engineering at work: the challenges for designing mechatronic products". *Engineering with Computers* vol. 29 (3), pp. 389–408. Publisher: Springer-Verlag.

Wayne, H. 2018. *Practical TLA+: Planning Driven Development*. Berkeley, CA, Apress.

Weyns, D. 2019. "Software Engineering of Self-adaptive Systems". In *Handbook of Software Engineering*, edited by S. Cha, R. N. Taylor, and K. Kang, pp. 399–443. Cham, Springer International Publishing.

Zheng, Y., S. Yang, and H. Cheng. 2019, March. "An application framework of digital twin and its case study". *J Ambient Intell Human Comput* vol. 10 (3), pp. 1141–1153.

Zhou, P., D. Zuo, K. Hou, Z. Zhang, J. Dong, J. Li, and H. Zhou. 2019, February. "A Comprehensive Technological Survey on the Dependable Self-Management CPS: From Self-Adaptive Architecture to Self-Management Strategies". *Sensors* vol. 19 (5), pp. 1033.

ACKNOWLEDGEMENTS

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins.

AUTHOR BIOGRAPHIES

CASPER THULE is a Post-Doc at the Centre for Digital Twins under DIGIT at Aarhus University and has an interest in tooling for co-simulation and digital twins. His email is casper.thule@eng.au.dk

CLÁUDIO GOMES is a Post-Doc at the Centre for Digital Twins under DIGIT at Aarhus University and focuses on numerical analysis within co-simulation and digital twins. His email address is claudio.gomes@eng.au.dk.

KENNETH G. LAUSDAHL is a researcher at the Centre for Digital Twins under DIGIT at Aarhus University. His research includes tool automation and language development for co-simulation and digital twins. His email address is kenneth@lausdahl.com.