Validating a Wireless Protocol Implementation at Binary Level through Simulation Using High Level Description of Protocol Properties in Light Esterel

Calypso Barnes*†, François Verdier†, Alain Pegatoquet†, Daniel Gaffé†, Jean-Marie Cottin*
*Électricité de France R&D, Chatou, France
Email: first_name.last_name@edf.fr
†Université Côte d'Azur, CNRS, LEAT, France
Email: first_name.last_name@unice.fr

***Abstract*—The development and debugging of a wireless protocol are complex tasks that many face in the industry and academia. This paper aims at facilitating those tasks by proposing a simulation framework that is capable of verifying and validating a protocol stack at binary level. This simulation framework is based on the co-simulation of QEMU and SystemC, which are interfaced through TLMu. An observer module was developed to analyze the traffic in the simulated network, which contains protocol properties modeled in Light Esterel to check that the frame exchanges comply with the protocol properties, in order to validate the protocol implementation.We describe the development of the simulation framework and its node models capable of executing the protocol's binary stack. We then explain the modeling of protocol properties in Light Esterel and their insertion in the simulation framework. Finally, we test the OCARI protocol for wireless sensor networks in the simulation framework.**

Keywords-Network Simulation; Co-Simulation; Protocol Verification; Wireless Sensor Networks; Embedded Software

I. INTRODUCTION

Wireless Sensor Networks (WSN) are an expanding technology employed in a wide and growing variety of applications. To communicate, they use a set of rules and conventions called protocol. Developing, debugging and deploying wireless sensor network protocols are complex tasks, especially when these protocols are aimed at applications requiring a high reliability, such as medical or industrial applications [1]. In such applications, a lost packet, a missed alarm or a sensor blocked due to interrupts can potentially lead to devastating consequences. That is why all wireless protocols, especially the ones aimed at more critical applications, have to be extensively tested and validated before they can be industrialized. While testing and debugging of WSN protocols heavily rely on testbeds, this phase can be facilitated by simulation.

In this paper we present a simulation framework for WSN based on QEMU (Quick EMUlator) [2] and SystemC co-simulation that aims at validating, verifying and debugging a wireless communication protocol by emulating the execution of the protocol's binary stack on a detailed model of the node's hardware. One of the main novelties of this framework is the addition of an observer module, containing protocol properties modeled with Light Esterel [3], that can detect whether a protocol property has been violated during simulation, halting the simulation to pinpoint the source of the bug thanks to the debugging capabilities provided. This approach will be tested on a protocol named OCARI (Open Communication protocol for Ad hoc Reliable industrial Instrumentation) [4] based on the IEEE 802.15.4 standard, which is currently in its pre-industrialization phase.

In Section II, we will explain the complexity in validating a protocol stack and the need to precisely model the hardware platform upon which the stack is executed. We will then describe in Section III the new simulation environment we developed including an observer module to verify protocol properties, and in Section IV we will explain how the protocol's properties are modeled in Light Esterel and inserted into simulation. Finally, in Section V, we will provide our simulation results that demonstrate the functionality of the simulation framework we developed. We then draw conclusions and discuss further improvements to be made on this simulation framework in Section VI.

II. THE COMPLEXITY OF WSN PROTOCOL VALIDATION

Protocols are geared to serve a number of needs and constraints, some of which are contradictory, such as tolerance to RF medium temporary disturbances, deterministic behaviors, a good link budget, power saving techniques and security. The resulting compromise is complex to implement and also to validate. It is highly likely that even after a communication protocol has been industrialized, some bugs might remain. According to [5], the average of bugs in released industrial software is about 15 to 50 per 1000 lines of delivered code. This is why finding new solutions for debugging and validation of code is of great interest to both industrialists and academia.

Validation heavily relies on human driven testing of the real platform and can take a very long time in the project. Testbeds are long to set up as each node has to be handled individually for software deployment and debugging. They also suffer from scenario limitations when it comes to the number of nodes in the network and their mobility or the environmental conditions (obstacles, interference, etc.). That is why simulation is an important step in the development of wireless protocols, to better diagnose software problems before the deployment on real hardware.

Simulation can offer better flexibility on environmental conditions, as well as the possibility to test attacks on the network more easily, to validate the security aspects of the protocol. Another one of the great benefits of validating a stack on a simulator is that the scenario execution can be replayed exactly, while usual hardware test-beds cannot truly reproduce the same run, due to the asynchronous nature of the system: the CPUs of network nodes do not share the same clock and even show clock drifts. Moreover, hardware debugging is intrusive as it modifies the execution timing of the software being inspected.

Validation through simulation of a protocol at behavioral level or at C code level might be faster than validation at binary code level, but it is less reliable. Indeed, validating the source code or a behavioral model of the protocol only provides a very crude notion of time [6]. Therefore, it cannot precisely model interrupts and multitasking for instance. Yet temporary errors that are hard to reproduce in hardware validation may have their origin in wrong timer programming, missed interrupts or interrupts taken too late leading to task scheduling issues, buffer overflows and packet loss. That is why validating the binary code on a realistic platform model is a much more reliable approach.

Moreover, a simulator capable of executing the protocol's binary code has the advantage of being compatible with testing protocols for which only the binary code is provided without the source files, to test that the specifications of this protocol are met. This is why we focused on simulators capable of executing the protocols stacks binary code for our purpose of protocol validation. Only a few existing network simulators offer this advantage. To our knowledge, these simulators are TOSSIM, ATEMU, AVRORA, COOJA and Worldsens (WSim/WSNet) [7]. However, they all suffer from a few limitations, mainly in terms of the type of node platform modeled. Indeed, Atemu emulates MICA-2 motes, Cooja associated to MSPSIM emulates nodes based on the Texas Instruments MSP430 processor, and Worldsens is also limited to models of the MSP430 processor based nodes. Moreover, Avrora does not model mobility or clock drift and Tossim is not capable of capturing properties related to interruptions or the codes execution time [8]. This is the reason why we decided to develop a new simulation framework that could be more easily adapted to a larger variety of hardware platforms. To ensure the durability of this framework, we focused on open source solutions.

III. Modeling a precise node hardware model with QEMU and SystemC

Our hardware platform model was developed with QEMU and SystemC. We decided to use TLMu as the interface between both simulation environments because it is more easily adaptable to our platform model.

A. *TLMu - An Interface between QEMU and SystemC*

Based on our need to execute the binary code of protocols for a wide range of platform, we have decided to create a simulation environment based on QEMU (Quick EMUlator) [2], a popular open source environment developed in C for the rapid prototyping of virtual platforms. We have also decided to associate QEMU with SystemC, a popular C++ library to model the architecture and behavior of hardware components, to model parts of the hardware platforms absent in QEMU and the network.

TLMu [9] is a wrapper for QEMU that integrates with SystemC TLM-2.0 models, which handles the communications and time synchronization between both simulation cores. It is based on the open source hardware/software emulation framework called QEMU-SystemC [10], which was proposed in 2007 by Màrius Montón and GreenSoCs. This project aimed at attaching devices modeled in SystemC to QEMU, with QEMU as the simulation master and SystemC as the simulation slave.

An advantage of TLMu is the possibility to have multiple QEMU platforms connected to SystemC in the same simulation. It can provide either only the CPU cores or a more complete system, and the CPU emulators are built as shared libraries, one library per supported architecture (e.g., libtlmu-arm.so). Each CPU executes as a different thread in the same process as SystemC.

1) Communications between QEMU and SystemC: The wrapper loads QEMU as a shared object and accesses QEMU's main function by a loaded function pointer from the shared object. The wrapper defines callback functions. In QEMU, the SystemC environment is registered as a memory region, which when accessed uses a callback to access the TLM memory bus model using TLM transactions to read or write memory spaces in SystemC. This allows switching the control between QEMU and SystemC. A callback function also handles the communications of events to QEMU, such as interrupts from SystemC to QEMU, or reset, sleep and wake up commands.

2) Handling time and synchronization: TLMu is run by using QEMU's icount feature which means that QEMU's virtual clock (vm_clock) advances according to the number of instructions executed. In SystemC, time progresses according to the information provided by the hardware platform. TLMu may trigger a synchronization between QEMU and SystemC in various cases: when the CPU makes I/O accesses into the main emulator, when the CPU gets interrupted or after the execution of a translation block. To trigger a synchronization, QEMU calls a callback sync_time function of TLMu, and passes as a function parameter its vm_clock time. If the sync_time function estimates that enough time has passed since the last synchronization, it triggers a new synchronization. The time between synchronizations is handled by the TLM quantum keeper [11].

B. Modeling the node's hardware platform and the network

The binary stack to be tested implements the OCARI protocol on an Atmel SAM3S platform based on the ARM Cortex-M3 processor and an IEEE 802.15.4 radio module, the Atmel AT86RF233, which are connected through an SPI interface, as well as three pins for interrupt notification, reset, and sleep/wakeup command.

To develop a model of the SAM3S platform, we reused the Cortex-M3 CPU model that is available in QEMU. The memories and different peripherals of the SAM3S (timers, power management controller, SPI, etc.) were modeled in the SystemC environment according to their behavioral description, which is found in the SAM3S datasheet provided by Atmel [12]. Only the peripherals that are enabled by the Power Management Controller during the execution of OCARI's binary code, and therefore useful to the execution of OCARI's protocol, are modeled, as seen in Figure 1.

The peripherals are connected through a TLM bus and communicate through loosely timed blocking TLM transactions. These transactions are objects that are defined by TLM 2.0 containing amongst other a data pointer, a data length and an address. They are sent between different SystemC modules from TLM initiator sockets to TLM target sockets. At the reception of a transaction in a target socket, a `b_transport` function is called defining what action to take.

The interrupt mechanism was not fully implemented by TLMu, though callbacks were provided as an interface between SystemC and QEMU. In our implementation, we reused the model of the Nested Vectored Interrupt Controller (NVIC) for the ARM Cortex-M3 that was already available in QEMU to complete the interrupt mechanism.

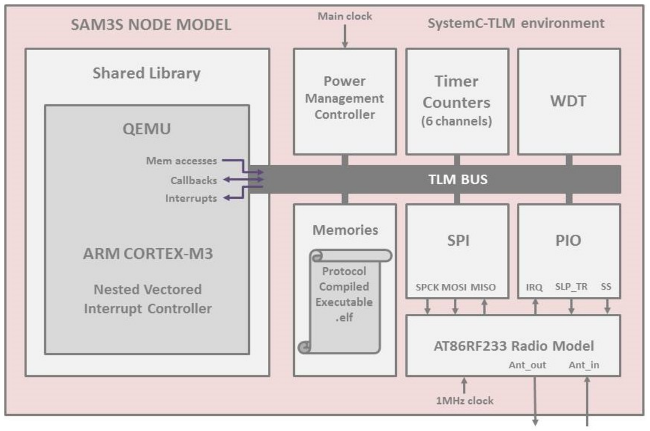


Figure 1. Simplified block diagram of the SAM3S platform modeled with SystemC and QEMU interfaced by TLMu.

The radio was also modeled in SystemC according to the AT86RF233 datasheet. The protocol controls the radio module through an SPI interface to read or write the radio registers or the frame buffer. A PIO pin is also used so that the radio can send back interrupts to the SAM3S platform. Frames that are sent out are sent as TLM transactions through the Antenna_out TLM socket. The frame is modeled as an array containing the bytes corresponding to the frame length and the Mac Protocol Data Unit (MPDU). Incoming frames are received through the Antenna_in TLM socket to later be interpreted by the protocol stack if the frame passes the radio filter.

It was verified that the model of the SAM3S and the radio were functionally correct by comparing frames output by the simulator to frames from real exchanges between nodes in a lab for similar scenarios [13]. It is possible to instantiate several models of the SAM3S platform modeled with QEMU, as well as more simple nodes that do not execute the binary code of a protocol but mimic the protocol's behavior.

In our simulation framework, all the node models are connected through a radio link module as shown in Figure 2. The radio link module relays the frames from the sender node to the other nodes according to the topology defined by the user. The user can also define times when new nodes will be added to the topology or times when a radio link between two nodes will be lost or created.

The radio link module also contains another module called the observer, which analyzes the frames exchanged as described in Section IV.

To be able to observe in an aesthetic manner the frames exchanged by the different nodes in simulation, the frames that go through the radio link module are saved in a pcap (packet capture) format file by using functions from the libpcap library to create packet headers and to encapsulate each packet. The frames in the pcap file can then be observed in a network analyzer, such as Wireshark for example. Using the same libpcap library, the radio link module is also able to extract frame packets from pcap files to generate traffic in the network, with the possibility of using frames recorded from exchanges between nodes in a lab.

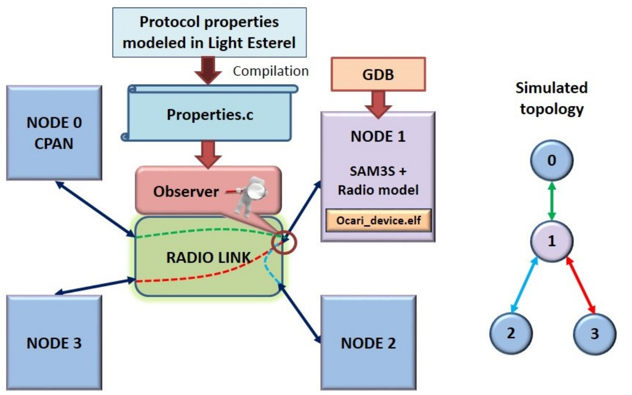


Figure 2. Connection between node models through a radio link module containing the observer focused here on node 1.

IV. Verifying Protocol Properties during Simulation

The verification of protocol properties is an important feature of the simulation framework. To implement this feature, we have developed an observer module.

A. The observer's role

The observer module is integrated into the simulator to analyze the frames exchanged between nodes. To have the capability of analyzing any frame that is exchanged in the network, the observer module is contained in the radio link module. Its role is to halt simulation if a protocol property is violated, so that the source of the bug causing the violation may be pinpointed thanks to the debugging capabilities provided by the simulation framework. The observer focuses on the frames sent and received from a specific node which is running the binary protocol stack under test.

B. Modeling and integrating protocol properties into simulation

We searched for a good solution to model protocol properties that could be easily introduced into our simulation environment. A communication protocol is considered to be a reactive system, and general purpose programming languages are not suited to design reactive systems: they are clearly inefficient to deal with the inherent complexity of such systems [14]. That is why we looked into using a language dedicated to reactive systems. Various synchronous languages have been designed such as Esterel [15]. Based on concurrency and synchronicity, they are model-based languages to allow formal verification of the system behavior. Their execution model is simple: first, the initial state is initialized and then, for each input event set, outputs are computed and then state is updated [14].

1) Light Esterel: Light Esterel (LE) [3] is a reactive synchronous programming language derived from Esterel V5 [15]. Just like Esterel, it is able to maintain a permanent interaction with its environment, such as communication protocols, man-machine interface drivers or VLSI chips.

The Light Esterel language units are named modules. Communication takes place between modules or between a module and its environment. Sub-systems communicates via events. The module interface declares the set of input events it reacts to and the set of output events it emits.

The textual or graphical Light Esterel views gives the possibility to write compact specifications for very complex systems. A system with thousands of states can generally be specified by an Esterel program of only a few hundred lines thanks to the explicit preemption and parallel operator. Moreover, this deterministic concurrent programming language can be compiled into a Finite State Machine (FSM) classically represented by an automaton. LE automata are Mealy machines and they have a set of input signals, which can be valued, to define transition triggers and a set of output signals that can be emitted when a transition is raised.

Consequently, Light Esterel programs can also be compiled into popular languages such as C, VHDL, CSharp, or other synchronous languages like Lustre. In our case we used the C program compilation of Light Esterel.

As Light Esterel has been designed to model reactive systems such as communication protocols, and because of the ease of insertion into the simulation framework, we decided to use this language to model protocol properties defined by the protocol's specifications.

2) Integration of protocol properties in the simulation framework: The objective of the observer module is to verify that the implementation of a protocol complies with its specified properties. The properties that we want to check are modeled in a LE program.

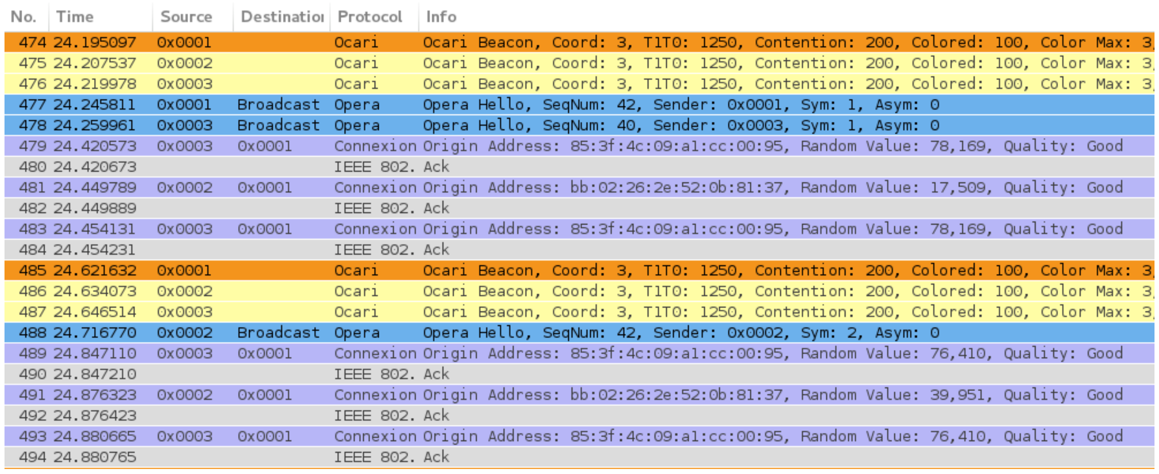
When the Light Esterel program is compiled into C to be integrated in the simulator, functions are defined where the function parameters are the inputs of the LE program and pointers to the output values.

The observer dissects the frames sent and received from a specific node that is running the binary protocol stack under test. If the information contained in those frames are relevant to the modeled protocol property, the corresponding Light Esterel C program function is called with the corresponding boolean presence predicat set to 1, followed by the signal value if the signal is valued. After the function call, the Light Esterel program outputs are checked to see whether the property has been violated, or if the conditions of the property have been met. The latter information helps in statistics on property coverage. Indeed, in certain scenarios, some properties might not be checked at all while other properties might be checked numerous times. The simulation scenarios have to be adapted in this case to make sure that all properties have been extensively put under test.

V. Experimental Results and interpretation

To demonstrate the functionality of the simulator we developed, we tested it with the OCARI protocol, to verify that the implementation of the protocol complies with the properties from its specifications. Some simple properties that we desired to check were modeled in LE, compiled into C, and inserted into the simulator. We ran different scenarios to check that the protocol properties we modeled weren't detected as violated by the simulator. Here we describe a particular scenario where an attack on the network was simulated to see how the nodes running the OCARI protocol binary stack reacted.

In the scenario in question, 3 nodes running the OCARI protocol on full SAM3S platform models are instantiated, one is the network coordinator (CPAN) loaded with the ocari_CPAN.elf binary code, and the other two nodes are running the ocari_DEVICE.elf binary code. A fourth node is added to the simulation that will act as a node attacking



```
No.  Time       Source  Destinatio Protocol  Info
474 24.195097  0x0001            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
475 24.207537  0x0002            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
476 24.219978  0x0003            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
477 24.245811  0x0001  Broadcast Opera     Opera Hello, SeqNum: 42, Sender: 0x0001, Sym: 1, Asym: 0
478 24.259961  0x0003  Broadcast Opera     Opera Hello, SeqNum: 40, Sender: 0x0003, Sym: 1, Asym: 0
479 24.420573  0x0003  0x0001    Connexion Origin Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 78,169, Quality: Good
480 24.420673                    IEEE 802. Ack
481 24.449789  0x0002  0x0001    Connexion Origin Address: bb:02:26:2e:52:0b:81:37, Random Value: 17,509, Quality: Good
482 24.449889                    IEEE 802. Ack
483 24.454131  0x0003  0x0001    Connexion Origin Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 78,169, Quality: Good
484 24.454231                    IEEE 802. Ack
485 24.621632  0x0001            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
486 24.634073  0x0002            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
487 24.646514  0x0003            Ocari     Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3,
488 24.716770  0x0002  Broadcast Opera     Opera Hello, SeqNum: 42, Sender: 0x0002, Sym: 2, Asym: 0
489 24.847110  0x0003  0x0001    Connexion Origin Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 76,410, Quality: Good
490 24.847210                    IEEE 802. Ack
491 24.876323  0x0002  0x0001    Connexion Origin Address: bb:02:26:2e:52:0b:81:37, Random Value: 39,951, Quality: Good
492 24.876423                    IEEE 802. Ack
493 24.880665  0x0003  0x0001    Connexion Origin Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 76,410, Quality: Good
494 24.880765                    IEEE 802. Ack
```

Figure 3. Simulation output frame traces from 2 regular OCARI protocol cycles viewed with Wireshark.

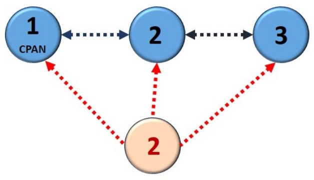


Figure 4. Network topology.

the network. The topology of the network can be seen in Figure 4.

The simulation was set up so that the attacker node is a simple node that copies a frame from the beginning of the simulation, more specifically a Beacon type frame from node 2. After 25 seconds of simulation, the attacker nodes starts sending the Beacon frame copied from node 2 at constant intervals of 50 ms. The fake beacon is received by all three other nodes in the network.

The simulation results can be observed through the terminal where the simulation was launched, but the frame exchanges can also be viewed through Wireshark, as they are saved in a pcap format. A plugin was created in Wireshark to dissect OCARI frames to correctly display frame information. Before the intervention of the attacker node, no anomaly is detected in the exchanges. In Figure 3, two cycles can be seen (a cycle beginning with a beacon frame from the CPAN) before the attacker node starts sending frames, with nodes 3 and 2 correctly relaying data to the CPAN, which are Connexion type frames with random data values for this particular application.

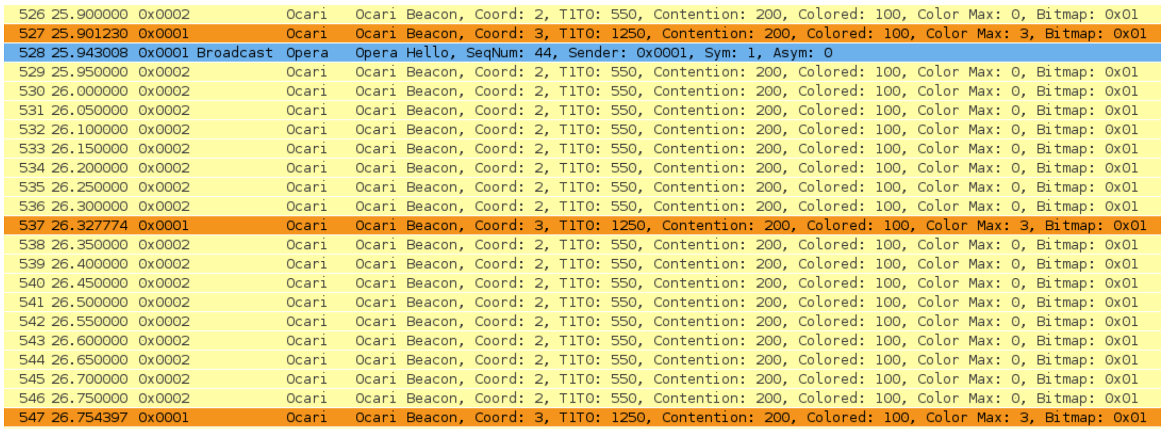
25 seconds into simulation, the attacker node starts sending Beacon frames pretending to be node 2. The simulation was stopped at 26.754397 seconds, as the observer in the simulator detected and indicated a ”Hello” property violation from node 2. The property in question states that once nodes are associated to the CPAN, a Hello message must be sent every 2 cycles. In the frame traces observed on Wireshark, we can see that nodes 2 and 3 completely stop sending frames after 25.55445 seconds of simulation time. In Figure 5, we can observe the two last cycles before the simulation stopped where no Hello messages were sent by node 2 and 3, causing the Hello property to be violated. The violation is detected as the last Beacon frame from the CPAN is received indicating the beginning of a new cycle without a Hello frame having been received.

To know what went wrong with node 2, and why it stopped sending frames, we used GDB to know which function of the ocari_DEVICE.elf code was being executed by node 2 when the simulation stopped. We have observed that node 2 was stuck in a while loop waiting for a timer to expire, and that the while loop was initiated by the mac_wrapper_reset_MaCARI function, indicating that the node couldn’t handle the frames sent by the attacker node and had to reset itself.

This experiment is an example of how this simulation environment facilitates testing protocols under more flexible scenarios and environmental conditions. Indeed, adding such an attacker node in simulation was relatively easy and would have been more complicated in a lab environment. The version of the OCARI protocol has been updated since this test with enhanced security. Not all protocol properties were inserted in simulation for these tests, only a few that we wished to check. Extra properties will be modeled and added into simulation in the future, in order to validate them under very diverse scenarios to explore the limits of the protocol.

VI. Conclusion and Perspectives

In this paper, we presented a simulation framework based on QEMU/SystemC co-simulation and Light Esterel property modeling for the validation of wireless sensor network protocol implementations against their specified properties. We also showed the possibility of simulating an attack on



```
526 25.900000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
527 25.901230 0x0001           Ocari    Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01
528 25.943008 0x0001 Broadcast Opera    Opera Hello, SeqNum: 44, Sender: 0x0001, Sym: 1, Asym: 0
529 25.950000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
530 26.000000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
531 26.050000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
532 26.100000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
533 26.150000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
534 26.200000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
535 26.250000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
536 26.300000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
537 26.327774 0x0001           Ocari    Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01
538 26.350000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
539 26.400000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
540 26.450000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
541 26.500000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
542 26.550000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
543 26.600000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
544 26.650000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
545 26.700000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
546 26.750000 0x0002           Ocari    Ocari Beacon, Coord: 2, T1T0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
547 26.754397 0x0001           Ocari    Ocari Beacon, Coord: 3, T1T0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01
```

Figure 5. Simulation traces from the two last OCARI protocol cycles before the property violation viewed with Wireshark.

the network to find weaknesses in the protocol and fix them. This simulation framework still needs some more development but shows promising results.

Future improvements on the simulation framework will include random test generation and the possibility to reproduce scenarios played out on testbeds with real nodes using the exchange traces in pcap format. There will be improvements on the radio link module to represent more diverse environmental conditions and packet loss. To obtain a better scalability, we also wish to introduce a frame generator that imitates the behavior of several nodes running the protocol under test, in order to stimulate one node that is running the protocol's binary code on a full platform model.

Verifying that the execution of the protocol respects all of its defined properties in a wide range of scenario will prove that the protocol is reliable, and will facilitate its industrialization. With this simulation framework, the great number of hardware platforms supported by the open source tools used extends the approach to a wide range of protocols for all kinds of networked embedded systems.

References

[1] P. Suriyachai, U. Roedig, and A. Scott, "A survey of mac protocols for mission-critical applications in wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 2, pp. 240–264, 2012.

[2] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[3] D. Gaffé and A. Ressouche, "Algebraic framework for synchronous language semantics," in *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*. IEEE, 2013, pp. 51–58.

[4] K. Al Agha, M.-H. Bertin, T. Dang, A. Guitton, P. Minet, T. Val, and J.-B. Viollet, "Which wireless technology for industrial wireless sensor networks? The development of OCARI technology," *Industrial Electronics, IEEE Transactions on*, vol. 56, no. 10, pp. 4266–4278, 2009.

[5] S. McConnell, *Code complete*. Pearson Education, 2004.

[6] T. Chang, T. Watteyne, K. Pister, and Q. Wang, "Adaptive synchronization in multi-hop TSCH networks," *Computer Networks*, vol. 76, pp. 165–176, 2015.

[7] B. Musznicki and P. Zwierzykowski, "Survey of simulators for wireless sensor networks," *International Journal of Grid and Distributed Computing*, vol. 5, no. 3, pp. 23–50, 2012.

[8] A. Fraboulet, G. Chelius, and E. Fleury, "Worldsens: development and prototyping tools for application specific wireless sensors networks," in *Information Processing in Sensor Networks. 6th International Symposium on*. IEEE, 2007, pp. 176–185.

[9] "Transaction level eMulator (TLMu)," 2011, edgarigl.github.io/tlmu/. Accessed on Sep. 27, 2016.

[10] M. Montón, J. Engblom, and M. Burton, "Checkpointing for virtual platforms and SystemC-TLM," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 1, pp. 133–141, 2013.

[11] "Tlm 2.0 quantum keeper," 2010, www.embecosm.com/appnotes/ean1/html/ch09s01s03.html. Accessed on Sep. 27, 2016.

[12] "Sam3s arm cortex-m3 microcontroller," 2015, www.atmel.com/products/microcontrollers/arm/sam3s.aspx. Accessed on Sep. 27, 2016.

[13] C. Barnes, J.-M. Cottin, F. Verdier, and A. Pegatoquet, "Towards the verification of industrial communication protocols through a simulation environment based on qemu and systemc," in *ACM/IEEE 19th International Conference on Models Driven Engineering Languages and Systems (MODELS 2016)*, 2016.

[14] A. Ressouche, D. Gaffé, and V. Roy, "Modular compilation of a synchronous language," in *Software Engineering Research, Management and Applications*. Springer, 2008, pp. 157–171.

[15] G. Berry, *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.