POLITECNICO DI TORINO
Repository ISTITUZIONALE

SystemC Model Generation for Realistic Simulation of Networked Embedded Systems

Original
SystemC Model Generation for Realistic Simulation of Networked Embedded Systems / Lazarescu, M.T., Sayyah, P., Quaglia, D., Stefanni, F.. - ELETTRONICO. - (2012), pp. 423-426. (15th Euromicro Conference on Digital System Design Izmir, Turkey September 2012) [10.1109/DSD.2012.123].

Availability:
This version is available at: 11583/2507482 since: 2020-07-05T15:23:21Z

Publisher:
IEEE

Published
DOI:10.1109/DSD.2012.123

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

14 October 2026

SystemC model generation for realistic simulation of networked embedded systems

Mihai Lazarescu, Parinaz Sayyah
Department of Electrical Engineering,
Politecnico di Torino, Italy
firstname.lastname@polito.it

Davide Quaglia, Francesco Stefanni
EDALab s.r.l., Verona, Italy
firstname.lastname@edalab.it

***Abstract*—Verification and design-space exploration of today's embedded systems require the simulation of heterogeneous aspects of the system, i.e., software, hardware, communications. This work shows the use of SystemC to simulate a model-driven specification of the behavior of a networked embedded system together with a complete network scenario consisting of the radio channel, the IEEE 802.15.4 protocol for wireless personal area networks and concurrent traffic sharing the medium. The paper describes the main issues addressed to generate SystemC modules from Matlab/Stateflow descriptions and to integrate them in a complete network scenario. Simulation results on a healthcare wireless sensor network show the validity of the approach.**

I. Introduction

The design of today's embedded systems is becoming ever more complex due to the need to cover a large set of heterogeneous domains, such as digital (hardware and software), analog (hardware, MEMS, power sources, external environment) and network (communications with other systems). In this paper we focus on a realistic complex wireless sensor network application for health monitoring designed in the COMPLEX European project [1]. Each person wears a wireless node which senses different kinds of body information, analyses them and communicates with a base station on a shared radio channel which hosts also other concurrent traffic. Nodes consist of non-trivial software components running over a 32-bit system-on-chip architecture by ST-Microelectronics. To simplify application development, a model-driven approach has been chosen and software components are specified by using implementation-independent languages such as UML/MARTE and Stateflow.

To allow verification and design-space exploration of the platforms supporting this kind of applications, we need to simulate the following aspects:

- *system behavior*, modeled in an implementation-independent fashion, e.g., as a set of interacting state charts instead of using CPUs, memories, hardware accelerators;
- *network scenario*, consisting of channel behavior (e.g., path loss, collisions), communication protocols (e.g., IEEE 802.15.4), concurrent traffic and noise.

Research activity partially supported by the European project COMPLEX FP7-2007-IST-4-247999

A possible way to simulate all these aspects is co-simulation [2], i.e., the combined and synchronized execution of different simulation tools, e.g., Stateflow for the state charts of the application, VHDL for hardware components, NS-2 [3] for the network. This solution is time-consuming and, for this reason, in the COMPLEX project we decided to model all these aspects with the same language, namely SystemC, which has a great flexibility and extensibility.

To enable this methodology, the *generation and integration of SystemC models* from the different aspects of the application is crucial. In particular, this paper describes:

- the generation of SystemC modules from Stateflow specifications;
- the creation of network scenarios in SystemC by using the SystemC Network Simulation Library [4];
- the integration of the resulting modules in a comprehensive simulation scenario.

The paper is organized as follows. Section II describes the generation of SystemC models from Stateflow. Section III describes how SystemC is used to model a network scenario. Section IV introduces the case study. Section V describes the model integration into the SystemC platform while Section VI reports some results about communication performance.

II. Generation of SystemC from Stateflow

To implement the generation flow we extended HIFSuite tool [5] which provides an intermediate format, named Heterogeneous intermediate Format (HIF), and a set of functions to manipulate it. Moreover, the HIF language has a well-defined formal semantics, borrowed from the UNIVERCM [6] computational model. Therefore, the mapping is correct-by-construction.

Nevertheless, Stateflow state charts and HIF descriptions have some semantic differences, which have to be addressed during conversion:

1) Stateflow has hierarchical states and allows transitions between different hierarchical levels while HIF provides only flat automata.
2) Stateflow allows actions both on transitions and into states while HIF allows only actions on transitions.
3) Stateflow provides the so-called *condition actions* and *transition actions*, leading to complex semantics

behaviors. HIF provides only actions associated to transitions, which has a different semantic with respect to Stateflow actions.

4) Stateflow provides the so-called *junctions*, with associated transitions that have different semantics from transitions between states. HIF provides only transitions between states.
5) Stateflow provides events, whose broadcasting can lead to unexpected behaviors, when they are raised within states or as condition actions. HIF has labels, that can be raised only on transitions, and that have a very strict semantics.

Since HIF supports only flat EFSMs, the first step of Stateflow to SystemC conversion consists in removing state hierarchy by using a recursive algorithm.

Then, each HIF automaton will be mapped into a SystemC process. This mapping introduces a difference with respect to the original Stateflow behavior: on one hand, in Stateflow parallel states are executed in a pre-defined order thus avoiding explicit synchronization mechanisms for reading/writing data shared among different automata. On the other hand, SystemC does not have such a guarantee, and thus it requires explicit synchronization.

Stateflow junctions are a kind of "soft states" with a backtrack semantics according to which a set of consecutive junctions cannot be traversed until the destination state is reached. This backtrack behavior has been reproduced in our SystemC translation by exploiting the similar behavior of C++ function calls. Each junction and transition involving a junction generates a method call in SystemC. At the end of recursive calls, the last called function returns a flag to check whether a state has been successfully reached.

Stateflow provides two kinds of actions related to a transition, i.e., *condition action* and *transition action*. A transition action is executed only when a destination state is reached, while a condition action is executed as soon as the condition becomes valid. These kinds of actions create complex interactions, especially in case of events. To avoid such complex interactions, we decided to put some restrictions in the generation of SystemC as follows:

- Transition actions are supported, since their semantics matches the traditional semantics of EFSMs. In case of transition actions involving junctions, these actions will be "moved" to the last transition thus guaranteeing their execution only in case a valid state is reached.
- Condition actions are supported only if they do not raise events. Note that even the Stateflow documentation suggests to avoid this kind of behavior, since the graphical representation does not match the actual simulation behavior.

III. Modeling a network scenario with SystemC

The *SystemC Network Simulation Library* (SCNSL), freely available at [4], extends the SystemC simulation capability to network-oriented scenarios.

SCNSL *tasks* contain the application functionality which is being designed. From the point of view of a network simulator, a task is just the producer or consumer of packets and therefore its implementation is not important. However, for the system designer, task implementation is crucial and many operations are connected to its modeling, e.g., choice of abstraction level, validation, fault injection, HW/SW partitioning, mapping to an available platform, synthesis, and so forth. For this reason *TaskProxy* instances has been introduced to decouple task implementation from the backend which simulates the network.

Nodes are containers of tasks and provide access to the channel.

Channels are an abstraction of the transmission medium, and thus they simulate various communication aspects, like packet collisions. SCNSL provides models for both wired (full-duplex, half-duplex and unidirectional) and wireless channels.

Designers can either implement protocols ex-novo in custom tasks or rely on protocol models provided by the simulator through optional backend components named *Communicators*. Communicators can be interconnected with each other to create chains modeling protocol stacks as well as buffers to store packets, simulation tracing facilities and so forth.

Another critical point in the design of the tool has been the concept of *Packet*. Generally, the packet format depends on the corresponding protocol even if some header fields are almost always present, e.g., packet length and source/destination addresses. System design requires a bit-accurate description of the packet content, to test parsing functionality, while from the point of view of the network simulator the strictly required fields are the length (for bitrate computation) and some flags to mark collisions (if routing is performed by the simulator, source/destination addresses are also needed). To meet these opposite requirements in SCNSL, an internal packet format is used by the simulator, while the system designer can use different packet formats according to protocol design. The conversion between the user packet format and the internal packet format is performed in the TaskProxy.

IV. Case study: SPINE application

In this section, we describe a realistic Wireless Sensor Network (WSN) application, from the health monitoring domain. It couples all major aspects of WSNs (sensing, processing, radio transmission) with a non-trivial amount of local computation on the raw sensed data to reduce the total amount of energy consumed for transmission. This requires the use of a powerful 32-bit processor and optimization of the node HW/SW platform according to the goals of the COMPLEX project.

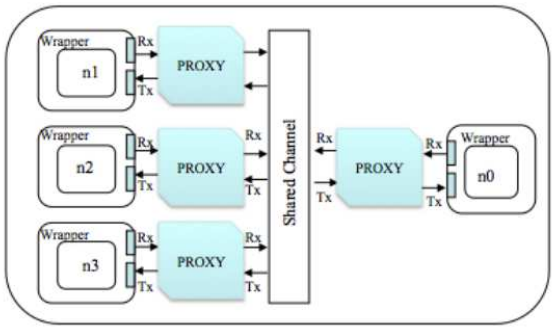

Figure 2. Top view of SPINE network scenario in SCNSL.

The application is a health monitoring-oriented virtual machine that conforms to the SPINE (Signal Processing in Node Environment [7]) specification. It provides the application developer with a set of functions like:

1) sensing various kinds of data (e.g. temperature, acceleration, blood oxigenation level, hearth pulse rate) and storing them inside local buffers;
2) performing a variety of filtering, threshold detection, and other mathematical functions over the buffers;
3) sending data packets to a base station either regularly, or when some interesting or emergency event occurs.

All the functions described above are dynamically triggered by the Base Station (BS) node, by sending configuration packets to the SPINE node. Examples of configuration parameters include the sampling period of individual sensors, the buffer sizes, the features (e.g., max, mean, median) that need to be computed. The last configuration packet contains the list of tasks (sampling, feature computation, radio packet dispatch) that must be activated. The SPINE nodes process configuration packets from the BS, perform the configured computations and transmit the results to the BS.

As shown in Figure 1, the SPINE behavior has been modeled in Stateflow by using three concurrent state charts, namely: *SpineSchedulerEngine*, *SpinePktProcessingEngine*, *Timer*. The *SpinePktProcessingEngine* is responsible for processing incoming packets (represented in Stateflow as PKT events), from the BS. The *Timer* state machine is in charge of activating each task with the appropriate period. It keeps a queue of task activation times, and wakes up the *SpineSchedulerEngine* whenever needed. Finally, the *SpineSchedulerEngine* executes the actual tasks. i.e., sampling and storing sensor data, computing signal processing functions, and sending data to the BS.

V. Interfacing SystemC code with SCNSL

This section illustrates how the SystemC code generated from the Stateflow model is integrated into the SCNSL simulation environment.

Table I
Network performance varying the number of nodes

Number of nodes	Transmission delay (*ms*)	Delivery ratio	Maximum backoffs hits	Average backoff (*ms*)
2	7.5	1.0	0	2
4	14	0.98	20	3.5
6	18	0.9	400	4.5
8	20	0.8	800	4.7

The top view of the experimental scenario is shown in Figure 2, where n0 represents the BS, and n1, n2 and n3 are the SPINE nodes. Figure 3 shows the architecture of the `SPINE_t` class which includes two `SC_THREAD`, namely `spineWakeupCLK` and `spine_TX_Pkt`. The `spineWakeupCLK` thread provides the basic timing functionality to the SPINE model. It uses the `wait` method to call `wakeup.notify`, which in turn calls the main SPINE model with the CLK periodic input event used by the `Timer` process. The `spine_TX_Pkt` thread is sensitive to the `sendPacket` event and thus it remains sleeping until the SPINE model has a data packet to send back to the BS. At this time, the SPINE model executes the `sendDataPkt.notify` method, which wakes up `spine_TX_Pkt` to start the packet transmission on the radio channel. The BS class is similar (only the application code differs).

VI. Experimental results

The IEEE 802.15.4 protocol is configured to operate in unslotted CSMA/CA mode and 16 bit short addresses are used in data packets. The communication is synchronized and controlled by the Base Station which starts by sending three configuration packets followed by the start command packet in a short interval. After receiving the start command, each SPINE node samples the accelerometers every 40 ms and sends a 64-byte data packet to the Base Station. The simulation is carried out for 20 seconds of simulated time (i.e. about 500 packets per node) and considers up to eight SPINE nodes, which may be placed, for example, on different people in the same room.

According to the IEEE 802.15.4 protocol, each SPINE node waits for a random number of backoff units (320 μs) within the $[0, 2^{BE} - 1]$ range before accessing the channel, where the backoff exponent *BE* is initially set to 3 and it is increased up to 7 if the channel is still found busy. Transmission is retried up to 4 times.

Table I shows the key performance numbers obtained from the network simulation, as a function of the total number of SPINE nodes in the network (excluding the base station). The average transmission delay increases and the packet delivery ratio decreases, as expected. Transmission delay increases due to the increase in retransmissions and backoff time.

VII. Conclusions

A methodology for the simulation of a wireless sensor node has been presented and validated. We described how

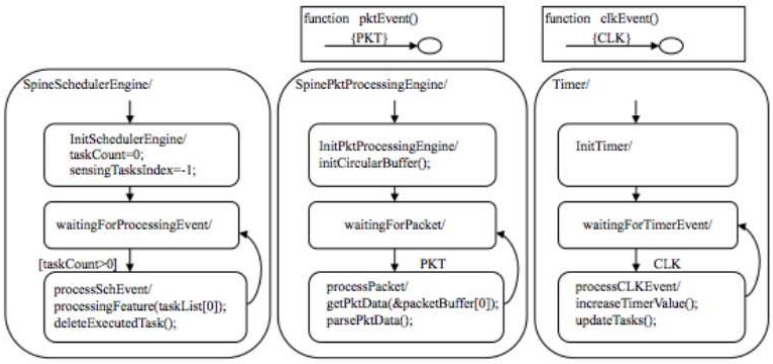

Figure 1. SPINE behavior described in Stateflow as three concurrent state charts.

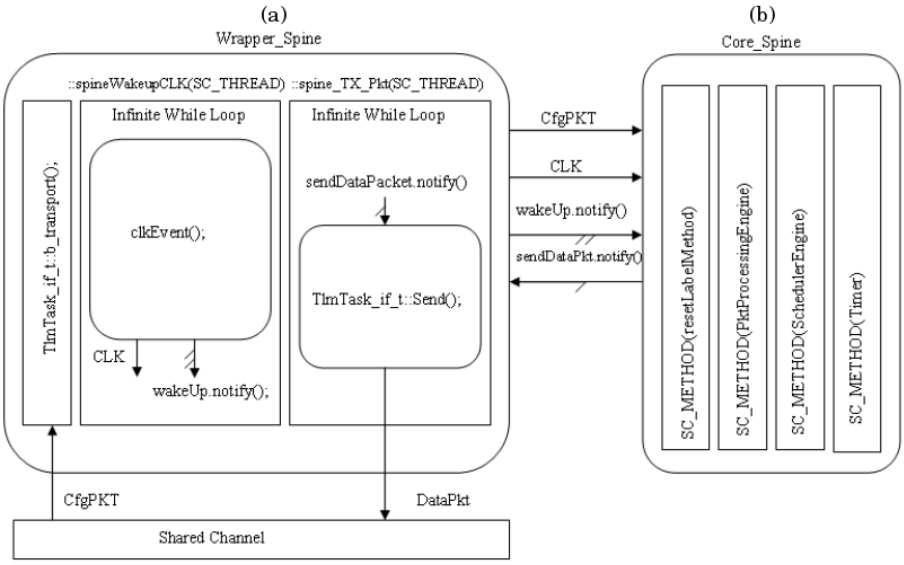

Figure 3. Architecture of the SCNSL model of the SPINE node

the node behavior can be modeled by using Stateflow and how HIFSuite can be used to transform it into a multi-process SystemC model. The generated SystemC model can be easily incorporated into a SystemC-based network simulator, called SCNSL, in order to explore aspects such as network latency under varying channel and traffic conditions. Future work will address the incorporation of a SystemC instruction-set simulator and a cycle-accurate power model of the hardware platform, to explore the impact of network-aware power management strategies on the battery lifetime.

REFERENCES

[1] European Commission, "COdesign and power Management in PLatform-based design space EXploration - COMPLEX," *URL: https://complex.offis.de/*, no. FP7-IST-247999, 2009.

[2] M. Chung and C.-M. Kyung, "Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 125–136, Feb. 2006.

[3] S. McCanne and S. Floyd, "NS Network Simulator – version 2," *URL: http://www.isi.edu/nsnam/ns*.

[4] "SystemC Network Simulation Library – version 1," 2008, URL: http://sourceforge.net/projects/scnsl.

[5] EDAlab, "HIFSuite home page," http://www.hifsuite.com/.

[6] L. Di Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, and S. Vinco, "UNIVERCM: The UNIversal VERsatile computational model for heterogeneous embedded system design," in *High Level Design Validation and Test Workshop (HLDVT), 2011 IEEE International*, nov. 2011, pp. 33 –40.

[7] Signal Processing In Node Environment, "SPINE home page," http://spine.tilab.com/.