HybridSim: A Modeling and Co-simulation Toolchain for Cyber-Physical Systems

Baobing Wang and John S. Baras
Department of Electrical and Computer Engineering
and the Institute for Systems Research
University of Maryland, College Park, MD 20742, USA
{briankw, baras}@umd.edu

***Abstract*—Cyber-physical systems (CPS) involve communication networks, computation algorithms, control systems and physical systems. Many CPS, such as Smart Buildings, are subject to very expensive deployment costs and complex network interactions. Thus comprehensive modeling and simulation of such systems are crucial to ensure that they function as intended before deployment. Given the multi-domain nature of CPS, it is more appropriate to use a heterogeneous simulation environment to study system dynamics. In this paper, we design and implement an integrated modeling and co-simulation toolchain, called HybridSim, for the design and simulation of CPS. Firstly, HybridSim can transform and import existing system components from multi-domains into SysML, which enables systems engineers to design CPS with only these imported SysML blocks. Secondly, HybridSim can generate Functional Mock-up Units (FMUs) and configuration scripts directly from SysML designs. Finally, HybridSim can co-simulate these FMUs according to the Functional Mock-up Interface standard to synchronize their corresponding simulators and exchange information between them. We demonstrate the convenience and efficiency of HybridSim using a comprehensive hydronic heating system model for Smart Buildings as the case study to investigate the impact of packet loss and sampling rate introduced by the communication network.**

***Index Terms*—Co-simulation, TinyOS, Modelica, Functional Mock-up Interface, Cyber-Physical Systems**

I. Introduction

One of the most pervasive applications of Cyber-Physical systems (CPS) is to integrate cyber environment and physical systems through computing and networking [1]. Traditionally, sensor measurements are fed to controllers through wired networks, which are also used to deliver control commands to actuators. However, for some applications (e.g., Smart Buildings, manufacture monitoring, etc.), it can be very expensive and complex to deploy and evolve such wired networks. For some other applications (e.g., unmanned aerial vehicles, healthcare monitoring, etc.), it will be impossible to use wired networks for communications. Therefore, it is preferable to integrate computational and physical devices through self-organized wireless networks.

However, wireless networks suffer from unreliable wireless channels, especially in indoor applications and body-area networks. It is imperative to understand how wireless networks and control systems affect each other as a function of network traffic, topology and background interference, in terms of the stability and performance of control systems before deployment [2]. Considering the complexity of CPS, simulations are preferable to pure mathematical analysis. Given the multi-domain nature of CPS, it is more appropriate to use a heterogeneous simulation environment to study system dynamics. However, the integration of numerical solvers for continues-time models and simulators for event-triggered models is an established, but far-from-trivial problem [3].

In addition, different domain engineering groups generally exploit different domain languages and tools to model and evaluate their designs, which may not be familiar to systems engineers. It is error-prone and inefficient if systems engineers have to work with these domain languages and tools directly to design and evaluate overall systems. Instead, the outputs of domain groups should be integrated into a unified framework that is friendly to systems engineers [4].

In this paper, we design and implement an integrated modeling and co-simulation toolchain, called HybridSim, for the design and evaluation of CPS. HybridSim has three key features. Firstly, HybridSim provides a unified framework for systems engineers to integrate the outputs of domain groups to design overall systems. Specifically, we consider that a CPS consists of three main subsystems: 1) a cyber environment in which the CPS is deployed; 2) a wireless sensor and actuator network that is used to deliver sensor measurements and control commands; and 3) a control and computation system that is responsible to process information and make decisions. We assume the second subsystem is designed and implemented in TinyOS that is the most popular operating system for low-power wireless devices, and the other two subsystems are modeled and simulated in Modelica. HybridSim can transform and import TinyOS implementations and Modelica models into SysML blocks. Section VI discusses how to extend HybridSim to support other domain tools and languages.

Secondly, systems engineers can design CPS by using the imported SysML blocks in a drag-and-drop fashion. Such SysML blocks appear as black boxes and all domain-specific details are hidden from systems engineers. HybridSim provides a set of SysML stereotypes for systems engineers to configure overall systems and provide guidance for code generation and co-simulations.

Finally, HybridSim can generate configuration scripts and simulation modules for domain implementations and models automatically from SysML designs. Based on the Functional Mock-up Interface (FMI) standard [5], HybridSim can

1550-6525/13 $26.00 © 2013 IEEE
DOI 10.1109/DS-RT.2013.12

co-simulate these subsystems by leveraging the advantages of their respective professional simulators and emulators. HybridSim provides a robust and flexible mechanism for data exchange and synchronization between these subsystems.

The convenience and efficiency of HybridSim is demonstrated by using a comprehensive hydronic heating system model for Smart Buildings as the case study. We investigate the impact of sampling rates, background traffic, and network sizes in wireless sensor networks on the performance of the control system modeled in Modelica.

The rest of this paper is organized as follows. We discuss related work in Section II and introduce some background briefly in Section III. In Section IV, we explain the design and implementation of HybridSim in details. The case study is presented in Section V. Finally we discuss some observations, limits and future work for HybridSim in Section VI and conclude this work in Section VII.

II. Related Work

Since the design and synthesis of CPS has become a hot topic in both industry and academia, several co-simulation paradigms have been proposed. TrueTime [6] extends Matlab/Simulink with libraries for co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics. However, physical and MAC layer protocols must be modeled from scratch, which is complex and time-consuming. In addition, providing support for higher layer protocols in TrueTime can be a formidable task because they generally utilize complex algorithms that are distributed in nature and encompass multi-hop communications [2].

Viptos [7] extends TOSSIM [8] by providing interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network, while allowing developers to use Ptolemy II to model the physical environment and other parts of the system. However, information can only flow from Ptolemy II to TOSSIM because there is no way for Ptolemy II to receive data from TOSSIM. Therefore, Viptos cannot be used to investigate the impact of network dynamics on the performance of overall CPS.

Wang et al. [3] proposed a model-based systems design framework for wireless sensor networks. They described a hierarchy of model libraries to model various behaviors and structures of sensor networks, and proposed a system design flow to compose both continuous-time and event-triggered modules to develop applications with support for performance study by simulations. However, they cannot take advantage of existing models and implementations, and thus their model libraries must be developed from scratch. Similarly, Bombino et al. [9] proposed a model-driven co-simulation framework based on SysML for discrete event modeling, and on Matlab/Simulink for continuous time modeling.

The Modelica/ns-2 co-simulation platform [2] integrates Modelica and ns-2 for CPS, with ns-2 deciding their communication times. Therefore, sending data between Modelica and ns-2 in response to events generated inside Modelica is not supported. The synchronization mechanism is improved later in [10]. NCSWT (Networked Control System Wind Tunnel) [1] integrates Matlab/Simulink with ns-2 according to the High Level Architecture (HLA) standard. To conform to the HLA standard, the time management mechanisms of both simulators are modified to enable synchronization between them. However, the overhead of NCSWT is very large, which takes more than one hour to complete a 98-second simulation. There are also other similar works on the integration of Matlab/Simulink and ns-2 [11], [12].

In all the above work, simulators involved in co-simulations are tightly coupled with each other, and even need to be modified to enable data exchange and synchronization. Comparatively, HybridSim adopts the FMI standard, and only interacts with FMI-compatible API that are supported by more than 40 commercial and open-source tools[1]. This implies the extensibility of HybridSim to integrate other tools. In addition, existing works require systems engineers to work with domain languages and tools directly, while HybridSim enables systems engineers to work with only SysML blocks by transforming and importing domain implementations and models.

III. Background

In this section, we briefly introduce the tools and standards based on which HybridSim is designed and implemented.

A. Functional Mock-up Interface

FMI [5] is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML files and compiled C-code. FMI provides a set of standard functions to exchange data between subsystems and synchronize them in communication steps. These subsystems are called FMI slaves, while the co-simulation coordinator is called FMI master. FMI compliant models are referred to as Functional Mock-up Units (FMUs)

B. TinyOS and Avrora

TinyOS [13] is an open source operating system (OS) designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. TinyOS emphasizes reacting to external events, extremely low-power operation and small memory footprint. Rather than a monolithic OS, TinyOS is object-oriented, providing a set of components that are included as-needed in applications by extending the C language with concepts of interfaces, modules and configurations. TinyOS is the most popular OS that is widely used to design and implement wireless sensor networks.

Avrora [14] is a set of emulation and analysis tools for programs written for MicaZ and mica2 sensor platforms. Avrora contains a flexible framework for emulating and analyzing executable binary files compiled from TinyOS applications, providing a clean Java API and infrastructure for experimentation, profiling, and analysis. Unlike TOSSIM [8], Avrora provides cycle-accurate instruction-level timing accuracy.

[1] Full tool list: https://www.fmi-standard.org/tools

C. Modelica

Modelica [15] is an object-oriented language for modeling of large, complex, and heterogeneous systems. Models in Modelica are mathematically described by differential, algebraic and discrete equations. Modelica is designed so that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations. Dymola [16] is a commercial modeling and simulation environment based on Modelica, which can compile Modelica codes, solve equations and simulate Modelica models. Specifically, Dymola can generate an FMU by wrapping a whole Modelica system model.

D. SysML

SysML [17] is a general-purpose modeling language for systems engineering, with graphical representations for the specification, analysis, design, verification and validation of systems and systems-of-systems involving hardware, software and information. SysML is defined as an extension of a subset of UML. IBM Rational Rhapsody [18] is a software platform for model-based systems engineering based on SysML, which can generate source codes for SysML behavior diagrams for interactive simulations. In this work, Rhapsody is used as our development environment.

IV. HybridSim: Design and Implementation

In this section, we first introduce the workflow of HybridSim, and then describe the model integration environment and co-simulation toolchain in more details.

A. HybridSim Workflow

HybridSim is an important module of our WSNDesign Plug-in for IBM Rational Rhapsody, which is still under development. Its workflow is shown in Fig. 1, which roughly consists of three major steps: model transformation, system configuration, and co-simulation.

Model Transformation. The design of a cyber-physical system usually involves multiple engineering groups from different domains. For example, to design a smart building, we can have one group working on the HVAC (heating, ventilation, and air conditioning) system, and another group working on the communication network. Generally, different domain groups exploit different languages and tools to model and evaluate their designs, which may not be familiar to systems engineers. HybridSim solves this problem by transforming and importing domain models into SysML, which is widely used for model-based systems engineering. Currently HybridSim can transform TinyOS components, Modelica modules and FMUs, and import them as SysML blocks.

System Configuration. Systems engineers can create SysML Block Definition Diagrams and SysML Internal Block Diagrams, by selecting TinyOS SysML blocks, Modelica SysML blocks and FMU SysML blocks to set up a system-level simulation environment to evaluate the performance and dynamics of an overall system. Particularly, systems engineers need to configure simulation parameters for both TinyOS emulator and Modelica simulator, and specify information exchanging between them.

Co-simulation. After systems engineers have set up a simulation environment, HybridSim will compile TinyOS application codes and Modelica models, and generate their corresponding FMUs. Meanwhile, related configuration files and simulation scripts are generated automatically as well. Then HybridSim launches the FMI co-simulation master to co-simulate TinyOS FMU and Modelica FMU, and outputs their corresponding results.

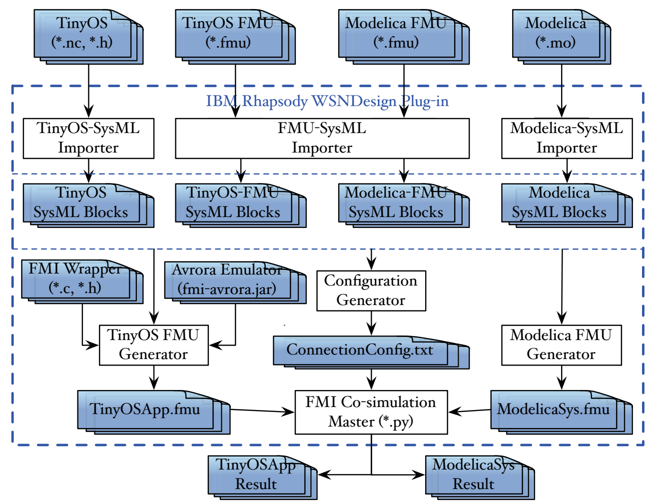


Fig. 1. HybridSim Toolchain Workflow

B. SysML-centric Integration Environment

HybridSim is developed according to the following principles:

- Domain engineers work with the languages and tools with which they are familiar. They should not be required to re-implement their designs just for overall system simulation and evaluation. Ideally, domain engineers just need to provide standard interfaces for systems engineers to exchange information and synchronize with other domain simulators.
- Systems engineers only need to work with their favorite languages and tools as well. For overall system integration and simulation, they take the outputs of domain engineering groups as black boxes. Systems engineers only need to work with the interfaces provided by domain implementations and models.
- Systems engineers should take advantage of exiting professional simulators/emulators in each domain for overall system evaluation. In addition, the burden to set up a co-simulation environment should be minimized for systems engineers.

Support Modelica. As SysML is widely used for model-based systems engineering, HybridSim selects SysML as the working language for systems engineers. Meanwhile, Modelica is becoming increasingly popular to model CPS in both industry and academia in recent years. For example, researchers at the Lawrence Berkeley National Laboratory have developed a Modelica Buildings Library with dynamic simulation models for building energy and control systems in Smart Buildings [19]. Therefore, HybridSim supports Modelica as a very important domain language. Specifically, HybridSim can import Modelica models in two formats into SysML blocks: both Modelica FMUs and Modelica source codes.

The FMU-SysML Importer reads the model description (i.e., *modelDescription.xml*) of a Modelica FMU to figure out its input and output ports, and then creates a SysML block with corresponding input and output flow ports. In addition, the `ModelicaFMU` stereotype is applied to this SysML block, which has a tag referring to the original FMU for simulation code generation.

Similarly, the Modelica-SysML Importer parses Modelica source codes to extract its input and output ports information, and then generates SysML blocks. Unlike SyM [20], HybridSim only extracts the high-level structure information of a Modelica module, and ignores its internal component connections and behavior definitions. Therefore, this importer is much simpler and very stable. The `ModelicaSource` stereotype is applied to generated SysML blocks to refer their behaviors to the original source codes, and specify which tool should be used to generate FMUs from these SysML blocks. Currently, Dymola is the only tool supported by HybridSim. However, it is very easy to extend HybridSim to support OpenModelica or other Modelica tools.

Support TinyOS. WSNs will play a fundamental role in future CPS, such as Smart Buildings [21]. As TinyOS is the most popular operating system to develop sensor applications, HybridSim supports TinyOS and its programming language, nesC, for communication networks design. Similarly, HybridSim can transform and import TinyOS components in two formats into SysML blocks: TinyOS FMUs and nesC source codes.

As Modelica FMUs, TinyOS FMUs are transformed and imported by the FMU-SysML Importer as well. However, the `TinyOSFMU` stereotype is applied to their corresponding SysML blocks, which provides additional tags for co-simulation setup in addition to the reference to the original FMU. To the best of our knowledge, HybridSim is currently the only tool that can generate TinyOS FMUs. Considering that design groups of communication systems may not be familiar with SysML and IBM Rational Rhapsody, an independent version of TinyOS FMU Generator of HybridSim is also available.

To import TinyOS components from nesC source codes, HybridSim takes advantage of the XML output feature of the nesC compiler, from where input and output interfaces of TinyOS components are extracted. If a component does not have any interface (i.e., it is a complete TinyOS application), the `TinyOSSource` stereotype is applied to its corresponding SysML block. Otherwise, the `TinyOSComponent` stereotype is applied.

Systems engineers can integrate a sensor network implementation into the overall system in two ways from source codes. The easiest way is to select an existing SysML block tagged with `TinyOSSource`. The other way is to create a new SysML block that is applied with `TinyOSSource`, and then specify a reference to an executable binary file, or create an Internal Block Diagram for it. Systems engineers can select and connect existing TinyOS components in a drag-and-drop fashion in this diagram to specify its source code. This enables systems engineers to develop simple sensor network applications for partial evaluation of overall systems before real applications are finished. Note that HybridSim only has a limited support for the second method, without interface compatibility checking for connections. In addition, although HybridSim can make use of existing imported TinyOS components, systems engineers cannot modify their source codes or create new TinyOS components in HybridSim. This is reasonable because systems engineers are not supposed to directly work with domain languages.

In order to exchange information with Modelica models, HybridSim provides a simple mechanism that allows systems engineers to input data coming from Modelica simulators into sensors, and vice verse. An example is shown in Fig. 3. To input data to sensors on a mote, systems engineers just need to add input flow ports to the SysML block (tagged with `TinyOSSource`) representing the application to be run on that mote. The name of an input port must exactly be the type of the target sensor (e.g., temperature, light, etc.). Currently, only scalar-valued sensors are supported. To get data out of motes, output flow ports should be added to their corresponding SysML blocks. The name of an output port must exactly be the name of a variable declared with module-scope in some TinyOS module. The `TinyOSVariable` stereotype should be applied to that port to specify in which module that variable is declared.

C. FMI-based Co-simulation Toolchain

HybridSim adopts the FMI standard to co-simulate TinyOS and Modelica, and synchronizes and exchanges information between their respective simulators by calling their FMI-compatible API. The implementation details of the Modelica FMU Generator, the TinyOS FMU Generator, the Configuration Generator and the FMI Co-simulation Master are described as follows.

Modelica FMU Generator. If a SysML block is tagged with `ModelicaFMU`, HybridSim just needs to forward the location of the corresponding FMU to the Configuration Generator. If that block is tagged with `ModelicaSource`, HybridSim calls Dymola to compile the source codes and generate an FMU.

TinyOS FMU Generator. SysML blocks tagged with `TinyOSFMU` are processed similarly as blocks tagged with `ModelicaFMU`. For blocks tagged with `TinyOSSource`, HybridSim needs to compile nesC source codes, prepare the TinyOS Emulator, and compile the FMI Wrapper source codes.

To compile nesC source codes, HybridSim first checks how the TinyOS application is created. If it is directly imported from TinyOS libraries, HybridSim just needs to set up certain environment variables and read its associated *Makefile* to compile it. Otherwise, if it is created in the second way as described above, HybridSim will collect information about all the required TinyOS components and generate a *Makefile*. The output of this step is a set of ELF (Executable and Linkable Format) files, which will be loaded by the TinyOS Emulator.

The TinyOS Emulator is developed based on Avrora [14] in Java, which is extended as follows. Firstly, the TinyOS Emulator provides interfaces to input data into sensors from outside interactively. Secondly, the TinyOS Emulator can extract runtime values of variables from TinyOS components. By processing the ports tagged with `TinyOSVariable`, the TinyOS Emulator can infer the actual variable names in ELF files, and their memory addresses when these ELF files are loaded into RAM. Monitors are added to monitor these RAM addresses, which can reconstruct the variable values when they are updated. These two extensions enable the TinyOS Emulator to exchange data with a Modelica simulator. Finally, a new synchronization mechanism is developed, which can synchronize all threads of sensor motes, get information about the next system-wide event from local event queues, and enable the FMI Co-simulation Master to control the size of each synchronization step.

The FMI Wrapper is developed based on FMU-SDK [22] in C, which is extended with a more flexible mechanism to handle variable references. In addition, the FMI Wrapper provides convenient interfaces to interactive with the TinyOS Emulator through the Java Native Interface framework. Finally, the FMI Wrapper can process the output of the Configuration Generator to generate *modelDescription.xml* and compile all files into a stand-alone FMU with the TinyOS Emulator included.

Configuration Generator. The main output of this module is two configuration files for co-simulation. The first one is an Avrora configuration script, which specifies a set of options for the TinyOS Emulator, sensor types for each mote, and a set of variables whose values should be extracted and the TinyOS components in which they are declared. The second one is a connection configuration file, which specifies connections of the output and input ports of a TinyOS FMU and a Modelica FMU. This file is used by the FMI Co-simulation Master to exchange information between the TinyOS Emulator and a Modelica simulator.

FMI Co-simulation Master. Fig. 2 describes the algorithm of the FMI Co-simulation Master, which is developed based on PyFMI [23] in Python. The master first parses the connection configuration file and records connections in two hashtables. The key of an entry is an output port of the given FMU, and the value is its corresponding input port of the other FMU. After the two FMU slaves have been instantiated and initialized, the master begins to co-simulate them in steps. In each step, the master first reads the values of output ports of the Modelica slave, which are then forwarded to the TinyOS slave. Next, the step size δ is computed as the minimum of the specified communication step size, and the two incremental times of the next event in the TinyOS slave and in the Modelica slave, respectively. Therefore, no interaction event is missed. Then the master stimulates the TinyOS slave to emulate up to $t+\delta$. After that, the values of its output ports are forwarded to the input ports of the Modelica slave, which is then stimulated to simulate up to $t+\delta$ similarly. When the specified end time of co-simulation is reached, the master terminates both slaves and releases the resource occupied by them. The co-simulation results for both slaves are stored in Dymola format, which can be loaded into Dymola for further analysis, or plotted in HybridSim directly.

Input: Two FMUs, stopTime, connSize, config.txt
Output: TOSResult.txt, MdlResult.txt

```
[TOSConn, MdlConn] ← parse(config.txt);
TOSSlave ← loadFMU(TinyOSApp.fmu);
MdlSlave ← loadFMU(ModelicaSys.fmu);
TOSResult = newDymolaWriter(TOSSlave);
MdlResult = newDymolaWriter(MdlSlave);
TOSSlave.initialize(); MdlSlave.initialize();
t ← 0;
while t < stopTime do
    values ← MdlSlave.get(MdlConn.keys);
    TOSSlave.set(MdlConn.values, values);
    δ ← getStep(connSize, TOSSlave, MdlSlave);
    TOSSlave.doStep(t, δ);
    values ← TOSSlave.get(TOSConn.keys);
    MdlSlave.set(TOSConn.values, values);
    MdlSlave.doStep(t, δ);
    TOSResult.writePoint(); MdlResult.writePoint();
    t ← t + δ;
end while
TOSSlave.terminate(); MdlSlave.terminate()
TOSSlave.free(); MdlSlave.free()
```

Fig. 2. FMI Co-simulation Master

V. Case Study

In this section, we use a comprehensive hydronic heating system as the case study to demonstrate the convenience and efficiency of HybridSim. Specifically, we investigate the impacts of packet loss and sampling rate that are introduced by wireless sensor networks on the heating system.

A. Modelica Model and TinyOS Application

Our Modelica model is developed based on an example provided by the Modelica Buildings Library [19], which comprehensively models the hydronic heating system for a building with energy storage and thermostatic radiator valves. Two rooms on the same intermediate floor are modeled using a dynamic model for the heat transfer through opaque constructions, with the same temperature above and below them. They share one common wall and have two windows. Realistic weather data traces from Chicago are fed into this model as the outside environment.

The hydronic heating system consists of a boiler, a storage tank and a radiator with a thermostatic valve in each room. The supply water temperature setpoint is reset based on the outside temperature. A three-way-valve mixes the water from the tank with the water from the radiator return. The pump has a variable frequency drive that controls the pump head. The building has a controlled fresh air supply with a heat recovery ventilator to preheat the outside air. Each room has a leakage model of the facade, through which the difference in air supply will flow if the supply and exhaust air are unbalanced.

A finite state machine is used to control the boiler and its pump. They are switched on when the temperature at the top of the tank is less than 1 Kelvin above the setpoint temperature for the supply water of the radiator loop. The boiler is switched on 10 seconds later than the pump. They are switched off when the temperature at the bottom of the tank reaches 55 °C. The pump is switched off 10 seconds later than the boiler.

In this case study, we only care about the temperatures of the two rooms, and the top and bottom of the tank, each of which is monitored by a temperature sensor. Originally, their readings are directly fed into the controller without any delay or loss. To investigate the impact of wireless sensor networks, the connections between the sensors and the controller are deleted. Instead, their readings are provided to four output ports respectively. Correspondingly, four input ports that are connected to the controller are created to get data from outside.

The TinyOS application consists of a set of relay motes and four sensor motes sampling the temperatures of the two rooms, and the top and bottom of the tank. In addition, a sink (i.e., base station) is assigned to collect all their readings through multi-hop communications. The Collection Tree Protocol [24] is selected as the routing protocol, and the Rician fading channel model is applied in the TinyOS Emulator.

Fig. 3 demonstrates a system configuration for one of our co-simulation scenarios. Four sensor motes (two on the left and two on the right) are created by referring to an existing ELF file, which is created by the nesC compiler. The sink and relay motes are created by referring to the *Makefile* for their source codes (the ELF file is actually compiled from this source codes), while the hydronic heating system model is imported from an existing Modelica FMU.

HybridSim first generates a configuration file for the TinyOS Emulator as shown in Fig. 4. The topology file is generated by HybridSim based on the ID and location of each mote as specified in Fig. 3. Each parameter is assigned a unique reference number, which is required by the FMI standard. The name of an output parameter is `N⟨id⟩_⟨component⟩__⟨outPort⟩`, while that of an input parameter is `N⟨id⟩_⟨inPort⟩`. This file is also parsed to generate the TinyOS FMU. The corresponding configuration for co-simulation setup is shown in Fig. 5.

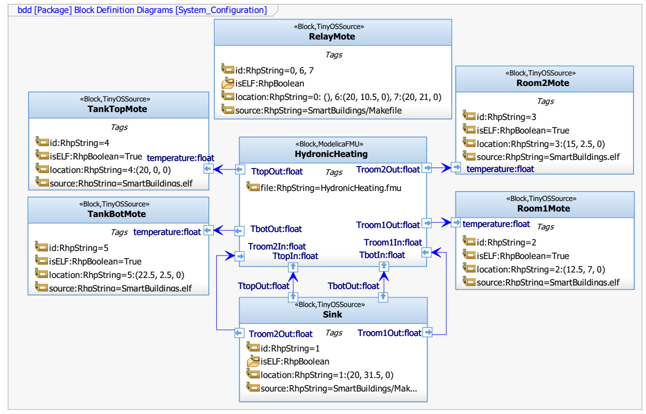


Fig. 3. System Configuration

```
# options for the Avrora emulator
options = -nodecount=8 -topology-file=top.txt ...
    SmartBuildings.elf
# direction refNum paramName initialValue
output 915 N1_SmartBuildingsC__Troom1Out 20
output 925 N1_SmartBuildingsC__Troom2Out 20
output 935 N1_SmartBuildingsC__TtopOut 20
output 945 N1_SmartBuildingsC__TbotOut 20
input 715 N2_temperature 20
input 725 N3_temperature 20
input 735 N4_temperature 20
input 745 N5_temperature 20
```

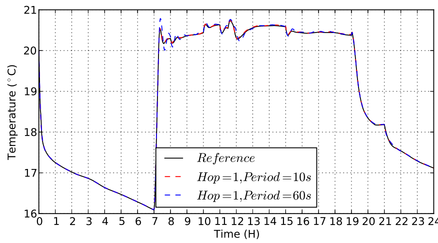
Fig. 4. TinyOS Emulator Configuration

```
# Direction: output = input
# HH: HydronicHeating, SB: SmartBuildings
# Abbreviated from the original file
HH->TtopOut = SB->N4_temperature
HH->TbotOut = SB->N5_temperature
HH->Troom1Out = SB->N2_temperature
HH->Troom2Out = SB->N3_temperature
SB->N1_SmartBuildingsC__TtopOut = HH->TtopIn
SB->N1_SmartBuildingsC__TbotOut = HH->TbotIn
SB->N1_SmartBuildingsC__Troom1Out = HH->Troom1In
SB->N1_SmartBuildingsC__Troom2Out = HH->Troom2In
```

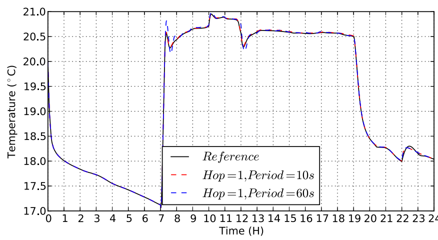
Fig. 5. Co-simulation Configuration

B. Co-simulation Results

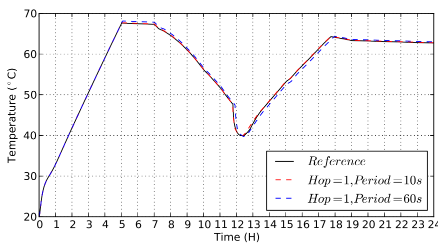
We first investigate the impact of sampling rate of sensor motes by simulating the system for one day. The communication step size for the TinyOS slave and the Modelica slave to exchange data and synchronize with each other is one second. Fig. 6 shows the simulation results when the sampling period is 10 seconds and 60 seconds, respectively. The four sensor motes locate within one-hop neighborhood of the sink, and no background traffic is considered. The simulation result of the original Modelica model in which all sensor readings are fed into the controller directly is used as the reference. As we can see, when the sampling period is 10 seconds, its impact is imperceptible. If the sampling period is increased to 60 seconds, there is a little variation, but its result is still very close to the reference. This is reasonable because temperature usually changes slowly. This indicates that wireless sensor networks can be used in some CPS to reduce deployment complexity and costs. Especially for slow response systems, such as our hydronic heating system, low sensor duty cycles can still guarantee acceptable system performance while increasing sensor lifetime significantly.



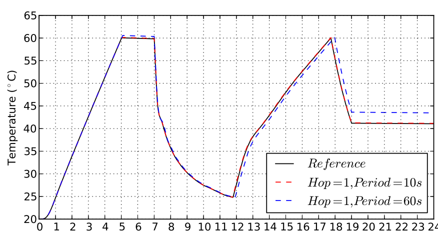
(a) Room1 Temperature



(b) Room2 Temperature



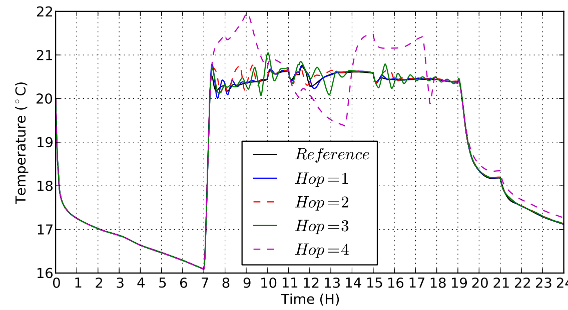
(c) Temperature of Tank Top



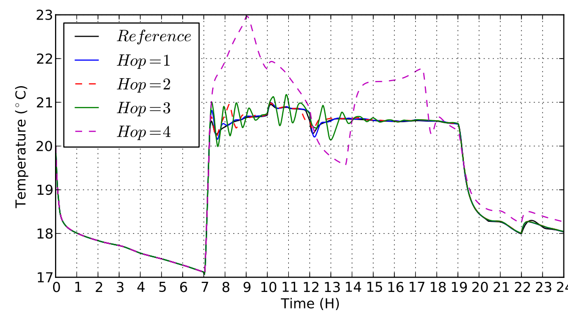
(d) Temperature of Tank Bottom

Fig. 6. Impact of Sampling Rates

Smart Buildings, especially commercial buildings, are not clean environments for sensor communications, which usually suffer from heavy background traffic interference, such as WiFi traffic and Bluetooth traffic. In this scenario, we simulate the hydronic heating system under heavy WiFi traffic interference. Specifically, we feed a real-world noise trace into the TinyOS Emulator, which is measured in the Meyer Library at Stanford University with a large HTTP download and other WiFi traffic going-on [25]. The simulation results for different network sizes are shown in Fig. 7, which indicate that it is impractical to deploy a sensor network with more than 3 hops from a sensor mote to the sink in our case. Note that we only analyze the simulation results directly in HybridSim in this case study, which can also be processed by exiting tools for Avrora emulator and Modelica simulator.



(a) Room1 Temperature



(b) Room2 Temperature

Fig. 7. Impact of Network Size

Table I shows the run-time efficiency of HybridSim. It can be seen that the performance of HybridSim depends on the network complexity. Actually, the efficiency is dominated by TinyOS Emulator that provides cycle-accurate instruction-level emulation.

TABLE I
Run-time Efficiency (Simulation Time = 24 Hours)

Scenario	# of Nodes	Runtime (H)
1 Hop	6	4.54
2 Hop	8	6.18
3 Hop	11	9.04
4 Hop	14	11.24

VI. Discussion

In this section, we discuss some observations, limits and future work for HybridSim.

Communication step sizes impact co-simulation overhead and accuracy. HybridSim adopts variable communication step sizes based on the time of the next event in both slaves to provide fine-grained interactions between them, so that no event potentially requiring interactions is missed. However, in some scenarios, such a big overhead is not necessary. For example, in our case study, the TinyOS slave and the Modelica slave only interact at sampling points, and the sampling period is much larger than variable communication steps. If communication step sizes are fixed to a reasonable value (e.g., 1 second), the overhead imposed by the co-simulation master can be reduced significantly, while a good accuracy is still guaranteed. Systems engineers need to study the trade-off between overhead and accuracy depending on their system characteristics.

HybridSim can be extended with more interesting features. Firstly, the ns-3 network simulator [26] can be integrated into HybridSim. Since ns-3 is a more general discrete-event simulator for both wireless and wired networks, and preferable for fast prototyping of network designs, this feature will make HybridSim more interesting to a larger community. An FMI wrapper for ns-3 should be developed, similar to our FMI wrapper for the Avrora emulator. In addition, it will be very interesting to integrate Matlab/Simulink into HybridSim as well. For this feature, an FMU generator for Matlab/Simulink models is desired.

Another future work is to enhance the coupling between HybridSim and IBM Rational Rhapsody, which currently only serves as the environment for the FMI Co-simulation Master. We are interested in integrating the Rhapsody simulator into HybridSim, which can generate simulation source codes for SysML Activity Diagrams and Statechart Diagrams. With this feature, HybridSim will become a much better design framework and toolchain for model-based systems engineering. Another interesting topic is to support distributed co-simulations, which is also specified in the FMI standard and is more suitable for large complex systems.

VII. Conclusions

In this paper, we designed and implemented a modeling and co-simulation toolchain, called HybridSim, for Cyber-Physical Systems. HybridSim can transform and import existing TinyOS components and Modelica models into SysML, so that systems engineers can design and simulate overall systems in a uniform framework. Based on the FMI standard, HybridSim provides a robust and flexible mechanism to exchange data and synchronize the TinyOS Emulator and the Modelica simulator. Therefore, HybridSim can leverage the advantage of their respective professional emulator and simulator, and investigate complex cyber-physical interactions. In addition, HybridSim enables domain engineering groups to work relatively independently, while facilitating systems engineers to design and evaluate overall systems by using outputs from domain groups. A comprehensive case study is discussed to illustrate the convenience and efficiency of HybridSim.

Acknowledgements

This work was partially supported by the National Science Foundation (NSF) under grant award CNS-1035655, and by the National Institute of Standards and Technology (NIST) under grant award 70NANB11H148.

References

[1] D. Riley, E. Eyisi, J. Bai, X. Koutsoukos, Y. Xue, and J. Sztipanovits, "Networked control system wind tunnel (ncswt): an evaluation tool for networked multi-agent systems," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2011, pp. 9–18.

[2] A. T. Al-Hammouri, M. S. Branicky, and V. Liberatore, "Co-simulation tools for networked control systems," in *Proceedings of the 11th international conference on Hybrid Systems: Computation and Control (HSCC)*, 2008, pp. 16–29.

[3] B. Wang and J. S. Baras, "Integrated modeling and simulation framework for wireless sensor networks," in *IEEE 21st WETICE*, 2012, pp. 1 – 6.

[4] P. Derler, E. Lee, and A.-S. Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.

[5] MAP FMI, "Functional Mock-up Interface," *https://www.fmi-standard.org/*.

[6] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. Arzen, "How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime," *IEEE Control Systems*, vol. 23, no. 3, pp. 16–30, 2003.

[7] E. Cheong, E. A. Lee, and Y. Zhao, "Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks," in *ACM SenSys*, 2005, pp. 302–302.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire tinyos applications," in *ACM SenSys*, 2003, pp. 126–137.

[9] M. Bombino and P. Scandurra, "A model-driven co-simulation environment for heterogeneous systems," *International Journal on Software Tools for Technology Transfer*, pp. 1–12, 2012.

[10] A. T. Al-Hammouri, "A comprehensive co-simulation platform for cyber-physical systems," *Computer Communications*, vol. 36, no. 1, pp. 8–19, Dec. 2012.

[11] T. Kohtamaki, M. Pohjola, J. Brand, and L. Eriksson, "PiccSIM toolchain - design, simulation and automatic implementation of wireless networked control systems," in *Networking, Sensing and Control, 2009. ICNSC '09. International Conference on*, 2009, pp. 49–54.

[12] O. Heimlich, R. Sailer, and L. Budzisz, "NMLab: A co-simulation framework for matlab and ns-2," in *Advances in System Simulation (SIMUL), 2010 Second International Conference on*, 2010, pp. 152–157.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, Nov. 2000.

[14] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *ACM/IEEE IPSN*, 2005.

[15] Modelica Association, "Modelica Language Specification 3.3," *https://www.modelica.org*.

[16] Dassault Systemes, "Dymola," *http://www.3ds.com/products/catia/portfolio/dymola*.

[17] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language (2nd Edition)*. Morgan Kaufmann Publishers Inc., 2011.

[18] IBM, "IBM Rational Rhapsody Help," *www.ibm.com/software/awdtools/rhapsody/*.

[19] Lawrence Berkeley National Laboratory, "Modelica Buildings Library," *http://simulationresearch.lbl.gov/modelica/*.

[20] C. J. Paredis, Y. Bernard, R. M. Burkhart, H.-P. de Koning, S. Friedenthal, P. Fritzson, N. F. Rouquette, and W. Schamai, "An overview of the sysml-modelica transformation specification," in *INCOSE International Symposium*, 2010.

[21] R. Fontugne, J. Ortiz, N. Tremblay, P. Borgnat, P. Flandrin, K. Fukuda, D. Culler, and H. Esaki, "Strip, bind, and search: a method for identifying abnormal energy consumption in buildings," in *ACM/IEEE IPSN*, 2013, pp. 129–140.

[22] QTronic GmbH, "FMU SDK 1.0.2," *http://www.qtronic.de/en/fmusdk.html*.

[23] JModelica.org, "PyFMI 1.2.1," *http://www.jmodelica.org/page/4924*.

[24] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *ACM SenSys*, 2009, pp. 1–14.

[25] H. Lee, A. Cerpa, and P. Levis, "Improving wireless simulation through noise modeling," in *ACM/IEEE IPSN*, 2007, pp. 21–30.

[26] NS-3 Consortium, "ns-3 Network Simulator," *http://www.nsnam.org/*.