Enabling Model Testing of Cyber-Physical Systems

Carlos A. González
University of Luxembourg
Luxembourg
gonzalez@svv.lu

Mojtaba Varmazyar
University of Luxembourg
Luxembourg
varmazyar@svv.lu

Shiva Nejati
University of Luxembourg
Luxembourg
nejati@svv.lu

Lionel C. Briand
University of Luxembourg
Luxembourg
briand@svv.lu

Yago Isasi
LuxSpace Sàrl
Luxembourg
isasi@luxspace.lu

ABSTRACT

Applying traditional testing techniques to Cyber-Physical Systems (CPS) is challenging due to the deep intertwining of software and hardware, and the complex, continuous interactions between the system and its environment. To alleviate these challenges we propose to conduct testing at early stages and over executable models of the system and its environment. Model testing of CPSs is however not without difficulties. The complexity and heterogeneity of CPSs renders necessary the combination of different modeling formalisms to build faithful models of their different components. The execution of CPS models thus requires an execution framework supporting the co-simulation of different types of models, including models of the software (e.g., SysML), hardware (e.g., SysML or Simulink), and physical environment (e.g., Simulink). Furthermore, to enable testing in realistic conditions, the co-simulation process must be (1) fast, so that thousands of simulations can be conducted in practical time, (2) controllable, to precisely emulate the expected runtime behavior of the system and, (3) observable, by producing simulation data enabling the detection of failures. To tackle these challenges, we propose a SysML-based modeling methodology for model testing of CPSs, and an efficient SysML-Simulink co-simulation framework. Our approach was validated on a case study from the satellite domain.

CCS CONCEPTS

• **Software and its engineering → Software system models**; **Model-driven software engineering**; **Software verification and validation**;

KEYWORDS

Cyber-Physical Systems, Software Testing, Model-Based Systems Engineering

ACM Reference Format:
Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. 2018. Enabling Model Testing of Cyber-Physical Systems. In *Proceedings of ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A Cyber-Physical System (CPS) is an integration of computation with physical processes [23]. In a CPS, a potentially large number of networked embedded computers, heavily interact with each other to monitor and control physical processes through sensors and actuators. Applications of CPSs [17] are expected to revolutionize [32] the way humans control and interact with the physical world.

Ensuring the dependability of CPSs is challenging due to specific difficulties in applying traditional testing techniques. Characteristic features of CPSs, such as the deep intertwining of software and hardware, or the complex and continuous interactions between the system and its environment, render difficult the automated execution of a large number of test scenarios within practical time, thus affecting the effectiveness of the testing process. Moreover, when hardware constraints come into play (e.g., hardware that can be damaged by action of the testing process, or that is developed after the software), the testing process can be significantly delayed, become highly expensive, or fail to cover certain scenarios.

One way to alleviate these challenges is to move as much as of the testing activity as possible to earlier design phases and away from the deployed system and, instead, conduct model testing [7], that is testing over executable design models of the system and its environment. The use of such executable models enables the running of a large number of test scenarios in a completely automated fashion. Such exploration of the system input space increases the chances of uncovering faults early in the system design. Model testing is in many ways similar to Model-in-the-Loop testing (MiL) [34], the first phase of testing in CPS development, except that the latter focuses on control algorithms and their interactions with physical models, not the software system. Since model testing is intended for testing the design of software systems, this activity is expected to take place in the Software-in-the-Loop phase (SiL) [34], at an early stage, once the software architecture and design are defined.

The level of detail at which models are described, and the amount of information produced during model executions, can also be tailored to meet various testing goals. For example, these may include the targeting of a particular type of defect, enhancing the selection of test cases at runtime to identify high-risk test scenarios, or deriving automated test oracles. High-risk test scenarios identified during model testing can then be employed to guide the testing activity at subsequent phases, for example at later stages of SiL when source code is available or during Hardware-in-the-Loop testing (HiL) [34].

One of the main challenges regarding model testing of CPSs is supporting model execution in such a way as to produce realistic testing results. The complexity of CPSs requires the combined use of models expressed with different formalisms, to represent physical aspects of the system and its environment, with their continuous dynamics, as well as software aspects of the system, including their time-discrete behavior. The model testing execution framework must therefore support model co-simulation for appropriate modeling formalisms to enable their coordinated execution [33]. Furthermore, the co-simulation process must be sufficiently fast to allow the completion of thousands of model executions within practical time. Last, the co-simulation process must also be (1) controllable, to precisely drive the runtime behavior of the system, and (2) observable, to produce meaningful simulation data, including sufficient insights into the system runtime behavior to support the analysis of test results.

Unfortunately, as discussed in Section 3, existing co-simulation approaches either do not support the execution of architectural and behavioral software models, e,g., [1–4, 6, 9], or fall short from fulfilling some of the additional features required for model testing [8, 19, 30, 35].

Contributions. Since SysML [28], the OMG[1] standard for systems engineering, is a modeling language commonly used in the industry, and Simulink is a de facto standard for what is referred to as function modeling (i.e., modeling of control algorithms and physical environment) in industrial settings, we propose a SysML-based modeling methodology, for the systematic construction of testable models of CPSs, and an efficient SysML-Simulink[2] co-simulation framework, equipped with the controllability and observability capabilities needed to enable model testing of CPSs. We validate the suitability of our approach with an industrial case study from the satellite domain.

Organization. Section 2 outlines our case study. Section 3 compares our work with the related techniques and tools. Section 4 presents our model testing methodology. Section 5 describes our co-simulation framework. Section 6 evaluates our approach. Section 7 concludes the paper.

2 MOTIVATION

We motivate our work with an industrial system from LuxSpace Sàrl[3], a leading system integrator for micro satellites and aerospace systems. This system is used to determine the satellite attitude and control its movements. It is referred to as the Attitude Determination and Control System (ADCS). ADCS controls the satellite autonomously, but it further enables engineers to control the satellite from the ground station. At any point in time, the ADCS will be in a certain mode, which will determine which ADCS capabilities are available. Figure 1 shows a conceptual view of the ADCS: it is composed of four subsystems, namely, APDM, ACM, DEM and MM.

[1]https://www.omg.org
[2]https://nl.mathworks.com/products/simulink.html
[3]https://luxspace.lu/

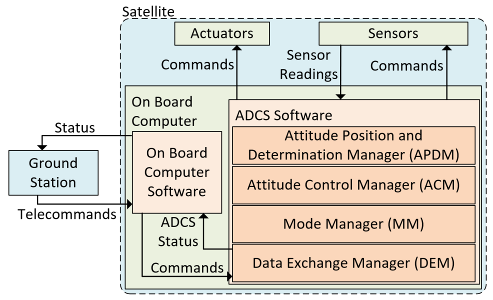


Figure 1: The ADCS of the satellite

The APDM is responsible for determining the satellite's attitude, by interacting with the satellite's sensors. The ACM is in charge of adjusting the satellite's attitude, by interacting with the satellite's actuators. The MM is responsible for monitoring and recording state changes. Finally, the DEM handles the communication with the on board computer, to enable the ADCS to receive telecommands, i.e., commands sent from the ground station to the satellite, and send telemetry data, i.e., data sent from the satellite to the ground station.

During the MiL phase, LuxSpace control engineers create function models with Simulink to simulate the environment of the satellite while in orbit; the ADCS functionality that is heavily driven by mathematical calculations, e.g., control algorithms; and the sensors and actuators the ADCS interacts with. Since these models do not cover software aspects, such as the architecture of the ADCS or the scheduling of ADCS's tasks, they must be completed during the early stages of SiL with the following artifacts:

1) Software architectural models that describe the internal decomposition of the ADCS into subsystems and tasks, and the interfaces and dependencies among them. The level of detail of such modeling is partially determined by the needs of our model testing objectives but is also driven by other development activities.

2) Software behavioral models that describe the ADCS functionality not already modeled in Simulink, as well as how the system transitions from one state to another.

3) Model integration points that specify how software behavioral models and function models are executed together, to enable co-simulation.

Once the modeling activity is completed, our goal is to automatically generate a co-simulation framework to simulate both the ADCS and its environment. To enable model testing, this framework must fulfill the following requirements:

Efficiency (EFF): The co-simulation framework shall be capable of performing thousands of model executions within practical time, typically overnight. This is required to efficiently explore the very large space of possible test scenarios.

SysML-Simulink Co-simulation (CO-S): The co-simulation framework shall support the synchronized execution of SysML software models and Simulink models. These choices are motivated by SysML being the most commonly used [5] system modeling language in industrial settings and an OMG standard supported by INCOSE[4], the International Council on Systems Engineering.

[4]https://www.incose.org/

Table 1: Degree of fulfillment of model testing requirements by existing co-simulation frameworks

	EFF	CO-S	NO-U	CONT	OBS
TASTE	High	Medium	Low	High	Low
INTO-CPS	?	Medium	Low	High	Low
OpenMeta	Medium	Low	Low	Low	Low
CyPhySim	?	Medium	Low	Medium	Low

Simulink, on the other hand, is widely used in many industrial domains for building function models.

No User Intervention (NO-U): Users shall only be involved in the modeling activity. The co-simulation framework shall execute models automatically, without requiring user intervention.

Controllability (CONT): The co-simulation framework shall enable the execution of models in a way that complies with the expected behavior of the future deployed system and its environment, at the level of abstraction required for conducting model testing. Controllability requires, among other things, the precise simulation of the scheduling of system tasks, and the data flows among subsystems and between the system and its environment.

Observability (OBS): The co-simulation framework shall save execution traces at runtime, capturing the information required to support test generation and analysis. This requires capturing, among other information items, data exchanges, the occurrence of external events, CPS state changes, and computation results. Each entry in these traces must be time-stamped.

Most of these requirements drive the construction of the architectural and behavioral software models. In Section 4, we provide some methodological guidance to support the systematic construction of testable models for CPSs.

3 BACKGROUND

A great deal of research has been conducted in recent years to identify effective ways to conduct model co-simulation, yielding numerous approaches [15]. However, the majority of them are not suitable for enabling model testing of CPSs since they do not support the inclusion of architectural and behavioral software models, e.g., [1–4, 6, 9, 14, 18, 29, 40].

To the best of our knowledge, TASTE [30], INTO-CPS [12, 19], OpenMETA [35], Cosimate[5] and CyPhySim [8, 22] are the co-simulation frameworks with the highest potential to enable model testing of CPSs, including software aspects. In what follows, we briefly describe these tools[6] and evaluate their suitability for model testing, based on the requirements identified in Section 2.

TASTE (The Assert Set of Tools for Engineering) is an open-source tool-chain for the development of embedded, real-time systems. TASTE relies on three different modeling languages: AADL (Architecture Analysis & Design Language) [11] for describing the logical and physical architecture of the system; ASN.1 (Abstract Syntax Notation One) [37] for describing data types and encoding rules; and SDL (Specification and Description Language) [36] for describing the overall orchestration of the system. The tool-chain uses the information in the models to generate skeleton code, that must then be manually completed by injecting the code describing the behavior of the different artifacts modeled. This additional code can be manually written or automatically generated from tools like Simulink or SCADE[7]. When the code is run, execution traces can be recorded in the form of Message Sequence Charts (MSC) [38], a graphical language for the description and specification of the interactions between system components.

CyPhySim is a Cyber-Physical Simulator based on Ptolemy II [10], a modeling and simulation framework for heterogeneous models. CyPhySim supports the Functional Mock-up Interface (FMI) standard [26] for the co-simulation of function models. In the FMI standard, models are encapsulated in so-called Functional Mock-up Units (FMU). Compliant simulation environments can then import and instantiate these FMUs to enable co-simulation[8]. When it comes to supporting software models, CyPhySim combines state machines for behavioral modeling, and the actor-oriented [21] language of Ptolemy II for structural modeling.

INTO-CPS is an integrated tool-chain for Model-Based Design (MBD) of CPSs. With INTO-CPS, the first step is to use SysML to model the decomposition of the CPS into a series of subsystems, represented as SysML blocks. It is important to mention that only the modeling of the subsystems' interfaces is conducted in this step. These SysML models are then used to produce both, a simulation configuration file describing how the execution of the different subsystems must be orchestrated, and descriptions of the subsystems' interfaces. In a second step, specific tools in the domain of each subsystem (physical, hardware, software) like OpenModelica[9], are used to load these descriptions and complete the modeling of the different subsystems. Once the modeling is done, the models are exported as FMUs. These FMUs, along with the simulation configuration file previously generated are then loaded into a Co-simulation Orchestration Engine (COE) [39] where the co-simulation takes place. As a result, a trace is obtained. INTO-CPS supports the co-simulation of structural and behavioral software models described with the VDM language [20].

OpenMETA is a tool suite for component- and model-based design of engineered systems. With OpenMETA, heterogeneous models are composed together to offer a unified representation of the system. When this unified model is executed, OpenMETA seamlessly invokes external simulation tools, such as Simulink or OpenModelica to execute the composed models. Even though OpenMETA can handle multiple simulation tools, we could not find one specifically dedicated to the simulation of software models. Finally, OpenMETA also incorporates mechanisms to define testing scenarios, whose execution can be automated.

After having introduced the different tools, we now focus on analyzing their suitability for enabling model testing. Our findings are summarized in Table 1.

Efficiency (EFF). The ideal scenario to identify the most efficient tool would be to model the same CPS in all of them, and then compare the time it takes for the co-simulation process to complete in each case. However, conducting such an experiment is a significant challenge since, for example, each tool supports different modeling formalisms, with varying degrees of expressiveness, that

[5]http://site.cosimate.com/
[6]Cosimate has been left out of the analysis because of the lack of documentation and the impossibility to download the tool.
[7]http://www.esterel-technologies.com/products/scade-suite/
[8]The standard defines a common interface for interacting with FMUs. When necessary, the solver needed to simulate the model can also be encapsulated into the FMU.
[9]https://openmodelica.org/

requires specific in-depth expertise. Therefore, we have analyzed the fulfillment of this requirement by considering the strategy that each tool follows to enable model co-simulation. In this regard, we have distinguished the following strategies: using an FMU-based approach (INTO-CPS, CyPhySim), using an approach based on code generation (TASTE), or directly invoking the different simulators (OpenMETA). Since TASTE relies on code generation and therefore minimizes the overhead resulting from invoking the different simulators, we believe it is likely to better fulfill this requirement. Regarding FMU-based approaches, their efficiency is largely dependent on what is embedded in the FMUs (e.g., compiled code, model solvers, etc) and therefore may vary from case to case.

SysML-Simulink Co-simulation (CO-S). None of the evaluated tools supports model co-simulation involving software behavioral models expressed with SysML. However, INTO-CPS, TASTE and CyPhySim support co-simulation of software behavioral models specified with other modeling formalisms (Medium).

User intervention (NO-U). We consider that none of the evaluated tools fulfills this requirement. With TASTE, the user must do some manual work to complete the automatically generated code, even before the co-simulation process can be started. When it comes to the rest of the tools, they are accompanied by a graphical environment that facilitates launching the co-simulation process. However, repeating the process many times in an automated fashion requires from the user additional, manual effort (setting up third party tools, writing automation scripts, interacting with commandline interfaces, etc).

Controllability (CONT). All the tools support the faithful simulation of data flows. With TASTE, INTO-CPS and CyPhySim, it is also possible to simulate scheduling algorithms, although in the case of CyPhySim with some limitations [31]. Finally, the controllability capabilities of OpenMETA are difficult to determine, since they come from the multiple simulation tools it can integrate. However, the absence of tools specifically dedicated to the simulation of software models, suggests that OpenMETA does not fulfill this requirement.

Observability (OBS). We consider that none of the tools fulfills this requirement. TASTE's Message Sequence Charts shows sequences of operation calls without timestamps. INTO-CPS yields the data stored in the models at any point in time, plus the sequence of execution of the FMUs, but it does not include sequences of operation calls or data flows. CyPhySim follows an approach very similar to that of Simulink and requires users to model the outputs from function models that must be plotted or displayed. Finally, the data produced by OpenMETA comes from the external simulation tools it can integrate. Therefore, once again, the absence of tools specifically dedicated to the simulation of software models, suggests that OpenMETA does not fulfill this requirement as well.

After the analysis of the most promising co-simulation frameworks, we conclude that none of them is entirely suitable for enabling model testing according to our requirements. In Section 5, we describe our proposal for a co-simulation framework for the execution of testable models of CPSs. Before that, in the next section, we provide some methodological guidance on how to build these models.

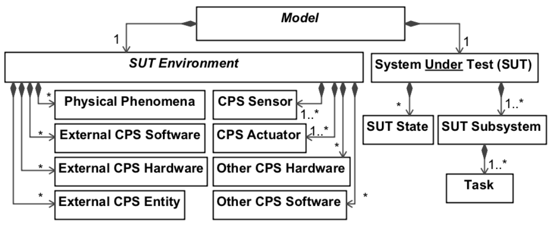


Figure 2: Information Model

4 MODELING METHODOLOGY

In this section, we present the overview of a modeling methodology, meant to be applied early during the SiL phase of development, to enable the systematic construction of testable CPS models. A complete description of the methodology can be found in [16].

Scope: The methodology covers the specification of architectural and behavioral models of a CPS, plus the integration of function models capturing hardware and environment, which are provided as an input to the modeling activity. The specification and validation of function models, which typically takes place earlier in the MiL stage of development, is out of scope. The integration process does not impose constraints on how function models are designed, other than the modeling language of choice.

Modeling Languages: SysML is used for structural and behavioral modeling of a CPS. Function models are expected to be described as Simulink models. As mentioned in Section 2, SysML and Simulink are popular in industrial settings. Besides, if needed, SysML syntax and semantics can be extended by using profiles.

Profile: This modeling methodology is supported by a profile, since the SysML's standard semantics falls short to properly model some of the aspects needed to build testable models of CPSs.

In the rest of the section, we cover the most relevant aspects of the modeling methodology, including the use of the profile.

4.1 Information Model

The information model in Figure 2 shows the concepts modeled with our methodology. The central one is the SUT, i.e., the software system under test. Internally, the SUT may be composed by a series of software subsystems, each of them responsible for running tasks, which are the basic units of behavior that can be scheduled. Also, the SUT may transition through a series of states.

As part of its normal operation, the SUT is expected to exchange data internally (among its different subsystems), and externally, with other entities within the CPS (sensors, actuators, other CPS software, etc), or outside the CPS (external hardware, human operators, etc). All these entities, regardless of whether they are external to the CPS or not, are considered to be part of the SUT environment. Entities in the SUT environment may also exchange data.

The level of detail at which the different concepts are modeled is driven by the minimum information required to enable the faithful simulation of the SUT. When modeling, we apply the following rules, to specify all the concepts from the information model:

- All concepts, except for tasks, are modeled by means of SysML blocks, fully specified within Block Definition Diagrams (BDD).
- All data exchanges among blocks are represented with connectors in SysML internal Block Diagrams (iBD) and object flows in Activity Diagrams (AD). The information conveyed in these data

Table 2: Model Testing Profile

Stereotype	Description
«SUT» [Block]	To denote the SUT block.
taskSchedulerFunction (String)	Name of the function in the generated code that will contain the scheduler's code.
timeStepSize (Integer)	Scheduler time step size in milliseconds.
«Data» [Block]	To denote data blocks.
«Configuration» [Block]	To add configuration data.
functionModelsPath (String)	Location of the function models to be executed during the simulation process.
«Matrix» [Element]	To specify attributes with a matrix data type.
numberOfRows (Integer)	Number of rows of the matrix.
numberOfColumns (Integer)	Number of columns of the matrix.
defaultValue (String)	Default values for each cell.
«Initialization» [Operation]	To denote blocks' operations to be executed only once, when the simulation starts.
Order (Integer)	Order of execution with respect to the rest of initialization operations.
«Background» [Operation]	To denote blocks' operations to be executed alongside the SUT tasks, during the simulation.
Order (Integer)	Order of execution with respect to the rest of background operations.
«Schedulable» [Operation]	To denote SUT tasks.
executionRateHz (Real)	Times per second the task must be scheduled.
executionOrder (Integer)	Position at which the task must be scheduled within the corresponding time slot.
estimatedCompletionTime (Real)	Estimated time for the task to complete.
«NotLoggable» [Action]	To denote the actions that must not be logged when generating the execution trace.

exchanges is modeled with blocks. Henceforth, we will refer to these blocks as "data blocks". Data blocks feature no behavior.

- SUT tasks are modeled as operations in the SUT subsystems' blocks. Their behavior is modeled with ADs or State Machines (SM).
- As SUT tasks, the behavior of the SUT environment blocks is also modeled with a combination of operations, ADs and SMs.
- ADs are specified by using specialized actions from the SysML specification, such as "WriteStructuralFeatureAction" or "CallOperationAction", among others.
- The way in which SysML and Simulink models interact with each other is modeled by adding opaque actions[10] to the ADs, where the behavior of SUT tasks and SUT environment blocks is described. These opaque actions can be regarded as "integration points" between SysML and Simulink models. Integration points are described in subsection 4.3.

4.2 Profile

SysML is not expressive enough to produce testable CPS models, as per our requirements. To enable the simulation process, aspects such as the scheduling of SUT tasks, or the mapping of Simulink data structures to SysML data types, must be modeled as well. Table 2 shows the stereotypes from the profile that accompanies this methodology, along with their respective tag definitions.

The «SUT» stereotype denotes the SUT block. Tag definitions provide basic information to facilitate the automatic inclusion of a task scheduler in the simulator (more about this in Section 5). The «Data» stereotype denotes data blocks. Simulink models make extensive use of matrices to store data, but SysML does not support this data type. The «Matrix» stereotype allows the specification of attributes with this data type. The «Configuration» stereotype

[10]An opaque action is a specialized action whose specification may be given in a textual concrete syntax other than SysML.

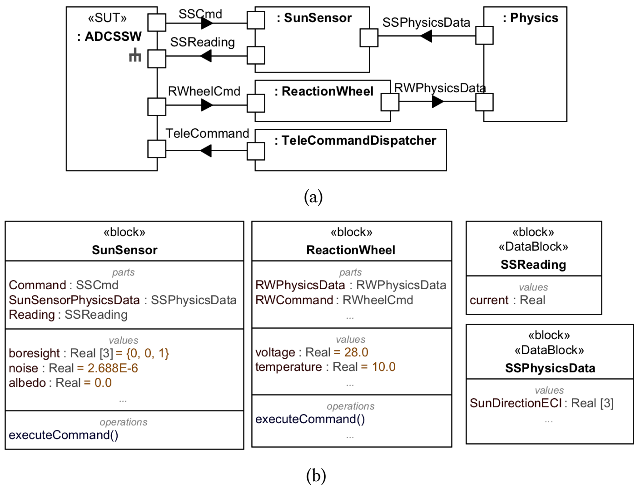


Figure 3: Excerpt of the SUT environment of the case study.

provides additional information needed to enable co-simulation. The «Schedulable» stereotype is used to indicate how SUT tasks must be scheduled for execution, and to provide estimates about their completion time. The «Background» stereotype is used to indicate which of the operations, modeling the behavior of the SUT environment blocks, must be executed in background, alongside the SUT tasks. Examples of these background operations are the ones describing the behavior of physical phenomena. Similarly, the «Initialization» stereotype allows for the specification of operations from SUT environment entities, to be executed only once, for initialization purposes. Finally, the «NotLoggable» stereotype is used to limit the verbosity of the execution trace produced when running the models.

4.3 Modeling Steps

With our methodology, the modeling process is structured in three different steps, namely: (1) specify the SUT environment, (2) specify the SUT architecture, and (3) specify the SUT behavior. They can be completed in any order, as long as the logical dependencies that might exist among them are respected.

(1) Specify the SUT environment. This step starts with the creation of one iBD[11] containing the SUT block, and one block per entity from the SUT environment. Each data flow between any pair of these blocks is then specified by linking the blocks with a connector. The connector is complemented with one data block to model the communicated information. BDDs[12] are then created to fully specify the SUT environment blocks and data blocks. Finally, the behavior of each operation added to SUT environment blocks is fully specified either with one AD or one SM.

Example. Figure 3(a) shows an excerpt of the iBD created to model the SUT environment from the case study. The SUT (ADCSSW) exchanges data with a sensor (SunSensor), an actuator (ReactionWheel), and another software system of the satellite (TeleCommandDispatcher). Sensors and actuators interact with the physical

[11]iBDs must be created in the context of an existing block. Therefore, in order to create the iBD specifying the SUT environment, a block called "model" must be created beforehand. This block will act as a container for the whole model.
[12]The methodology does not constrain the number of BDDs that can be created.

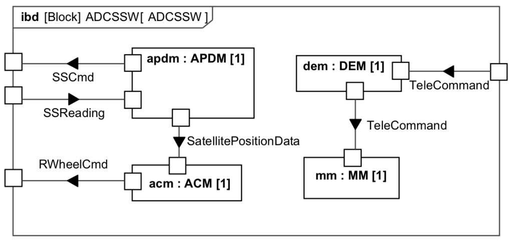


Figure 4: Excerpt of the SUT architecture of the case study

world (Physics). Figure 3(b) shows an excerpt of the BDD where some of the SUT environment blocks and data blocks are specified.

(2) Specify the SUT architecture. In the second step, the subsystems forming the SUT and the data flows among them are modeled in one iBD. Additionally, each incoming (outgoing) data flow of the SUT block modeled in step one must be linked to the SUT subsystem where the incoming (outgoing) data is consumed (produced). Finally, BDDs are created to fully specify the attributes of the different SUT subsystems and the new data blocks. In our methodology, SUT subsystems are not further decomposed into sub-subsystems. This is because a two-level decomposition into subsystems and tasks allows for modeling behavioral aspects at a sufficient level of precision to enable model testing.

Example. Figure 4 shows an excerpt of the iBD created for modeling the SUT architecture from our case study. The APDM subsystem is in charge of interacting with the sun sensor. The ACM is in charge of commanding the reaction wheel, and uses data produced by the APDM subsystem in the process. The DEM subsystem is the receptor of the telecommands sent to the SUT and forwards them to the MM subsystem.

(3) Specify the SUT behavior. After modeling the SUT architecture, we can fully specify the SUT tasks. A SUT task is specified by adding one operation stereotyped as «Schedulable» to the corresponding SUT subsystem block. If additional, auxiliary behavior must be modeled to fully describe the behavior of the subsystem then additional operations, not stereotyped, can also be added. The behavior of all the operations, whether they correspond to SUT tasks or not, is then specified by means of ADs and SMs.

Example. Figure 5(a) shows an excerpt of the BDD created for defining the tasks of some of the SUT subsystems. The APDM subsystem determines the satellite's attitude and position, and gets readings from the sun sensor. The ACM subsystem adjusts the satellite's attitude and commands the reaction wheel. Finally, the MM subsystem monitors and updates the state of the SUT, in response to the reception of telecommands. Annotations display the data gathered in the «SUT» and «Schedulable»[13] stereotypes.

Figure 5(b) shows the AD specifying the behavior of the task "getSunSensorReading" from the APDM subsystem. Following the control flow[14] (highlighted in the figure), it can be seen that, first, a command with the opcode "SSTM" is created. This is the command the sun sensor executes to produce the readings. The command is

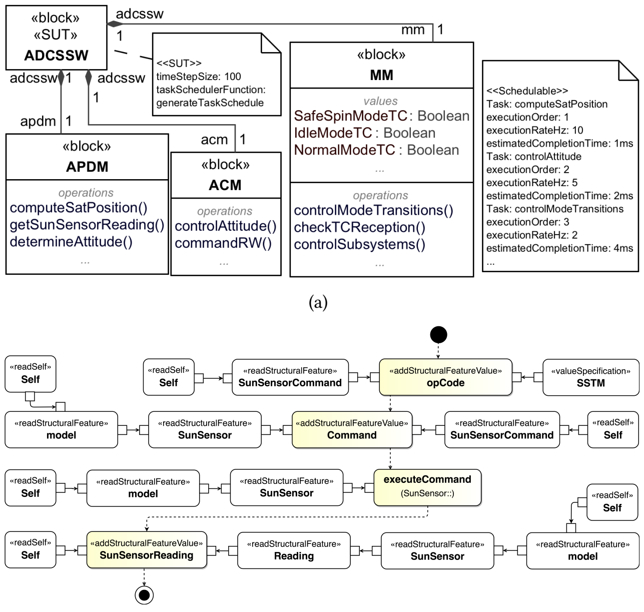


Figure 5: Excerpt of the SUT behavior of the case study.

then sent to the sun sensor. Once sent, the operation "executeCommand" in the sun sensor is invoked. Finally, the readings stored in the sun sensor, after executing the command, are retrieved.

4.4 Integration of Function Models

To enable co-simulation, the co-simulation framework needs information on how SysML and Simulink models interact. Our methodology allows for the use of opaque actions in ADs, to define integration points indicating when the control flow must be handed over to a given Simulink model. To specify these opaque actions, the utilization of structured text, with a specific and defined syntax is enforced. This text is organized in the following sections:

Operation: To indicate the operation being executed. For now, only the keyword "ExecuteExternalModel" is supported.

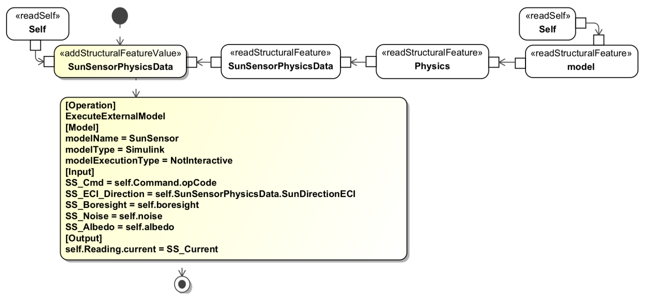
Model: To specify values for the properties "modelName", "modelType" and "modelExecutionType". For now, "modelType" must be fixed to "Simulink". The execution type can be "NotInteractive", "InteractiveAndSynchronized" and "InteractiveAndNotSynchronized". The execution of Simulink models is discussed in Section 5.

Input: To specify input parameter values for the model, expressed as assignment statements. The left side represents the name of the parameter in the Simulink model. The right side represents the name of the attribute in the SysML block, from which the value is retrieved at runtime.

Output: To specify the values retrieved from the Simulink model, expressed as assignment statements. The left side represents the name of the attribute in the SysML block where the value will be stored. The right side represents the name of the output parameter in the Simulink model.

Example. Figure 6 shows the AD specifying the behavior of the operation "executeCommand" from the sun sensor. The behavior of the sun sensor is modeled in a Simulink model called "SunSensor", that must be executed to obtain the sun sensor readings. In order

[13] Tagged values of some of the tasks are omitted for brevity.
[14] The control flow determines the ordered sequence of actions to be executed. These actions have parameters whose values must be provided. The purpose of the non-highlighted actions in the AD is to indicate how the parameter values are obtained. In this regard, "Self" makes reference to the APDM block, and "model" to the block that contains the whole model.



```
[Operation]
ExecuteExternalModel
[Model]
modelName = SunSensor
modelType = Simulink
modelExecutionType = NotInteractive
[Input]
SS_Cmd = self.Command.opCode
SS_ECI_Direction = self.SunSensorPhysicsData.SunDirectionECI
SS_Boresight = self.boresight
SS_Noise = self.noise
SS_Albedo = self.albedo
[Output]
self.Reading.current = SS_Current
```

Figure 6: "executeCommand" operation: "SunSensor" block

to do that, an opaque action was added to the AD. When the AD is executed, and the opaque action is reached, control is handed over to Simulink to run the "SunSensor" model. In the process, as indicated in the [Input] section, information from the SysML model (self.albedo, self.noise, etc) is used to set the parameter values of the Simulink model (SS_Albedo, SS_Noise, etc). When the execution of the Simulink model finishes, control is returned to the AD, along with the sun sensor readings (SS_Current), that are then transferred to the SysML model (self.Reading.current), as indicated in the [Output] section.

5 MODEL EXECUTION

We now introduce our approach for the automatic generation of a SysML-Simulink co-simulation framework, with the aim of fulfilling the requirements elicited in Section 2. Henceforth, we will refer to this framework as the CPS simulator, or simply as the simulator.

Figure 7 shows an overall description of the approach. SysML models are passed as an input to a code generator that produces the C++ code of the simulator. This code replicates the behavior specified in the SysML models, and thanks to the data stored in the integration points, it is capable of invoking the execution of the corresponding Simulink models through the MatLab runtime. To start the simulation process, the test cases invoking the simulator must provide some parameters. They include the specification of the initial status of the CPS and its environment, the simulation time span, and the list of external events that must be triggered during the simulation. Finally, as a result of the simulation, an execution trace is produced. In contrast to fUML [27]-based model execution engines like Papyrus Moka[15] or the Cameo Simulation Toolkit[16], using a SysML code generator enables us to fulfill all of our requirements such as the precise scheduling of tasks, efficient test execution, or the generation of adequate execution traces, as further discussed below. Similar to SysML, though code generation from Simulink models would a priori be more efficient, we choose to execute these models through the MatLab runtime. However, in practice, it is common for such models to include blocks provided by external suppliers, as in our case study for some sensors. As a result, code generation may not be complete and additional external libraries must be integrated with the generated code, leading to significant effort overhead.

The code snippet in Listing 1 shows how the CPS simulator operates internally. The simulator works by emulating the passing of

[15]https://www.eclipse.org/papyrus/index.php
[16]https://www.nomagic.com/product-addons/magicdraw-addons/cameo-simulation-toolkit

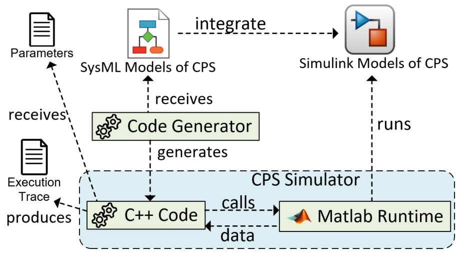


Figure 7: Overview of the co-simulation approach

```
1  string Simulator::run(Status *initStatus, long duration, vector<
       ↪ eventData> eventList) {
2    int time = 0;
3    setSimulatorStatus(initialStatus);
4    this->log->logSimulatorStatus(time, this);
5    this->runInitializationOperations();
6    for (int tStep = 0; tStep < duration; ++tStep) {
7      time = tStep * this->tStepSize;
8      this->triggerEvents(eventList, time);
9      this->runBackgroundOperations();
10     TaskSchedule ts = this->SUTClass->generateTaskSchedule(tStep);
11     vector<TaskData>::iterator task;
12     for (task = ts.list.begin(); task < ts.list.end();  ++task) {
13       this->log->generateLogEntry(time, "Task␣Start", task);
14       task->taskImpl();
15       time += task->estimatedCompletionTime;
16       this->log->generateLogEntry(time, "Task␣End", task);
17     }
18   }
19   return this->log->getLog();
20 }
```

Listing 1: Simulation loop

physical time (lines 2, 6, 7 and 15), henceforth simulated time. This is done with the help of the information provided in the SysML models (the time step size specified in the «SUT» stereotype, and the estimated time of completion of each SUT task, specified in the «Schedulable» stereotype). First, the initial status of the system (CPS and environment) is set (line 3). Then, the initialization operations are executed (line 5). After this, at each time step, the simulator handles the triggering of external events (line 8); executes the background operations (line 9); gets the list of SUT tasks scheduled for execution in that time step (line 10); and executes them (lines 12 and 14). Throughout the code there are calls to the class responsible for the generation of the execution trace (lines 4, 13 and 16), which is returned at the end.

In what follows, we describe some relevant aspects of the code generation process, the scheduling of tasks, the SysML-Simulink co-simulation process, and how external events are handled.

5.1 Basic aspects of the code generator

SysML blocks, attributes and operations are translated into C++ classes, attributes and methods in a straightforward manner. When it comes to behavioral aspects, the translation of ADs is facilitated by the utilization of SysML specialized actions, whose semantics are very close to the equivalent C++ constructs. Examples of these are translating "readSelfActions", used to access the context of the AD, into the "this" C++ keyword, or translating "writeStructuralFeatureValueActions" into assignment statements. The translation of the rest of standard actions, with the exception of opaque actions which are discussed further below, is conducted in a similar way. SMs are translated by applying the state design pattern [13].

```
struct TaskData {
  double estimatedCompletionTime;
  double executionRate;
  int executionOrder;
  std::function<void(void)> taskImpl;
};

struct TaskSchedule {
  int timeStep;
  std::vector<taskData> taskList;
};
```

Listing 2: Data structures for the SCS

The events triggering the transitions in the SM are represented as Boolean variables. The generation of the code responsible for producing an execution trace is based on the analysis of the architectural and behavioral elements of the model. The architectural information is used to print timestamped[17] snapshots of the CPS and its environment. The code generated out of the ADs and SMs adds timestamped entries to the execution trace, logging what data is read or written, what internal operations or Simulink models are invoked, or how the system transitions from one state to another.

5.2 Task Scheduling

Our code generator produces simulators of CPSs running based on a Static Cyclic Scheduler (SCS) [24], a type of scheduler commonly used in the development of embedded systems. With a SCS, the task schedule is computed statically, prior to executing the system. To compute it, the simulated time is divided into slots, and tasks, which are expected to be executed periodically, are assigned to them depending on their period.

Generating code for simulating a SCS is a two-step process. First, two data structures (Listing 2) are created: “TaskData”, for storing what is needed to execute a given task (tagged values from the «Schedulable» stereotype, plus a pointer “taskImpl” to the class method with the task’s implementation), and “TaskSchedule”, for storing the task schedule at a given time step. In the second step, described in the next example, a method is added to the SUT class to produce the actual task schedule.

Example. We return here to the excerpt of the system from our case study, introduced in Figure 5(a). With the tagged values of the «SUT» and «Schedulable» stereotypes displayed in the annotations, the code generator adds a method called “generateTaskSchedule" (Listing 3) to the class ADCSSW. This method is repeatedly called by the simulator (line 10 in Listing 1), to generate the task schedule at each time step. When this occurs, the elapsed time is calculated (line 4), and then, the same procedure is repeated for each task: the task’s period is computed (lines 8, 14 and 20) and, if the elapsed time is a multiple of it (lines 9, 15 and 21), then the task is added to the list of scheduled tasks (lines 12, 18 and 24), after having set its estimated time of completion (lines 10, 16 and 22), and the pointer to the task’s implementation (lines 11, 17 and 23). Finally, the data structure containing the list of scheduled tasks is returned.

5.3 SysML-Simulink Co-Simulation

When an integration point is added, every time the C++ code of the corresponding AD is executed in the simulator, the Simulink model specified in the integration point is executed, according

```
1  TaskSchedule ADCSSW::generateTaskSchedule(int timeStepNumber) {
2    TaskData t;
3    long taskPeriod;
4    long currentTime = timeStepNumber * 100;
5    TaskSchedule ts;
6    ts.timeStep = timeStepNumber;
7    ts.taskList.reserve(3);
8    taskPeriod = (long) (1.0 / 10.0 * 1000);
9    if (currentTime % taskPeriod == 0) {
10     t.estimatedCompletionTime = 1.0;
11     t.taskImpl = std::bind(&APDM::computeSatPosition, this->apdm);
12     ts.taskList.push_back(t);
13   }
14   taskPeriod = (long) (1.0 / 5.0 * 1000);
15   if (currentTime % taskPeriod == 0) {
16     t.estimatedCompletionTime = 2.0;
17     t.taskImpl = std::bind(&APDM::determineAttitude, this->apdm);
18     ts.taskList.push_back(t);
19   }
20   taskPeriod = (long) (1.0 / 2.0 * 1000);
21   if (currentTime % taskPeriod == 0) {
22     t.estimatedCompletionTime = 4.0;
23     t.taskImpl = std::bind(&MM::controlModeTransitions, this->mm);
24     ts.taskList.push_back(t);
25   }
26   return ts;
27 }
```

Listing 3: Task schedule generation

to the value of the “modelExecutionType" attribute (“Interactive-AndSynchronized", “InteractiveAndNotSynchronized" and “NotInteractive"). This is done with the help of an automatically generated helper class, which interacts with the MatLab runtime, where the execution of the Simulink model takes place. In what follows, after providing some generic background on how the MatLab runtime executes Simulink models, we describe the most complex co-simulation scenario, corresponding to the “InteractiveAndSynchronized" case. The other cases are simpler ones.

Generalities about the execution of Simulink models. We refer to the data stored in Simulink blocks[18] as the model state. The execution of a Simulink model consists in the computation of its state at successive, discrete points in time, over a given time span. To execute a Simulink model, the MatLab runtime emulates the passing of time, henceforth simulated time, and uses a numerical solver tool, to conduct the computations of the model state. The amount of simulated time elapsed between two consecutive computations is known as the time step size[19]. As described below, both the time step size and the model state, play a key role in how the CPS simulator interacts with the MatLab runtime, to properly handle the co-simulation process.

Interactive and synchronized co-simulation. In many Simulink models, the computation of the model state, at a given point in time, depends on the previously computed model state and on the time step size, which, in its turn, represents a certain amount of time elapsed in the physical world. As an example of this, in our case study, the computation of the satellite’s position, at a given point in time, is based on the previously calculated position, and on the amount of time elapsed since then. With this type of models, in order to properly handle the co-simulation process, the simulated time inside the MatLab runtime must be as synchronized as possible with the simulated time in the CPS simulator. The goal here is to make possible for SUT tasks in the CPS simulator to retrieve the state information from function models, at the time step

[17]With simulated time.

[18]Not to be confused by SysML blocks.

[19]MatLab has two kinds of solvers : fixed-step and variable-step solvers. The description here applies to fixed-step solvers.

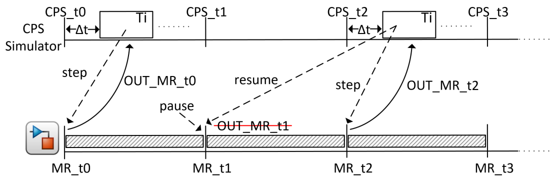


Figure 8: SysML-Simulink co-simulation

that is the closest to when the tasks are scheduled for execution. Figure 8 illustrates the process. The horizontal lines represent the simulated time in the CPS simulator (on top) and in the MatLab runtime (at the bottom). Variables on each correspond to the same points in time (CPS_t$_0$ = MR_t$_0$, CPS_t$_1$ = MR_t$_1$, CPS_t$_2$ = MR_t$_2$), and the simulation time span is the same in both cases. In the figure, a task, T$_i$, scheduled for execution at CPS_t$_0$ + Δt, in the first time step, invokes the step by step execution of a Simulink model. When this occurs, the MatLab runtime computes the model state corresponding to the first discrete point in time MR_t$_0$, yields the corresponding output OUT_MR_t$_0$, and then pauses the execution, before the computation of the next model state takes place. Due to the time step size in the Simulink model, it is not possible to obtain an output at MR_t$_0$ + Δt = CPS_t$_0$ + Δt, when T$_i$ is really executed. Since the output closest in time to CPS_t$_0$ + Δt that the MatLab runtime can yield is OUT_MR_t$_0$, this is the one the CPS simulator retrieves. After this, the execution of the Simulink model remains paused until T$_i$ is scheduled for execution again. When this occurs at CPS_t$_2$ + Δt, in the third time step, the execution of the Simulink model is resumed. Since the next point in time is MR_t$_1$, the MatLab runtime produces the corresponding output OUT_MR_t$_1$. However, MR_t$_1$ is not close in time to CPS_t$_2$ + Δt. The output closest in time to CPS_t$_2$ + Δt that the MatLab runtime can offer is OUT_MR_t$_2$. Therefore, the CPS simulator advances the execution of the Simulink model, ignoring the outputs produced (crossed out in the figure), until MR_t$_2$ is reached[20]. Once OUT_MR_t$_2$ is computed, the CPS simulator retrieves the corresponding output, and the execution of the Simulink model is paused again. The process is then repeated in subsequent executions of T$_i$.

In our case study, the time step size for SUT tasks is the same in Simulink and in the CPS simulator. Hence, in the scenario above, the SUT tasks can run in a fully synchronized way in Simulink and the CPS simulator. However, the SUT tasks interact with some Simulink model blocks representing hardware and environment entities, and these model blocks have a time step size that is a fraction of the time step size of the SUT tasks. Therefore, we use the mechanism above mainly to coordinate execution of SUT tasks and Simulink blocks representing hardware and physical entities.

5.4 Handling external events

As part of its normal operation, a CPS will typically have to react to external events, triggered at different moments in time. For example, in our case study, the ADCS must react to the reception of telecommands. As stated in Subsection 5.1, the simulator represents these events with Boolean variables. The triggering of a given event is reflected by setting to “true" the corresponding variable. In order to do this effectively, the simulator must receive from the test case, the list of events to be triggered during the simulation, along with the precise point in simulated time when that must occur. As shown in Figure 9, during the simulation, at the beginning of each time step, the simulator uses this information to update the status of the different event variables. Once one of them is enabled, the corresponding event is handled, as soon as the task responsible for it is scheduled for execution.

[20]To do this, the CPS simulator uses the helper class to keep track of the elapsed time in the MatLab runtime, and the time step size of the Simulink model.

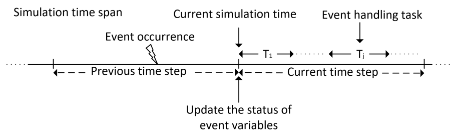


Figure 9: External Event handling

5.5 Requirements fulfillment

After discussing in detail our co-simulation framework, we explain how it fulfills the requirements elicitated in Section 2.

Efficiency (EFF): With our approach, performance bottlenecks are due to the execution of Simulink models. However, the results obtained when evaluating the CPS simulator of the case study (see Section 6) show that our approach is sufficiently efficient to be suitable to support model testing.

SysML-Simulink Co-simulation (CO-S): Thanks to the integration points in SysML models, and the automatic generation of a helper class that handles the interaction with the MatLab runtime, our co-simulation approach supports the integrated execution of SysML and Simulink models.

No User Intervention (NO-U): The integrated execution of Simulink models is conducted automatically. Also, external events can be defined and passed as a parameter to the simulator before the execution starts. All these allow for the automated end-to-end simulation of CPSs.

Controllability (CONT): Our modeling methodology facilitates the construction of testable models of CPSs at the level of precision required for model testing. These models are then faithfully executed by the automatically generated CPS simulator, including the precise scheduling of system tasks, and the data flows among the SUT's subsystems, and between the SUT and its environment.

Observability (OBS): Generating an execution trace during the simulation enables the observation, at an adequate level of detail and with timestamps, of triggered events, scheduled tasks, and data flows, among other types of information. Such traces can then be used to check whether the model executions comply with expected, relevant properties concerning various aspects of the system.

6 EVALUATION

In this section, we present the result of evaluating whether our modeling and co-simulation approach is suitable for the practical application of model testing. The evaluation has been conducted on the LuxSpace case study introduced in Section 2.

6.1 Research Questions (RQs)

RQ1: *How long does it take for the CPS simulator to execute test cases?* Test cases characterize the simulation scenarios (initial status of the CPS and its environment, simulation time span, events to be triggered) to be executed on the CPS simulator. We expect a close relationship between the simulation time span defined in a test case, and the amount of time needed for the CPS simulator to run it. Therefore, the shorter the simulation time span, the more test cases the simulator should be able to execute in a given period of time. On the other hand, every time a test case is run, the MatLab runtime must initialize the Simulink models involved in the simulation, which might cause some significant overhead when running a large number of test cases. To answer this RQ, we have set up an experiment to evaluate these factors.

RQ2: *Is our modeling methodology applicable for realistic CPSs?* Other than model size, which is determined by the system, two aspects impact directly the modeling effort needed to produce testable CPS models with our approach: the level of precision at which behavior is specified, and the integration of Simulink models. We provide effort estimates and discuss these two factors.

6.2 Experiment Design

To analyze how the simulation time span impacts the performance of the simulator, we have prepared an experiment consisting in the execution of two realistic test suites, both with an aggregate simulation time span of 100 000 seconds. The first test suite contains a small number (20) of test cases with a long simulation time span of 5000 seconds, the amount of time approximately needed for the satellite of the case study to complete one orbit around earth. The second one contains a large number (1000) of test cases, each with a much shorter simulation time span of 100 seconds. Even though the aggregate simulation time span is the same in both test suites, we expect to see additional overhead in the execution of the second one, due to the much larger number of test cases that are run.

When it comes to the strategy followed to generate these test cases, since the objective here is to report on the performance of the simulator, we decided to go with a simple one, consisting in randomly generating test cases to achieve path coverage in the ADs of the LuxSpace SysML models. In the future, though, we plan on using more sophisticated, search-based [25] test case generation techniques to conduct model testing of CPSs.

Finally, to mitigate the impact of random variations on the validity of the experiment, each test suite was run 10 times and average values for the different measurements were computed.

6.3 Experiment Results

RQ1. Table 3 summarizes the results obtained from conducting the experiment. Running 100 000 seconds (≈ 27.7 hours) of simulated time required between 29 688 to 34 771 seconds (≈ 8.2 to 9.7 hours), that is, the ratio of execution time to simulated time is roughly between 0.3 and 0.4. Though the main driver of execution time is simulated time, there is approximately 1.5 hours of extra time needed to complete the execution of the second test suite. This confirms that the initialization of Simulink models produces a significant overhead when a larger number of test cases is run.

When it comes to evaluating the practicality of the approach, the results show that CPS models, such as the realistic one we developed for our case study, can be verified overnight, every day. Furthermore, the efficiency of the verification process could be increased even more by rerunning only the test cases affected by model changes, or by parallelizing the process on multiple cores, since test cases can be run independently from each other. In conclusion, we consider that our co-simulation framework is efficient enough to enable model testing of CPSs in practical time.

Table 3: Experiment results

	Test Suite 1	**Test Suite 2**
Number of test cases (TC)	20	1000
Simulation time span per TC	5000s	100s
Simulink models initialization time per TC	5.198s	4.337s
Aggregate simulation time span (ST)	100 000s	100 000s
Total execution time (ET)	29 688s	34 771s
Ratio of ET to ST	0.297	0.348

Settings: Computer: Intel i7-4870HQ 2.5 GHz 16 GB RAM. OS: Windows 10 64 bits. MatLab version: R2010a

RQ2. The SysML model specifying the architecture and behavior of the CPS from the case study is of manageable size: 40 blocks specified in 6 BDDs and 2 iBDs, 30 ADs, and 1 SM. Building this model required an effort of approximately 2 man-months, plus an additional man-week to train LuxSpace control engineers. Among the different tasks conducted towards completing the model, two of them stood out as the most laborious ones, taking approximately 60 percent of the total effort: integrating Simulink models within the SysML model, and specifying software behavioral aspects. The former required a joint effort with LuxSpace control engineers to fully specify the integration points with Simulink. Particularly laborious was the analysis of Simulink models, to match their numerous input and output parameters with SysML blocks' attributes. In general, communication between software engineers and control and mechanical engineers developing Simulink models can be expected to be a source of confusion requiring special attention. When it comes to the second task, utilizing a model execution approach based on code generation requires behavioral aspects to be specified at a high level of precision to reduce the complexity of the code generation process. Since models are built once and then updated over the life time of a system, and given the criticality of most CPSs and the fact they often need to be certified based on design models, we consider that the cost of our modeling methodology is acceptable.

7 CONCLUSIONS

The model testing of cyber-physical systems (CPSs) aims at detecting design errors and critical scenarios early in the development process, based on heterogeneous models of the software, hardware, and physical environment, thus alleviating the challenges faced with CPSs. We presented a SysML-based modeling methodology and a SysML-Simulink co-simulation framework to enable such testing. The evaluation conducted on a satellite case study shows that they facilitate the systematic construction of testable CPS models and enable their execution in practical time, among other pre-defined requirements.

In the future, we plan on combining our approach with more sophisticated techniques for the automatic generation of test cases and test oracles, in order to conduct more effective testing.

ACKNOWLEDGMENTS

We gratefully acknowledge the funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277) and from the Fonds National de la Recherche, Luxembourg (FNR 11606543).

REFERENCES

[1] An-jelo Gian C. Abad, Lady Mari Faeda G. Guerrero, Jasper Kendall M. Ignacio, Dianne C. Magtibay, Mark Angelo C. Purio, and Evelyn Q. Raguindin. 2015. A simulation of a power surge monitoring and suppression system using LabVIEW and multisim co-simulation tool. In *Proceedings of the 8th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management, HNICEM 2015*. 1–3.

[2] Ahmad T. Al-Hammouri. 2012. A Comprehensive Co-simulation Platform for Cyber-physical Systems. *Computer Communications* 36, 1 (December 2012), 8–19.

[3] Daniele Antonioli and Nils Ole Tippenhauer. 2015. MiniCPS: A Toolkit for Security Research on CPS Networks. In *Proceedings of the 1st ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy, CPS-SPC 2015*. 91–100.

[4] D. Bian, M. Kuzlu, M. Pipattanasomporn, S. Rahman, and Y. Wu. 2015. Real-time co-simulation platform using OPAL-RT and OPNET for analyzing smart grid performance. In *Proceedings of the IEEE Power & Energy Society General Meeting 2015*. 1–5.

[5] Mary Bone and Robert Cloutier. 2009. The current state of model based systems engineering: Results from the OMG SysML request for information 2009. In *Proceedings of the 8th Conference on Systems Engineering Research*. 225–232.

[6] T. Brezina, Z. Hadas, and J. Vetiska. 2011. Using of Co-simulation ADAMS-SIMULINK for development of mechatronic systems. In *Proceedings of the 14th International Conference Mechatronika*. 59–64.

[7] Lionel C. Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. 2016. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 789–792.

[8] Christopher Brooks, Edward A. Lee, David Lorenzetti, Thierry S. Nouidui, and Michael Wetter. 2015. CyPhySim: A Cyber-physical Systems Simulator. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015*. 301–302.

[9] W. Stuart Dols, Steven J. Emmerich, and Brian J. Polidoro. 2016. Coupling the multizone airflow and contaminant transport software CONTAM with EnergyPlus using co-simulation. *Building Simulation* 9, 4 (August 2016), 469–479.

[10] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (January 2003), 127–144.

[11] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional.

[12] J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock. 2015. Cyber-Physical Systems Design: Formal Foundations, Methods and Integrated Tool Chains. In *Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015*. 40–46.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

[14] H. Georg, S. C. Müller, C. Rehtanz, and C. Wietfeld. 2014. Analyzing Cyber-Physical Energy Systems:The INSPIRE Cosimulation of Power and ICT Systems Using HLA. *IEEE Transactions on Industrial Informatics* 10, 4 (November 2014), 2364–2373.

[15] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2017. Co-simulation: State of the art. *CoRR* abs/1702.00686 (February 2017). arXiv:1702.00686

[16] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. 2018. A SysML-Based Approach for Model Testing of Cyber-Physical Systems. Tech. Rep. TR-SNT-2018-2. SnT Centre - University of Luxembourg.

[17] Siddhartha Kumar Khaitan and James D. McCalley. 2015. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal* 9, 2 (June 2015), 350–365.

[18] X. Koutsoukos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, and J. Sztipanovits. 2018. SURE: A Modeling and Simulation Integration Platform for Evaluation of Secure and Resilient Cyber-Physical Systems. *Proc. IEEE* 106, 1 (January 2018), 93–112.

[19] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh. 2016. Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In *Proceedings of the 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS, CPS Data 2016*. 1–6.

[20] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, Shin Sahara, Marcel Verhoef, Peter W. V. Tran-Jørgensen, Tomohiro Oda, and Paul Chisholm. 2018. VDM-10 Language Manual.

[21] E. Lee and S. Neuendorffer. 2004. Classes and subclasses in actor-oriented design. In *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE 2004*. 161–168.

[22] E. A. Lee, M. Niknami, T. S. Nouidui, and M. Wetter. 2015. Modeling and simulating cyber-physical systems using CyPhySim. In *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2015*. 115–124.

[23] Edward Ashford Lee and Sanjit Arunkumar Seshia. 2017. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach, 2nd Edition*. MIT Press.

[24] Jane W. S. Liu. 2000. *Real-Time Systems*. Prentice Hall.

[25] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Software Testing, Verification & Reliability* 14, 2 (June 2004), 105–156.

[26] Modelica Association. 2014. Functional Mock-up Interface for Model Exchange and Co-Simulation Version 2.0. https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf

[27] OMG. 2017. Semantics of a Foundational Subset for Executable UML Models (fUML) 1.3 Specification. http://www.omg.org/spec/SysML/1.5/

[28] OMG. 2017. Systems Modeling Language Specification (SysML). Version 1.5. http://www.omg.org/spec/SysML/1.5/

[29] Ioannis Papaefstathiou, Gregory Chrysos, and Lambros Sarakis. 2015. COSSIM: A Novel, Comprehensible, Ultra-Fast, Security-Aware CPS Simulator. In *11th International Symposium on Applied Reconfigurable Computing, ARC 2015 (Lecture Notes in Computer Science)*, Vol. 9040. Springer, 542–553.

[30] Maxime Perrotin, Eric Conquet, Julien Delange, André Schiele, and Thanassis Tsiodras. 2011. TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future. In *15th International SDL Forum: Integrating System and Software Modeling, SDL 2011 (Lecture Notes in Computer Science)*, Vol. 7083. Springer Berlin Heidelberg, 26–37.

[31] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org. http://ptolemy.org/books/Systems

[32] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John A. Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference, DAC 2010*. 731–736.

[33] F. Schloegl, S. Rohjans, S. Lehnhoff, J. Velasquez, C. Steinbrink, and P. Palensky. 2015. Towards a classification scheme for co-simulation approaches in energy systems. In *Proceedings of the 2015 International Symposium on Smart Electric Distribution Systems and Technologies, EDST 2015*. 516–521.

[34] Hesham Shokry and Mike Hinchey. 2009. Model-Based Verification of Embedded Software. *IEEE Computer* 42, 4 (April 2009), 53–59.

[35] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. 2014. OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems. In *From Programs to Systems. The Systems perspective in Computing. ETAPS 2014 (Lecture Notes in Computer Science)*, Vol. 8415. Springer Berlin Heidelberg, 235–248.

[36] ITU-T: Telecommunication and Standardization Sector of the International Telecommunication Union. 1999. Specification and Description Language (SDL): Reference Manual. https://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf

[37] ITU-T: Telecommunication and Standardization Sector of the International Telecommunication Union. 2002. Abstract Syntax Notation One (ASN.1): Specification of basic notation. https://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

[38] ITU-T: Telecommunication and Standardization Sector of the International Telecommunication Union. 2011. Message Sequence Chart (MSC): Recommendation ITU-T Z.120. https://www.itu.int/rec/T-REC-Z.120-201102-I

[39] Casper Thule. 2016. Verifying the Co-Simulation Orchestration Engine for INTO-CPS. *CEUR Workshop Proceedings* 1744 (November 2016).

[40] B. Wang and J. S. Baras. 2013. HybridSim: A Modeling and Co-simulation Toolchain for Cyber-physical Systems. In *Proceedings of the 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2013*. 33–40.