software verification & validation

securityandtrust.lu

A SysML-Based Methodology for Model Testing of Cyber-Physical Systems

Carlos A. González, Mojtaba Varmazyar, Shiva Nejati and Lionel Briand
Software Verification and Validation Department, SnT Centre
University of Luxembourg, Luxembourg

Yago Isasi Parache
LuxSpace, Luxembourg

Version 1.0

May 9, 2018

Technical Report Number: TR-SNT-2018-2

UNIVERSITÉ DU LUXEMBOURG

Contents

1 Introduction

A Cyber-Physical System (CPS) is an integration of computation with physical processes [2]. CPSs are characterized by the presence of a, potentially large number of embedded computing and communication devices, which monitor and control the physical world through sensors and actuators. These devices interact and exchange data with each other through networks, that might be potentially extended over a large geographical area. CPSs are already present in sectors such as transportation, energy, medicine, space or manufacturing, and they are expected to become ubiquitous in a near future. CPSs have the potential to revolutionize how we construct and operate engineered systems, thus transforming the way in which we interact with the physical world. Applications of CPSs are expected to have an enormous impact on society and the economy.

Considering the expected impact and pervasiveness of CPSs, ensuring their dependability is a matter of the utmost importance. Validation and Verification (V&V) is essential to ensure dependability of software systems. Among all V&V techniques, the most prevalent one is testing. Unfortunately, aspects that characterize the behavior of a CPS, such as the continuous and complex interactions with the physical world, or the deep intertwining of hardware and software, turn the testing of these systems into a highly expensive, time-consuming process, at best, or make testing infeasible, at worst. These challenges in the application of existing testing techniques, render CPSs untestable in practice.

To enable testing of untestable systems, such as CPSs, model testing [1] was recently proposed as a vision, in which the key idea is to frame testing on models rather than operational systems. The goal of model testing is to raise the level of abstraction of testing from operational systems to models of their behaviors and properties. The models that underlie model testing should be executable representations of the relevant aspects of a system and its environment. It is expected that model testing would bring early and cost-effective automation to the testing of many critical systems that defy existing testing techniques, thus significantly improving their dependability.

This document is a joint study by LuxSpace SARL (LuxSpace) and the Software Verification and Validation department (SVV) from the SnT Centre at the University of Luxembourg on how to enable model testing of CPSs. It starts by eliciting the requirements that must be fulfilled to conduct model testing in an effective manner. Once the requirements have been elicited, they are used to both identify what modeling language and modeling tool are most suitable for the job, and specify a modeling methodology for the creation of executable, model testing-enabled models. The study is complemented with the application of the modeling methodology to a satellite's attitude control system currently being developed by LuxSpace.

2 Requirements to enable model testing of CPSs

In order to conduct model testing of a CPS, the first step is to create one or more models gathering all the relevant[1] aspects about the CPS and its environment. Once created, those models will then be executed as part of the testing process. These models, along with the tools and artifacts used to create them and execute them, as well as the testing process itself, must feature certain characteristics, in order to be considered valid for conducting model-testing, or "model testing-enabled". In the rest of this section, we present what is required from models, modeling languages, modeling tools, model execution tools, and the testing process, in order to be considered suitable for model testing.

2.1 R1: Requirements about the model(s)

(a) **Structural information**: Model(s) shall describe the internal structure of the CPS. That is, its software and hardware units; the interfaces among them, and with elements external to the CPS; as well as any internal data they manipulate.

(b) **Behavioral information**: Model(s) shall describe the internal behavior of the CPS. That is, the abstract states of the system; the transitions among these states and their triggering events; the algorithms or mathematical computations characterizing its software and hardware units; and, in general, any other aspect related to the way they are executed, such as execution rates, scheduling policies, etc.

(c) **Environmental information**: Model(s) shall describe the external environment that can impact or be impacted by the execution of the CPS. This might include engineered systems, external to the CPS, or natural phenomena.

(d) **Uncertainty**: Model(s) shall describe any uncertainties that could impact the behavior of the CPS, e.g., accuracy issues with sensor readings, unreliable data transmission channels, rounding error propagation in mathematical calculations, etc.

2.2 R2: Requirements about the modeling language(s)

(a) **Expressiveness**: The modeling language(s) shall be expressive enough to describe all the information needed to specify model testing-enabled models.

(b) **Extensibility**: The modeling language(s) shall feature mechanisms to extend their semantics, if deemed necessary to build model testing-enabled models.

[1] Only those aspects that are of relevance for the testing activity should be taken into account during the modeling stage.

(c) **Industrial popularity and acceptance**: The modeling language(s) shall be widely accepted by the industry and have comprehensive, high-quality tool support.

2.3 R3: Requirements about the modeling tool

(a) **Language support**: The modeling tool shall provide complete support for the modeling language features required to create model testing-enabled models.

(b) **Model execution support**: The modeling tool shall provide support for model execution either through a model execution engine, or by facilitating mechanisms to enable code generation.

2.4 R4: Requirements about model execution

(a) **Co-Simulation**: The model execution tool shall support the integrated execution, i.e. co-simulation, of software models and function models. Henceforth, we will refer to this model execution tool as the co-simulation framework.

(b) **Efficiency**: The co-simulation process shall be efficient enough to facilitate running thousands of test cases within practical time.

(c) **No user intervention**: The co-simulation framework shall be capable of automatically running series of co-simulations without user intervention, to facilitate the automation of the testing process.

(d) **Observability**: As a result of the co-simulation process, the co-simulation framework shall provide an execution trace containing "snapshots" of the CPS at each step of the execution process, plus any data relevant to steer the testing activity, e.g., external events received; data gathered/produced by the different CPS subsystems; current state of the CPS and its internal units; internal algorithms or mathematical/physical calculations that have been scheduled for execution, or just executed (along with their inputs and outputs); simulation time, etc. The amount, level of detail and nature of the data collected in the execution trace shall be configurable according to the testing goals.

(e) **Controllability**: 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. This requires the precise simulation of the scheduling of system tasks, and the data flows among subsystems and between the system and its environment.

2.5 R5: Requirements about test automation

(a) **Test case generation**: The generation of test cases shall be guided by the automatic analysis of the data collected in the execution traces produced by the model execution tool.

(b) **Test oracles**: Test oracles shall be able to automatically check whether a test case has passed or not, and do so by analyzing the corresponding execution trace. This analysis shall allow oracles to evaluate temporal properties (e.g., order in the execution of tasks), timing properties (e.g., task execution rates), state properties (e.g., illegal state transitions), data properties (e.g., correctness of data sent/received by a subsystem) and, in general, any other type of property relevant for checking the corresponding test case.

3 Modeling and specification of CPSs

Once the requirements have been specified, it is the time to start with the modeling activity. We begin by selecting a modeling language and a modeling tool for the job. After that, we justify the need for a modeling methodology, and overview our proposal. In the rest of subsections, this modeling methodology is described in detail.

3.1 Selection of the modeling language and tool

The modeling language selected for the modeling and specification of CPSs is the Systems Modeling Language (SysML) [3]. The election of SysML is motivated by the following facts:

1. SysML supports the specification of complex systems formed by the combination of hardware, software, information, people, facilities, etc. and therefore, it features the necessary expressiveness to model CPSs (R2a).
2. The semantics of SysML constructs can be adjusted or extended by using profiles (R2b).
3. SysML is the OMG[2] standard for systems engineering, and a popular and widely accepted modeling language in industrial settings (R2d).

Even though there is a plethora of open source and commercial tools that offer SysML support, no one is perfect for the job, the majority of them lacking the ability to execute models. Among all of them, we have selected MagicDraw. MagicDraw is a popular modeling tool which provides comprehensive support for SysML modeling (R3a). On the not so bright side, even though Magic Draw incorporates a model execution engine, its capabilities fall short to cover complex model execution scenarios. However, this can be overcome by using an approach based on code generation, with the help of its model API (R3b).

3.2 Modeling methodology overview

The selection of a modeling language and a modeling tool, though necessary, it is not enough to conduct the modeling activity in an adequate manner. In order to effectively model a system (or system-to-be), it is also necessary to define a modeling methodology. That methodology must be tailored to the domain of the system under analysis, it must be aligned with the objectives pursued and, above all, it must provide precise guidance on how to represent the system, its context, and the relationships between them, with the modeling language of choice.

We propose here a SysML-based methodology for the specification of architectural and behavioral software models of CPSs, plus the integration of function

[2]https://www.omg.org

models (e.g. Simulink models) describing hardware, software and environment aspects. These function models must be provided as an input to the modeling activity, and their specification lies outside the scope of this methodology. In relation to the requirements that models must fulfill in order to enable model testing of CPSs (see Subsection 2.1), the current version of this modeling methodology addresses the fulfillment of the following ones: (R1a) structural information, (R1b) behavioral information, and (R1c) environmental information. The fulfillment of the fourth requirement (R1d) uncertainty will be addressed in a future version of this document. When it comes to the requirements that the co-simulation framework must fulfill, this modeling methodology contributes to the fulfillment of (R4a) Co-Simulation, (R4d) Observability and (R4e) Controllability.

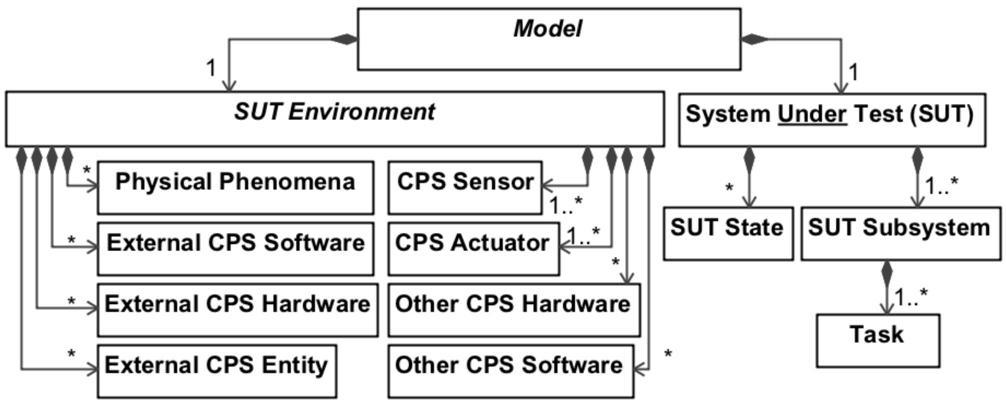

Figure 1: Information model

Figure 1 shows the concepts modeled with our proposed SysML-based methodology. The main concept is the SUT, i.e., the software system under test. The SUT may be composed by a series of software subsystems. Each of these subsystems is responsible for conducting a series of tasks (the basic units of behavior that can be scheduled). The SUT may also transition through a series of states. As part of its normal activity, the SUT may exchange data internally (among its 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.

Figure 2 shows the four main modeling steps, namely, the specification of the SUT environment, the specification of the SUT architecture, the specification of the SUT behavior and the specification of additional aspects to enable model testing. Even though an order for the completion of these steps has been set, it is neither mandatory to follow it, nor necessary to complete one step before starting the next. However, since each step depends on the previous one for its completion, and in order to maximize the effectiveness of the modeling effort, it is recommended to try to complete each step to a certain extent before moving onto the next.

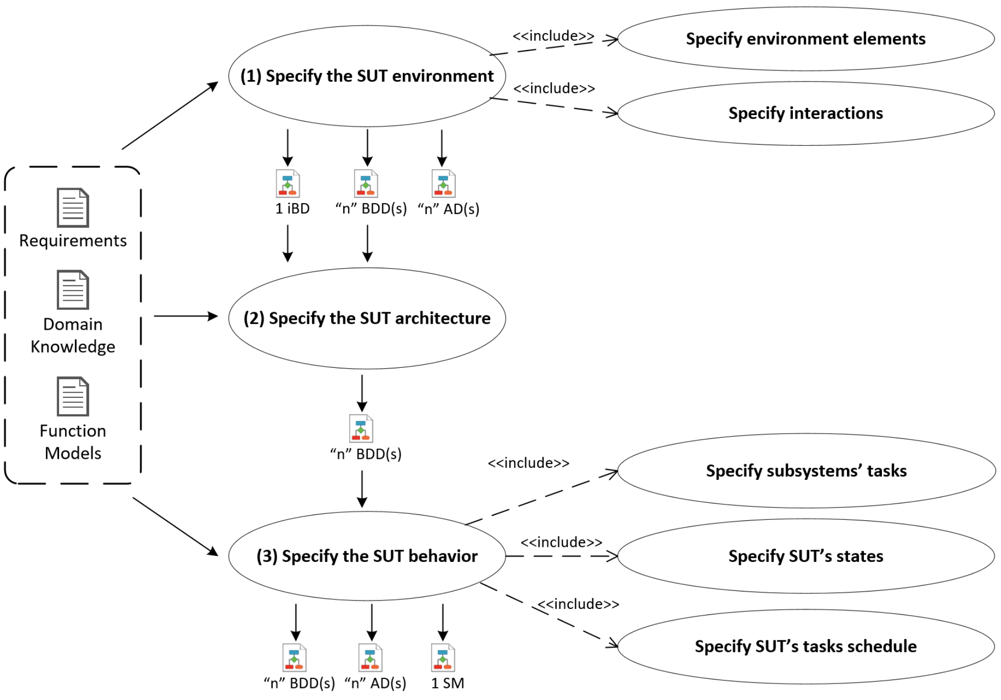

Figure 2: Methodology overview

In order to apply this modeling methodology, the presence of a SysML profile is required (Appendix B). This profile contains a series of stereotypes and tagged values, needed to customize some SysML modeling constructs to make them conform with the methodology's guidelines.

In the following subsections, the different modeling steps enforced by this methodology will be covered in detail. This also includes giving precise information on both, how to use the SysML profile, and when it is acceptable to move from one step to the next one.

3.3 Step 1: Specify the SUT environment

Purpose: The purpose of this step is to specify the boundaries between the system and the relevant external environment (hardware, software, people, etc.), the system interacts with.

Inputs: Requirements, domain documentation, function models.

Outputs: 1 internal block diagram (iBD), 1 or more block definition diagrams (BDDs), 1 or more activity diagrams (ADs).

Requirements fulfilled: (R1c) Environmental information. (R1a) Structural information (partially).

Guidelines: Conduct the following substeps to complete the specification of the

SUT environment:

1. Create one block[3] and name it “Model”. Henceforth, we will refer to this block as the “model block”.

2. Add the «Configuration» stereotype[4] to the model block.

3. Create one iBD for the model block.

4. Create one empty block representing the SUT, name it, and add it to the iBD. Henceforth, we will refer to this block as the “SUT block”.

5. Add the «SUT» stereotype to the SUT block.

6. Identify the relevant elements (sensors, actuators, people, external software systems, etc), the SUT must exchange information with. Create one empty block for each of them, name them, and add them to the iBD. Henceforth, we will refer to these blocks as “environment blocks”.

7. Complete the following actions for each environment block:

    - Identify one way the environment block and the SUT block exchange information with each other.
    - Create one port in the environment block, and one port in the SUT block.
    - Create a connector and use it to link both ports.
    - Create one BDD.
    - Create one empty block representing the information exchanged, name it, and add it to the BDD. Add the «Data» stereotype to the block. Henceforth, we will refer to these blocks as “data blocks”.
    - Go back to the iBD, and set the data type of the two ports to the data block added to the BDD.
    - Set the data type of the connector linking both ports, to the data block added to the BDD. Adjust the direction of the data flow (from the SUT block to the environment block, or vice versa).
    - Repeat the previous six actions until all the relevant interactions between the SUT block and the environment block have been modeled.

[3]The reader is referred to Appendix A for detailed guidelines on how to correctly specify the different SysML constructs supported by this modeling methodology.

[4]Appendix B provides detailed guidelines on how to use the different stereotypes that support this modeling methodology.

Thus far, we have specified empty blocks for the SUT, the relevant elements from the environment, and the way the SUT and these elements exchange data. However, it might also be necessary[5] to explicitly specify the way in which some of these elements from the environment (e.g. sensors and actuators) interact with the physical world. In that case, we proceed as follows:

8. Create one block to represent the physical world, name it, and add it to the iBD.

9. Repeat the procedure described in substep 7 to specify the interactions between the physical world and the relevant elements from the environment.

Now, in order to complete the specification of the context, it is necessary to specify 1) the structure of the data blocks corresponding to the identified data exchanges, 2) the structure and behavior of the environment blocks and 3) the structure and behavior of the physical world block. We start by describing how to specify the data blocks.

10. Complete the following actions for each of the empty data blocks created in substeps 7 and 9:

    - Open the BDD containing the data block.
    - Identify the pieces of information[6] that characterize the data that is going to be represented by the data block.
    - For each of those pieces of information that can be adequately modeled by using a simple data type[7], create a property for it, and add it to the data block.
    - For each of those pieces of information that cannot be adequately modeled by using a simple data type, create a new block for it and link this block to the data block by means of a composition.
    - Apply the previous 3 actions to any newly created block, to complete the specification of their properties.

Once the data blocks have been specified, we now continue by specifying the structure of the environment blocks.

11. Complete the following actions, for each of the environment blocks created in substep 6:

    - Create one BDD.

[5]This can be determined by checking whether the behavior of elements such as sensors and actuators is specified by means of function models and, if so, by analyzing their expected inputs and outputs.

[6]There is no one-size-fits-all rule for this. The modeler can freely group or decompose information considering a variety of criteria (e.g. logical relationship, convenience, etc.) as long as its inner nature is preserved in the process.

[7]See Appendix A for information about data types

- Add the environment block to the BDD.
- For each of the interactions specified in substep 7, between the environment block and the SUT block, create a property and add it to the environment block. Set the data type of this property to the data block characterizing that interaction.
- For each of the interactions specified in substep 9, between the environment block and the physical world block, create a property and add it to the environment block. Set the data type of this property to the data block characterizing that interaction.
- Follow the same procedure described in substep 10 to identify and model any additional internal data that might be necessary to completely specify the internal structure of the environment lock.

After specifying the structure for each environment block, it is time to specify their behavior.

12. Complete the following actions, for each of the environment blocks created in substep 6:

    - Open the BDD containing the corresponding environment block.
    - Identify one relevant functionality[8] of the element of the environment (sensor, actuator, etc) being analyzed.
    - Create an operation to represent that functionality, name it, and add it to the corresponding block.
    - Set neither input parameters nor a return value for the operation. Any data needed by the operation must be read from the properties of the block containing the operation, or from the properties of the other block (SUT or physical world) involved in the interaction. Any data produced by the operation must be stored in the properties of the block containing the operation.
    - Create an activity, name it with the operation's name, and add it to the block.
    - Link the operation to the activity.
    - Create one AD for the activity.
    - Describe[9] in the AD the sequence of actions, control and data flows that characterize the functionality being modeled. In case the functionality is characterized by an external function model, the AD must contain the sequence of actions, control and data flows, necessary to specify the integration of that function model.

[8]In this context, a relevant functionality is something that the element from the environment does when interacting with the SUT or the physical world, and that if it is not specified in the model, then the testing activity cannot be conducted properly.

[9]Detailed information on how to specify activities can be found in Appendix A.

- Repeat the previous 7 actions until all the relevant functionality of that element from the environment has been modeled.

Once the structure and behavior of environment blocks have been specified, the only remaining steps are the specification of the structure and behavior of the physical world block. When it comes to specifying its structure, proceed as follows:

13. Create one BDD.

14. Add the physical world block to the BDD.

15. For each of the interactions between environment blocks and the physical world block specified in substep 9, create a property and add it to the physical world block. Set the data type of this property to the data block characterizing that interaction.

16. Follow the same procedure described in substep 10 to identify and model any additional internal data that might be necessary to completely specify the internal structure of the block.

17. Finally, the behavior of the physical world block can be specified in the same manner described in substep 12 for the specification of the behavior of environment blocks.

At this point, the specification of the SUT environment has been completed.

Remarks:

- This modeling methodology does not provide any mechanism to explicitly differentiate hardware and software elements.

- Once the SUT block has been created and stereotyped, the modeler can also start working on step 2 (Section 3.4: Specify the SUT architecture).

- Even though following the instructions would result in creating one BDD per data block (environment block), it is up to the modeler to evaluate the real need for this. This methodology does not impose constraints on the number of BDDs needed to represent data blocks (environment blocks) and therefore, multiple data blocks (environment blocks) can also be grouped in the same BDD. The decision of whether to use one or more BDDs can be made attending to factors such as the number of relevant elements from the environment, the number of interactions between the SUT and these elements, as well as the complexity of the data exchanged.

- Interactions between blocks cannot be bidirectional. If the SUT interacts with one element from the environment by both sending and receiving data, the modeler should specify this as two one-direction interactions.

- When identifying the functionality of environment blocks, only the one related to the interactions the block is involved in should be considered. In general, when modeling the SUT environment, only those aspects that are relevant for testing the SUT should be taken into account.

Examples: In what follows, we are going to show how to apply these guidelines to specify the environment of a satellite's attitude control software system (ADCSSW). Figure 3(a) shows what the iBD looks like after completing the first 5 substeps. It can be seen that it contains an empty block called "ADCSSW", which has been stereotyped as «SUT». This is the SUT block.

Substep 6 is about identifying the relevant elements from the environment and creating empty blocks for each of them. In the example, the satellite's attitude control software system will interact with the satellite's on board computer, the satellite's sensors (sun sensors, magnetometers and gyroscopes), and the satellite's actuators (reaction wheels and magnetorquers). Figure 3(b) illustrates how several empty blocks representing these elements have been added to the iBD.

With the relevant elements from the environment in place, it is the time to specify how the SUT block (ADCSSW) interacts with them. The satellite's attitude control software system interacts with sensors and actuators by command them and, in the case of sensors, by receiving their readings. The interaction with the on board computer consists in sending status reports and receiving commands. Figure 3(c) shows the iBD resulting from completing substep 7. The system block has been linked to the blocks representing sensors, actuators and the on board computer by means of ports and connectors. The connectors represent data exchanges and therefore each of them is characterized by an empty data block (only the names of these blocks are shown to reduce clutter).

Substeps 8 and 9 address the specification of what we have called the physical world. In the example, as it will be seen later on, the behavior of sensors and actuators is specified by means of function models. These functional models work by receiving some of their inputs from some other function models, responsible for simulating some physical phenomena and conducting some complex mathematical calculations. Since the behavior of sensors and actuators could not be specified properly without taking these dependencies into account, it is necessary to reflect this in the iBD. We do this in substep 8 by creating a new block, which will gather the behavior corresponding to physical phenomena and mathematical calculations characterizing the physical world. After this, in substep 9, this block is linked to the blocks corresponding to sensors and actuators, again, by means of ports and connectors. The iBD resulting from completing substeps 8 and 9 can be seen in Figure 3(d). In this case, the block characterizing the physical world has been called "Physics".

Once substep 9 is completed, it is the time to specify the internal structure and behavior of the existing blocks. We start with the specification of the internal structure of data blocks in substep 10. Each of these data blocks characterize one interaction between two blocks. Therefore, the properties defined in each data block specify the data that will be exchanged between two blocks when the

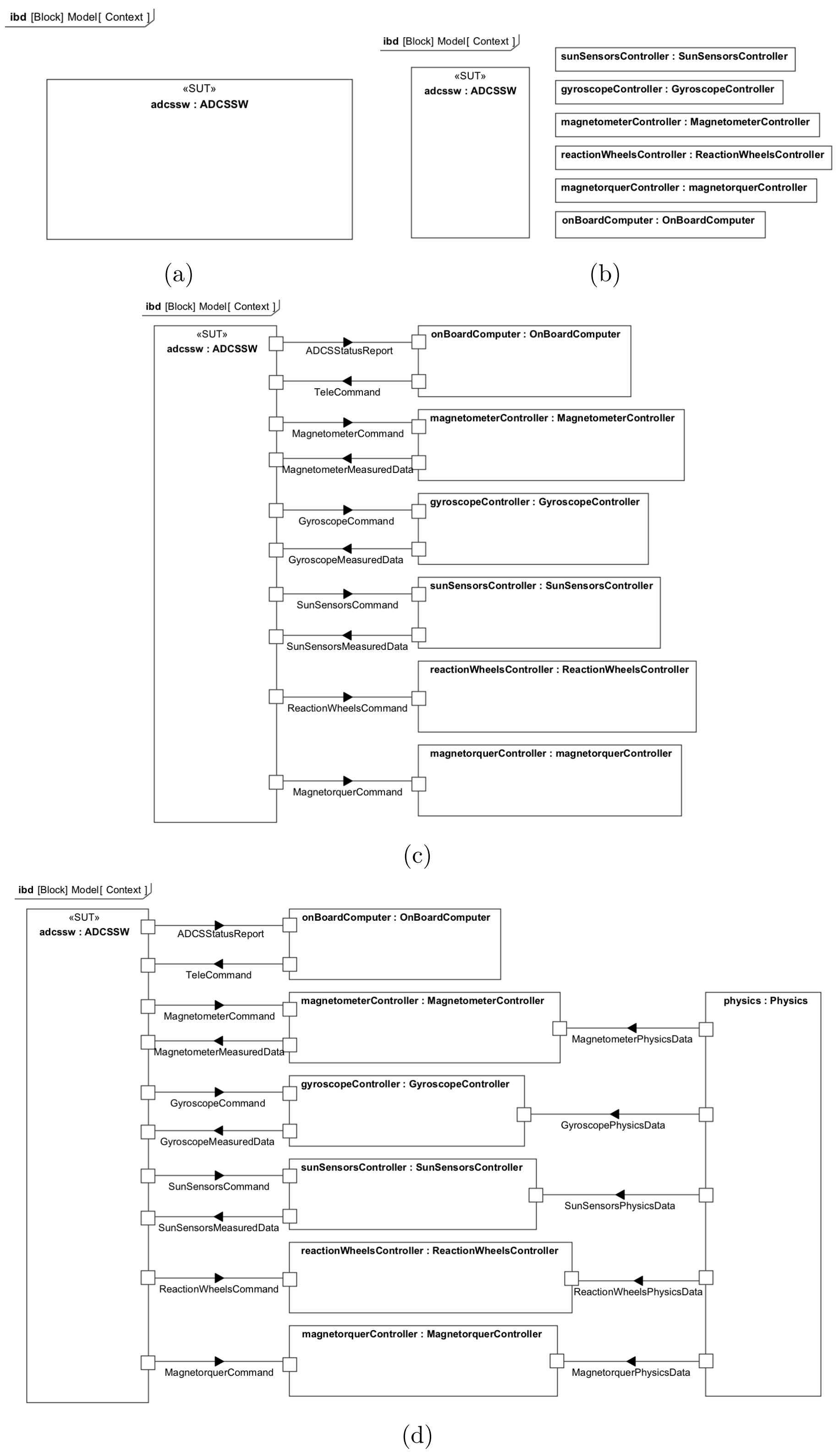

Figure 3: Evolution of the iBD of the example as substeps 1 to 9 are completed.

corresponding interaction takes place. Figure 4(a) shows the internal structure of some of the data blocks characterizing the interactions between the satellite's attitude control software system and the satellite's sensors. For example, the block called "MagnetometerMeasuredData" specifies the data that the satellite's attitude control software system will receive from the magnetometer when both interact. Similarly, Figure 4(b) show the internal structure of the data blocks that characterize the data exchanged between the physical world block and the satellite's sensors and actuators.

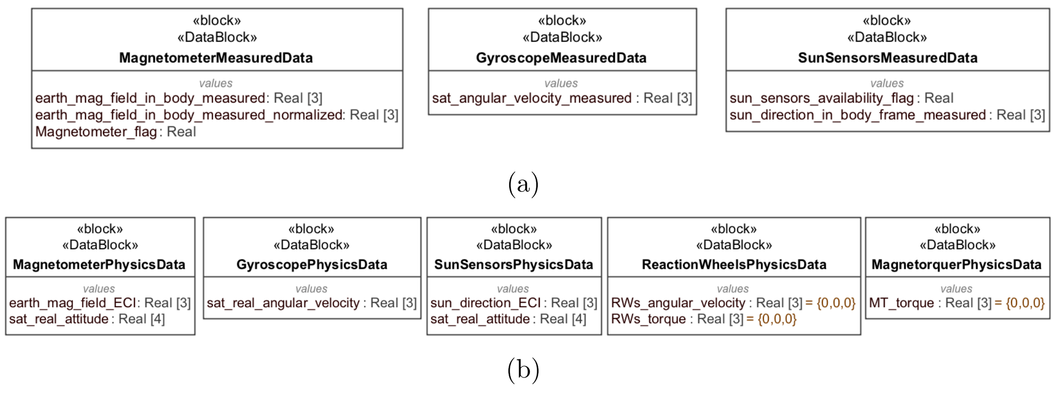
«block»
«DataBlock»
MagnetometerMeasuredData
values
earth_mag_field_in_body_measured: Real [3]
earth_mag_field_in_body_measured_normalized: Real [3]
Magnetometer_flag : Real
«block»
«DataBlock»
GyroscopeMeasuredData
values
sat_angular_velocity_measured : Real [3]
«block»
«DataBlock»
SunSensorsMeasuredData
values
sun_sensors_availability_flag : Real
sun_direction_in_body_frame_measured : Real [3]
(a)
«block»
«DataBlock»
MagnetometerPhysicsData
values
earth_mag_field_ECI: Real [3]
sat_real_attitude : Real [4]
«block»
«DataBlock»
GyroscopePhysicsData
values
sat_real_angular_velocity : Real [3]
«block»
«DataBlock»
SunSensorsPhysicsData
values
sun_direction_ECI : Real [3]
sat_real_attitude : Real [4]
«block»
«DataBlock»
ReactionWheelsPhysicsData
values
RWs_angular_velocity : Real [3] = {0,0,0}
RWs_torque : Real [3] = {0,0,0}
«block»
«DataBlock»
MagnetorquerPhysicsData
values
MT_torque : Real [3] = {0,0,0}
(b)

Figure 4: Internal structure of some of the data blocks specified in substep 10.

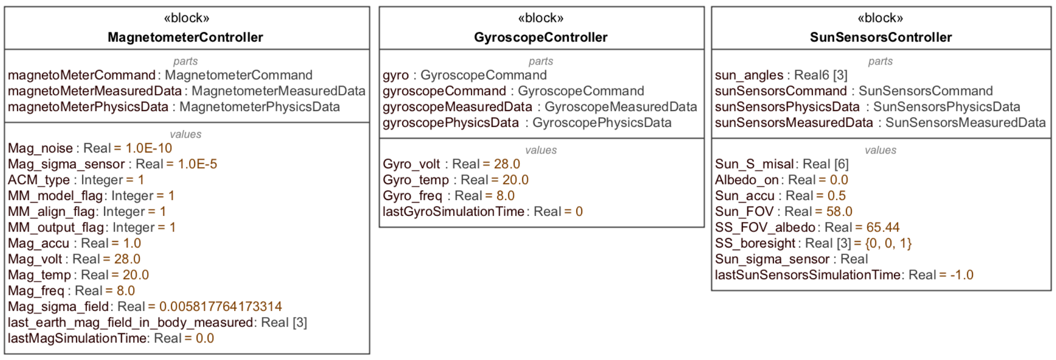
«block»
MagnetometerController
parts
magnetoMeterCommand : MagnetometerCommand
magnetoMeterMeasuredData : MagnetometerMeasuredData
magnetoMeterPhysicsData : MagnetometerPhysicsData
values
Mag_noise : Real = 1.0E-10
Mag_sigma_sensor : Real = 1.0E-5
ACM_type : Integer = 1
MM_model_flag : Integer = 1
MM_align_flag : Integer = 1
MM_output_flag : Integer = 1
Mag_accu : Real = 1.0
Mag_volt : Real = 28.0
Mag_temp : Real = 20.0
Mag_freq : Real = 8.0
Mag_sigma_field : Real = 0.005817764173314
last_earth_mag_field_in_body_measured : Real [3]
lastMagSimulationTime : Real = 0.0
«block»
GyroscopeController
parts
gyro : GyroscopeCommand
gyroscopeCommand : GyroscopeCommand
gyroscopeMeasuredData : GyroscopeMeasuredData
gyroscopePhysicsData : GyroscopePhysicsData
values
Gyro_volt : Real = 28.0
Gyro_temp : Real = 20.0
Gyro_freq : Real = 8.0
lastGyroSimulationTime : Real = 0
«block»
SunSensorsController
parts
sun_angles : Real6 [3]
sunSensorsCommand : SunSensorsCommand
sunSensorsPhysicsData : SunSensorsPhysicsData
sunSensorsMeasuredData : SunSensorsMeasuredData
values
Sun_S_misal : Real [6]
Albedo_on : Real = 0.0
Sun_accu : Real = 0.5
Sun_FOV : Real = 58.0
SS_FOV_albedo : Real = 65.44
SS_boresight : Real [3] = {0, 0, 1}
Sun_sigma_sensor : Real
lastSunSensorsSimulationTime : Real = -1.0

Figure 5: Internal structure of the blocks corresponding to the satellite's sensors.

After specifying the internal structure of data blocks in substep 10, substep 11 focuses on modeling the internal structure of those blocks representing the relevant elements from the environment. Figure 5 shows the internal structure of the blocks characterizing the satellite's sensors. It can be seen how the "parts" compartment, in each of these blocks, contains properties for the data blocks corresponding to the interactions in which these blocks are involved. It can also be seen how the "values" compartment includes a number of additional properties. These properties represent additional data needed by the functional models characterizing the sensors' behavior, such as sensor configuration data.

To complete the specification of the relevant elements from the environment, we also need to specify their behavior. We do this in substep 12. Figure 6 shows how

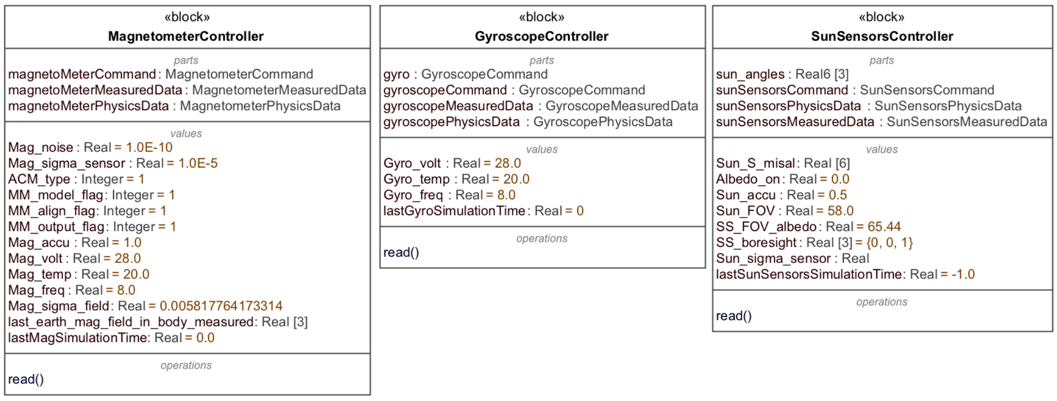
«block»
MagnetometerController
parts
magnetoMeterCommand : MagnetometerCommand
magnetoMeterMeasuredData : MagnetometerMeasuredData
magnetoMeterPhysicsData : MagnetometerPhysicsData
values
Mag_noise : Real = 1.0E-10
Mag_sigma_sensor : Real = 1.0E-5
ACM_type : Integer = 1
MM_model_flag: Integer = 1
MM_align_flag: Integer = 1
MM_output_flag: Integer = 1
Mag_accu : Real = 1.0
Mag_volt : Real = 28.0
Mag_temp : Real = 20.0
Mag_freq : Real = 8.0
Mag_sigma_field: Real = 0.005817764173314
last_earth_mag_field_in_body_measured: Real [3]
lastMagSimulationTime: Real = 0.0
operations
read()
«block»
GyroscopeController
parts
gyro : GyroscopeCommand
gyroscopeCommand : GyroscopeCommand
gyroscopeMeasuredData : GyroscopeMeasuredData
gyroscopePhysicsData : GyroscopePhysicsData
values
Gyro_volt : Real = 28.0
Gyro_temp : Real = 20.0
Gyro_freq : Real = 8.0
lastGyroSimulationTime : Real = 0
operations
read()
«block»
SunSensorsController
parts
sun_angles : Real6 [3]
sunSensorsCommand : SunSensorsCommand
sunSensorsPhysicsData : SunSensorsPhysicsData
sunSensorsMeasuredData : SunSensorsMeasuredData
values
Sun_S_misal: Real [6]
Albedo_on: Real = 0.0
Sun_accu : Real = 0.5
Sun_FOV : Real = 58.0
SS_FOV_albedo: Real = 65.44
SS_boresight: Real [3] = {0, 0, 1}
Sun_sigma_sensor : Real
lastSunSensorsSimulationTime: Real = -1.0
operations
read()

Figure 6: Operations added to the blocks corresponding to the satellite's sensors.

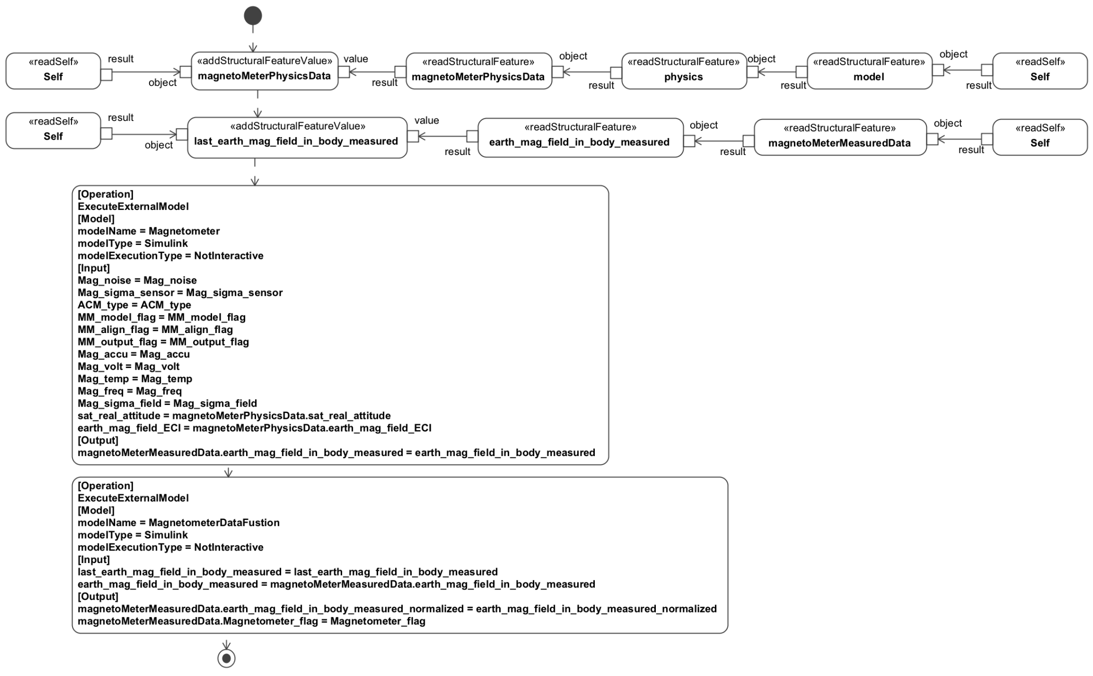
«readSelf»
Self
result
object
«addStructuralFeatureValue»
magnetoMeterPhysicsData
value
result
«readStructuralFeature»
magnetoMeterPhysicsData
object
result
«readStructuralFeature»
physics
object
result
«readStructuralFeature»
model
object
result
«readSelf»
Self
«readSelf»
Self
result
object
«addStructuralFeatureValue»
last_earth_mag_field_in_body_measured
value
result
«readStructuralFeature»
earth_mag_field_in_body_measured
object
result
«readStructuralFeature»
magnetoMeterMeasuredData
object
result
«readSelf»
Self
[Operation]
ExecuteExternalModel
[Model]
modelName = Magnetometer
modelType = Simulink
modelExecutionType = NotInteractive
[Input]
Mag_noise = Mag_noise
Mag_sigma_sensor = Mag_sigma_sensor
ACM_type = ACM_type
MM_model_flag = MM_model_flag
MM_align_flag = MM_align_flag
MM_output_flag = MM_output_flag
Mag_accu = Mag_accu
Mag_volt = Mag_volt
Mag_temp = Mag_temp
Mag_freq = Mag_freq
Mag_sigma_field = Mag_sigma_field
sat_real_attitude = magnetoMeterPhysicsData.sat_real_attitude
earth_mag_field_ECI = magnetoMeterPhysicsData.earth_mag_field_ECI
[Output]
magnetoMeterMeasuredData.earth_mag_field_in_body_measured = earth_mag_field_in_body_measured
[Operation]
ExecuteExternalModel
[Model]
modelName = MagnetometerDataFustion
modelType = Simulink
modelExecutionType = NotInteractive
[Input]
last_earth_mag_field_in_body_measured = last_earth_mag_field_in_body_measured
earth_mag_field_in_body_measured = magnetoMeterMeasuredData.earth_mag_field_in_body_measured
[Output]
magnetoMeterMeasuredData.earth_mag_field_in_body_measured_normalized = earth_mag_field_in_body_measured_normalized
magnetoMeterMeasuredData.Magnetometer_flag = Magnetometer_flag

Figure 7: Activity diagram describing the read() operation for the magnetometer.

the same blocks characterizing the satellite's sensors from Figure 5 now include another compartment called "operations", and that one operation called "read()" has been added. The idea here is that, when invoked, these operations will execute the behavior of the corresponding sensor. Figure 7 shows the activity diagram describing the behavior of the "read()" operation for the magnetometer. In this particular case, that behavior is decomposed is 4 action items:

1. With the first "addStructuralFeatureValue" action, the property called "magnetoMeterPhysicsData" is filled with the data stored in the property of the same name in the physics block. This is because, as it was mentioned earlier, sensors depend on certain data from the physical world. That data is produced and stored when the behavior of the "physics" block is executed,

which is independent of the sensor's behavior being described here.

2. One of the inputs for the function models describing the behavior of the magnetometer is the value of the last read made by the sensor. With the second "addStructuralFeatureValue" action, the property "last_earth_mag_field_in_body_measured" is filled with that value.

3. Once the first two actions are completed, the first of the two function models characterizing the magnetometer behavior is launched. The action describes in a textual form the model being launched as well as its inputs and outputs. For the case of the inputs, each line corresponds to a parameter. The left side is the name of the parameter in the function model, and the right side is the name of the property in the magnetometer block from which the corresponding value can be retrieved. For the case of the outputs, the left side corresponds to the properties in the magnetometer block where the output data must be stored, and the right side corresponds to the output parameter in the function model, where the actual data is located.

4. Finally, the fourth action launches the second function model. Once completed, the "magnetoMeterMeasuredData" property of the magnetometer block will contain the information read by the sensor.

The behavior for the rest of relevant elements from the environment, as well as the structure and behavior of the physical world block are not shown, since they are specified in the same manner.

3.4 Step 2: Specify the SUT architecture

Purpose: The purpose of this step is to describe the subsystems representing the internal structure of the SUT block introduced in the previous step.

Inputs: The iBD from step 1 (Section 3.3: Specify the SUT environment), the BDDs from step 1, requirements, domain documentation, function models.

Outputs: 1 or more BDDs.

Requirements fulfilled: (R1a) Structural information (partially).

Guidelines: Conduct the following substeps to complete the specification of the SUT architecture:

1. Create a BDD.

2. Add[10] the SUT block to the BDD.

[10]Modern modeling tools such as Magic Draw keep a repository with all the modeling elements added to the model. This repository is the place where to look for modeling elements that must

3. Identify the subsystems[11] of the SUT based on the analysis of the SUT's responsibilities. Create an empty block for each of them, name them, and add them to the BDD. Henceforth, we will refer to these blocks as "subsystem blocks".

4. For each subsystem block, create a composition link and connect the SUT block and the subsystem block with it.

Once all the subsystem blocks have been created and properly linked, the modeler must proceed as follows, to specify the data manipulated by both the SUT block and the subsystem blocks.

5. For each data block created in step 1, during the specification of the interactions between the SUT and its environment, identify the subsystem blocks that will manipulate that information, and create a property in each of them, its corresponding data type being that data block. There is one exception to this, though: if the modeler detects that two or more subsystem blocks are responsible for manipulating a given data block then, instead of creating a property in each of them, that property must be created in the SUT block.

6. If, after completing the previous substep, there still are data blocks for which no subsystem block is responsible, then the property corresponding to that data block must be created in the SUT block.

Once the previous two substeps are completed, the SUT block and the subsystem blocks will feature a series of properties, corresponding to the data blocks identified in step 1. However, chances are that some of those blocks will also be responsible for manipulating some other, additional "internal data". In what follows we describe how to proceed to specify such internal data.

7. For each subsystem block, identify the additional internal data, that block should be responsible for, and for each piece of data identified, find out whether it can be adequately modeled by using a primitive data type. If that is not the case, then that piece of data must be characterized by a new data block. In order to specify it, the modeler must follow here, the same instructions described in step 1, substep 10, to specify data blocks.

8. Once each piece of data has been identified and characterized with either a simple data type or a data block, the modeler must create a property for it in the subsystem block being analyzed. In the particular case that a given property is found to be needed in more than one subsystem block, then that property should be moved to the SUT block.

be reused across multiple diagrams (such as the SUT block). Avoid duplicating existing modeling elements. This modeling methodology clearly states when modeling elements must be created by using the word "Create". For the rest of cases, it is assumed that the modeling element being mentioned was created before and therefore, it must be reused. The only exception to this, already mentioned, has to do with the creation or re-utilization of diagrams.

[11]See the first entry in the "Remarks" subsection for more on this.

Remarks:

- This modeling methodology imposes some constraints in relation to how to decompose the SUT block:
  - Subsystem blocks cannot contain other subsystem blocks. Therefore the SUT block cannot be decomposed in multiple levels.
  - Subsystem blocks must feature some behavior[12] when step 3 (Section 3.5: Specify the SUT behavior) is completed.
  - The responsibility to manage state transitions, in case the SUT features a state-based behavior, should not be shared across more than one subsystem block.
- Once the modeler has identified at least one subsystem block, she can also start working on step 3 (Section 3.5: Specify the SUT behavior).
- The modeler is free to determine the number of BDDs resulting from the completion of this step. Depending on the number of blocks created, it might be a good idea to distribute the decomposition of the SUT block across several BDDs.
- Step 2 has been described as a series of substeps to be carried out in sequential order, but this is only to help the reader to understand what it takes to specify the SUT architecture. In general, experienced modelers do this in no particular order and through a series of refinements, as they gain more understanding of the system.

Examples: It is the time now to specify the subsystems of the satellite's attitude control software system (ADCSSW). Figure 8(a) shows the BDD resulting from the application of the first four substeps from the guidelines above. It can be seen that the system has been decomposed in four subsystems, namely "APDM", "ACM", "DEM" and "MM". The "APDM" subsystem will group the tasks for determining the satellite's attitude and position, which among others, include the acquisition of sensors' readings. The "ACM" subsystem will group the tasks for adjusting the satellite's attitude, which among others, include the commanding of the satellite's actuators. The "DEM" subsystem will group the tasks that allows the SUT to exchange information with the satellite's on board computer. Finally, the"MM" subsystem will be responsible for managing the different states the SUT can be in.

With the empty subsystem blocks in place, we can now proceed with substep 5, and identify which of these subsystem blocks will be responsible for handling the data blocks created, when specifying how the SUT interacts with its environment. Figure 8(b) shows how some properties have been added to the "APDM", "ACM"

[12]Behavior here refers to either operations or signal receptions, other than "getters" and "setters"

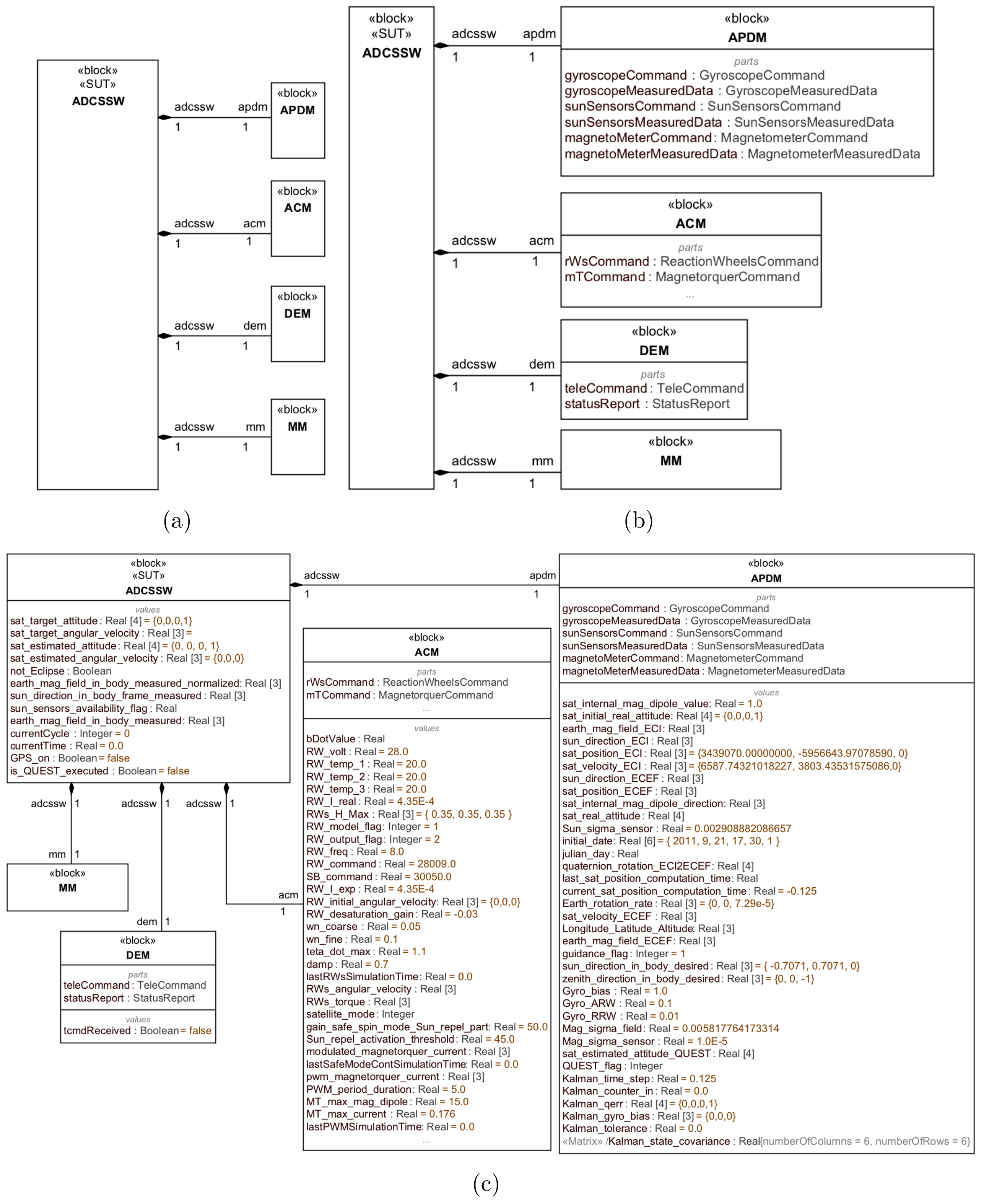
«block» «SUT» ADCSSW
adcssw 1 apdm 1 «block» APDM
adcssw 1 acm 1 «block» ACM
adcssw 1 dem 1 «block» DEM
adcssw 1 mm 1 «block» MM
(a)
«block» APDM
parts
gyroscopeCommand : GyroscopeCommand
gyroscopeMeasuredData : GyroscopeMeasuredData
sunSensorsCommand : SunSensorsCommand
sunSensorsMeasuredData : SunSensorsMeasuredData
magnetoMeterCommand : MagnetometerCommand
magnetoMeterMeasuredData : MagnetometerMeasuredData
«block» ACM
parts
rWsCommand : ReactionWheelsCommand
mTCommand : MagnetorquerCommand
«block» DEM
parts
teleCommand : TeleCommand
statusReport : StatusReport
«block» MM
(b)
«block» «SUT» ADCSSW
values
sat_target_attitude : Real [4] = {0,0,0,1}
sat_target_angular_velocity : Real [3] =
sat_estimated_attitude : Real [4] = {0, 0, 0, 1}
sat_estimated_angular_velocity : Real [3] = {0,0,0}
not_Eclipse : Boolean
earth_mag_field_in_body_measured_normalized : Real [3]
sun_direction_in_body_frame_measured : Real [3]
sun_sensors_availability_flag : Real
earth_mag_field_in_body_measured : Real [3]
currentCycle : Integer = 0
currentTime : Real = 0.0
GPS_on : Boolean = false
is_QUEST_executed : Boolean = false
«block» ACM
values
bDotValue : Real
RW_volt : Real = 28.0
RW_temp_1 : Real = 20.0
RW_temp_2 : Real = 20.0
RW_temp_3 : Real = 20.0
RW_I_real : Real = 4.35E-4
RWs_H_Max : Real [3] = { 0.35, 0.35, 0.35 }
RW_model_flag : Integer = 1
RW_output_flag : Integer = 2
RW_freq : Real = 8.0
RW_command : Real = 28009.0
SB_command : Real = 30050.0
RW_I_exp : Real = 4.35E-4
RW_initial_angular_velocity : Real [3] = {0,0,0}
RW_desaturation_gain : Real = -0.03
wn_coarse : Real = 0.05
wn_fine : Real = 0.1
teta_dot_max : Real = 1.1
damp : Real = 0.7
lastRWsSimulationTime : Real = 0.0
RWs_angular_velocity : Real [3]
RWs_torque : Real [3]
satellite_mode : Integer
gain_safe_spin_mode_Sun_repel_part : Real = 50.0
Sun_repel_activation_threshold : Real = 45.0
modulated_magnetorquer_current : Real [3]
lastSafeModeContSimulationTime : Real = 0.0
pwm_magnetorquer_current : Real [3]
PWM_period_duration : Real = 5.0
MT_max_mag_dipole : Real = 15.0
MT_max_current : Real = 0.176
lastPWMSimulationTime : Real = 0.0
«block» APDM
values
sat_internal_mag_dipole_value : Real = 1.0
sat_initial_real_attitude : Real [4] = {0,0,0,1}
earth_mag_field_ECI : Real [3]
sun_direction_ECI : Real [3]
sat_position_ECI : Real [3] = {3439070.00000000, -5956643.97078590, 0}
sat_velocity_ECI : Real [3] = {6587.74321018227, 3803.43531575086,0}
sun_direction_ECEF : Real [3]
sat_position_ECEF : Real [3]
sat_internal_mag_dipole_direction : Real [3]
sat_real_attitude : Real [4]
Sun_sigma_sensor : Real = 0.002908882086657
initial_date : Real [6] = { 2011, 9, 21, 17, 30, 1 }
julian_day : Real
quaternion_rotation_ECI2ECEF : Real [4]
last_sat_position_computation_time : Real
current_sat_position_computation_time : Real = -0.125
Earth_rotation_rate : Real [3] = {0, 0, 7.29e-5}
sat_velocity_ECEF : Real [3]
Longitude_Latitude_Altitude : Real [3]
earth_mag_field_ECEF : Real [3]
guidance_flag : Integer = 1
sun_direction_in_body_desired : Real [3] = { -0.7071, 0.7071, 0}
zenith_direction_in_body_desired : Real [3] = {0, 0, -1}
Gyro_bias : Real = 1.0
Gyro_ARW : Real = 0.1
Gyro_RRW : Real = 0.01
Mag_sigma_field : Real = 0.005817764173314
Mag_sigma_sensor : Real = 1.0E-5
sat_estimated_attitude_QUEST : Real [4]
QUEST_flag : Integer
Kalman_time_step : Real = 0.125
Kalman_counter_in : Real = 0.0
Kalman_qerr : Real [4] = {0,0,0,1}
Kalman_gyro_bias : Real [3] = {0,0,0}
Kalman_tolerance : Real = 0.0
«Matrix» /Kalman_state_covariance : Real{numberOfColumns = 6, numberOfRows = 6}
«block» DEM
values
tcmdReceived : Boolean = false
(c)

Figure 8: Evolution of the BDD of the example as substeps 1 to 8 are completed.

and "DEM" blocks to reflect this. In particular, the "APDM" block features the properties corresponding to the data blocks specifying how the SUT interacts with sensors; the "ACM" block features the properties corresponding to the data blocks specifying how the SUT interacts with actuators; and finally, the "DEM" block features the properties corresponding to the data blocks specifying how the SUT interacts with the on board computer. Since properties for all the data blocks characterizing the interactions between the SUT and the environment have been defined, there is no need to apply substep 6, and define additional properties in the system block (ADCSSW).

Finally, to complete the specification of the SUT architecture, in substeps 7 and 8 we must identify any additional internal data that the subsystem blocks are going to be responsible for. Figure 8(c) shows the additional properties added to the different subsystem blocks. It can be seen that some properties have also been added to the SUT block. As stated in substep 8, instead of duplicating these properties in each of the subsystem blocks where they are necessary, they were directly specified in the SUT block.

3.5 Step 3: Specify the SUT behavior

Purpose: The purpose of this step is to describe the tasks, each of the subsystem blocks introduced in step 2 (Section 3.4: Specify the SUT architecture) is responsible for. The combination of all these tasks is what will characterize the SUT's behavior.

Inputs: The BDDs from step 2 (Section 3.4: Specify the SUT architecture), requirements, domain documentation, function models.

Outputs: 1 or more BDDs, 1 or more ADs, 1 state machine (SM).

Requirements fulfilled: (R1b) Behavioral information.

Guidelines: We begin the specification of the SUT's behavior, by describing the tasks not directly related to managing SUT's states. In order to do this, proceed as follows:

1. Complete the following actions, for each subsystem of the SUT:
    - Open the BDD containing the corresponding subsystem block.
    - Identify one relevant task[13] conducted by the subsystem being analyzed.
    - Create an operation to represent that task, name it, and add it to the corresponding block.
    - Add the «Schedulable» stereotype[14] to the operation.
    - Set neither input parameters nor a return value for the operation. Any data needed by the operation must be read either from the properties of the block containing the operation, or from the properties of any other subsystem block, or from the properties of the SUT block, or from the properties of any of the environment blocks. This last case only applies when the operation in question is part of the specification

[13]In this context, a relevant task is something that the subsystem does, and whose execution must be scheduled..

[14]Appendix B provides detailed guidelines on how to use the different stereotypes that support this modeling methodology.

of an interaction between the SUT and its environment. When it comes to the data produced by the operation, if any, it must be stored in the properties of the block where the operation is specified.

- Create an activity, name it with the operation's name, and add it to the subsystem block.
- Link the operation to the activity.
- Create one AD for the activity.
- Describe[15] in the AD the sequence of actions, control and data flows that characterize the behavior of the task. In case the behavior of the task is characterized by an external function model, the AD must contain the sequence of actions, control and data flows, necessary to specify the integration of that function model.
- Repeat the previous 8 actions until all the tasks that subsystem is responsible for have been specified.

Sometimes, the modeler may discover that some behavior is repeated across several tasks. In these occasions, the modeler can create a separated operation[16], inside the subsystem block to encapsulate that piece of behavior, but without adding the «Schedulable» stereotype. That operation can then be used to complete the specification of those tasks, like any other modeling construct. This strategy can also be applied to simplify the specification of a complex task without cluttering the corresponding AD.

At this point, the tasks not related to managing SUT's states have been modeled. If the SUT features a state-based behavior, then we proceed as follows:

2. Create a BDD.

3. Identify the internal states of the SUT. Create an empty block for each of them, name them, and add them to the BDD. Henceforth, we will refer to these blocks as "state blocks".

4. Create one abstract block, name it, and add it to the BDD. This block will be used as some sort of generic state to group all the state blocks under. Henceforth, we will refer to this block as the "generic state block"

5. For each state block, create a generalization link and use it to connect that state block to the generic state block.

6. For each state block, create an association link and use it to connect that state block to the system block.

7. Identify the subsystem block responsible for managing the SUT's states. Create a composition link and connect this subsystem block to the generic state block with it.

[15]Detailed information on how to specify activities can be found in Appendix A.
[16]Along with its corresponding activity.

After specifying the blocks for the SUT's states, and before specifying the state transitions, it is necessary to specify the events the SUT must react to.

8. Identify the subsystem block responsible for managing the SUT's states.

9. Do as follows, for each of the events the SUT must react to.

    - Add a property to that subsystem block. Name it after the event. Set its type to "Boolean".
    - Create an operation with neither parameters nor returned value in the generic state block, and name it after the property created previously.

Once the events have been specified, it is the moment to specify how the SUT transitions from one state to another.

10. Create an operation with neither input parameters nor returned value, name it, and add it to the subsystem block that manages the SUT's states. The behavior of this operation will specify how the SUT transitions from one state to another.

11. Add the «Schedulable» stereotype[17] to the operation.

12. Specify how the SUT transitions among the different states by completing the following actions:

    - Create an SM and add it to the subsystem block responsible for managing the SUT's states.
    - Add to the SM an "Initial node".
    - Add to the SM one state for each of the state blocks and name them the same.
    - Analyze the way the SUT must transition among the different states, and link the states in the SM accordingly by means of transitions.
    - Link each transition with the property created in substep 9, that corresponds to the event that triggers that transition.

13. Link the SM to the operation created in substep 10[18].

At this point, events, states and transitions have been specified. In order to complete the specification of the state-based behavior, two additional operations must be specified.

[17] Appendix B provides detailed guidelines on how to use the different stereotypes that support this modeling methodology.

[18] The specifics on how a state machine can be linked to an activity can be found in Appendix A.

14. Create an operation with neither input parameters nor returned value, name it, and add it to the subsystem block responsible for managing the SUT's states.

15. Add the «Schedulable» stereotype to the operation.

16. Create an activity, name it with the operation's name, and add it to the block.

17. Link the operation to the activity.

18. Create one AD for the activity.

19. Describe in the AD the sequence of actions, control and data flows that characterize the reception of any of the events modeled (e.g., when one of the properties in a subsystem block is above/below a given threshold). As a result, this AD must activate/deactivate the properties created in substep 9. Once any of the properties is activated, it is assumed that the corresponding event has been triggered.

20. Create an operation with neither input parameters nor returned value, name it, and add it to the subsystem block that manages the SUT's states. This operation will be responsible for executing the behavior that the SUT must feature when entering into a given state.

21. Add the «Schedulable» stereotype to the operation.

22. Create an activity, name it with the operation's name, and add it to the block.

23. Link the operation to the activity.

24. Create one AD for the activity.

25. Describe in the AD the sequence of actions, control and data flows that characterize the behavior of the SUT when entering in any of the modeled states, including the deactivation of the property that was activated, when the event was triggered.

At this point, there is only one pending task to complete the specification of the system's behavior, that is, describing how the system's tasks are going to be scheduled. In order to do this, proceed as follows:

26. Set the tagged definition "timeStepSize" in the SUT block to the number of milliseconds that each time slot of the scheduler is going to last.

27. Set the tagged definition "executionRateHz" for each operation stereotyped as «Schedulable».

28. Set the tagged definition "estimatedCompletionTime" for each operation stereotyped as «Schedulable».
29. Set the tagged definition "executionOrder" for each operation stereotyped as «Schedulable». The order of execution of tasks (1, 2, 3...) must be set globally (at the SUT level, not at the subsystem level).

Remarks:

- The SUT block should not have tasks associated to it. Tasks should be distributed across the subsystem blocks.
- Only one SM should be specified to describe state transitions. This SM must be invoked from only one operation in the subsystem block responsible for managing the SUT's states.
- Step 3 has been described as a series of substeps to be carried out in sequential order, but this is only to help the reader to understand what it takes to specify the SUT's behavior. In general, experienced modelers do this in no particular order and through a series of refinements, as they gain more understanding of the system.

Examples: We begin the specification of the SUT's behavior, by creating operations in the subsystem blocks, for those tasks not directly related to managing the states of the ADCSSW system. Figure 9 shows the operations defined for the subsystem blocks "APDM", "ACM" and "DEM". In the case of the "APDM" block, these are the operations that must be conducted to determine satellite's position and attitude. When it comes to "ACM", the operations displayed in the figure are the ones conducted to control satellite's attitude. Operations defined in "DEM" allow the system to interact with the on board computer. Finally, no operations have been added to the "MM" since is the subsystem responsible for managing the states of the SUT.

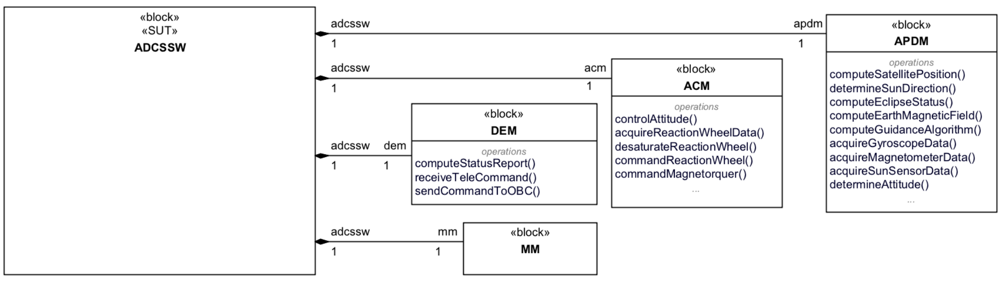

Figure 9: Definition of tasks not related to managing SUT's states.

The behavior of each of these operations has been defined by means of ADs. Figure 10 displays the AD corresponding to the activity "controlAttitude", linked to the operation with the same name in the block "ACM". In this particular case,

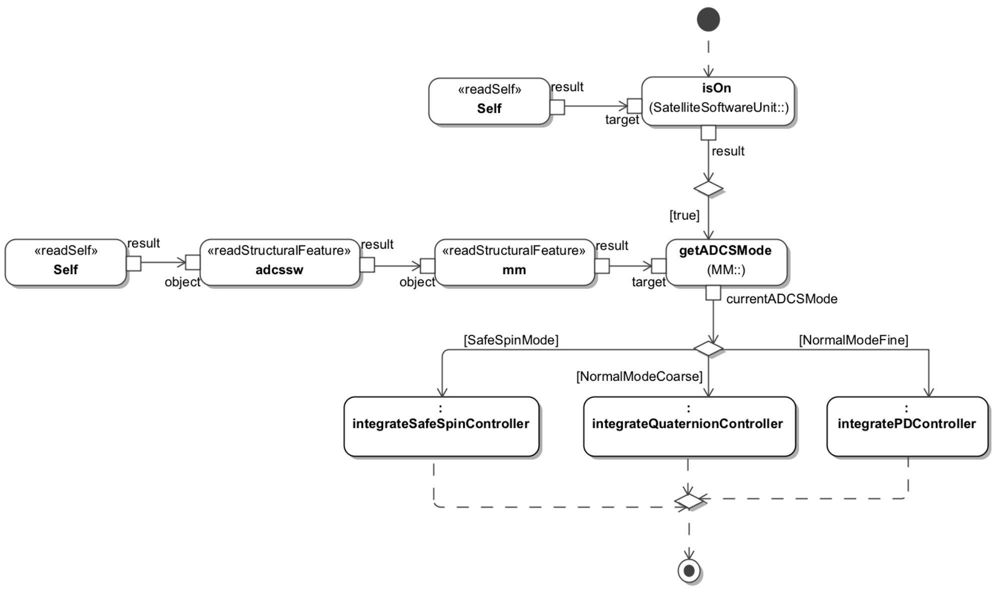

Figure 10: Behavior of the task "controlAttitude" in the "ACM" subsystem.

some of the behavior for this activity has been encapsulated in separated operations (not shown in figure 9), as indicated in the guidelines.

Substeps 2 to 7 are about specifying the states of the SUT. Figure 11(a) shows the result of applying these substeps to the SUT of the example. Blocks named "IdleMode", "SafeSpinMode", "NormalModeCoarse" and "NormalMode-Fine" have been created to represent these states (or modes). They are specializations of the generic state block called "ADCSMode', which is linked to the "MM" block by means of a composition link. Finally, the state blocks are linked to the SUT block by means of associations.

Once the blocks for the different states have been created, in substeps 8 and 9, we identify the events that can cause the SUT to transition from one state to another. In our example, the SUT responds to the following events: "Separation from Launcher", "Pointing Error Under 15 degrees", "Pointing Error above 20 degrees", and the reception of telecommands forcing the SUT to transition to any of the supported states. Figure 11(b) shows how a Boolean property for each of these events has been added to the "MM" subsystem. Also, operations named after these properties have been added to the generic state block.

After specifying the events, we can now model the transitions between the different states (substeps 10 to 13). Figure 11(c) shows how an operation called "executeTransitions()" has been added to the "MM" block. This operation will be responsible for transitioning the SUT from one state to another, anytime one of the previous events is triggered. The behavior of the "executeTransitions()" operation is specified by means of the SM in figure 12. It can be seen how the boolean properties previously added to the "MM" block have been linked to the different transitions defined.

To complete the specification of the state-based behavior of the SUT (sub-

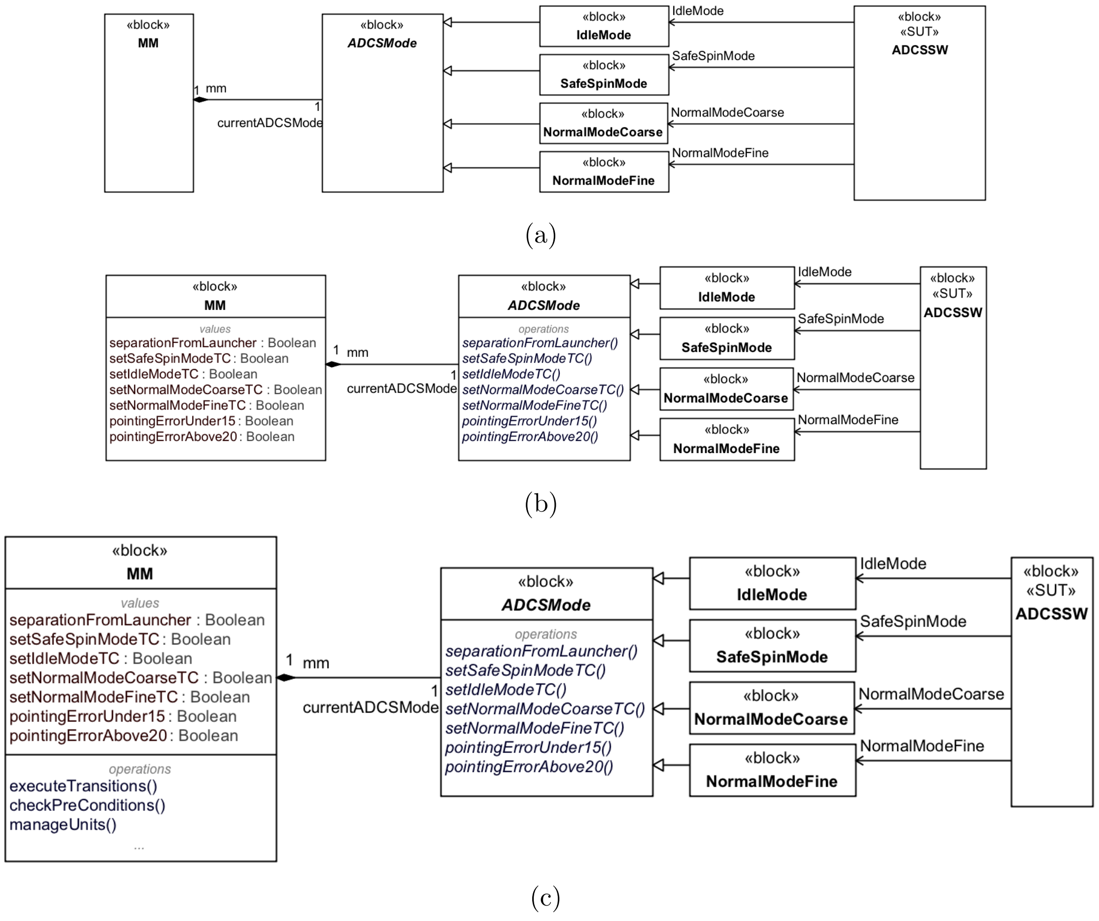

Figure 11: Specification of the states of the SUT.

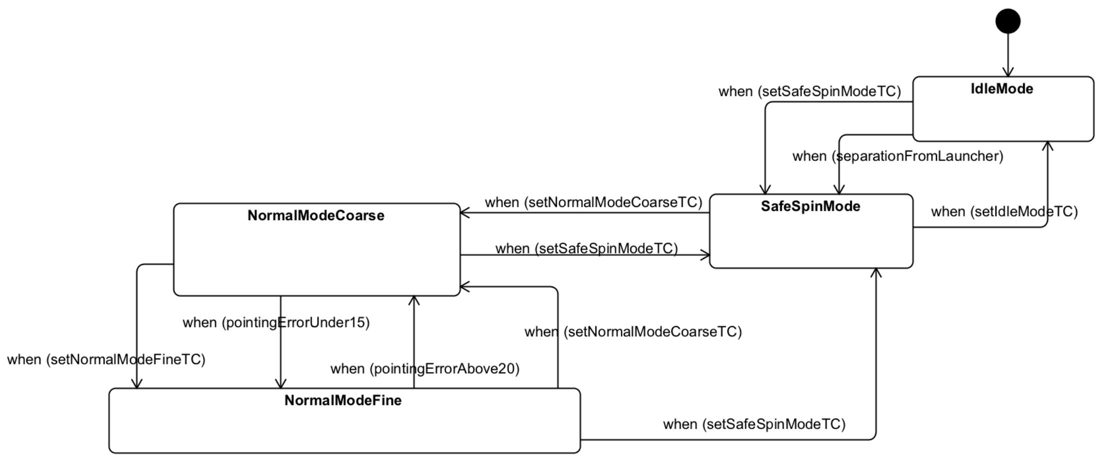
IdleMode
when (setSafeSpinModeTC)
when (separationFromLauncher)
SafeSpinMode
when (setIdleModeTC)
when (setNormalModeCoarseTC)
NormalModeCoarse
when (setSafeSpinModeTC)
when (pointingErrorUnder15)
when (setNormalModeFineTC)
when (pointingErrorAbove20)
when (setNormalModeCoarseTC)
NormalModeFine
when (setSafeSpinModeTC)

Figure 12: State transitions of the SUT of the example.

steps 14 to 25), we must specify two additional operations. One to check whether any of the events have been triggered, and another one to determine the behavior of the SUT when entering into a specific state, once a transition has been executed. Figure 11(c) shows how two operations called “checkPreconditions()” and “manageUnits()” have been added to the “MM” block. The behavior of these two operations is specified by means of two ADs (not shown here). In particular, the AD for the “checkPreconditions()” operation checks out whether the property “tcmdReceived” of the DEM subsystem is enabled. If so, it means that one telecommand was received and stored in the “teleCommand” property of the “DEM” block. That property is then inspected, to enable the corresponding flag in the “MM” block. The rest of preconditions are checked out in a similar way. When it comes to “manageUnits()’, the corresponding AD checks out what is the current state of the SUT, and invokes some operations in the different subsystem blocks. It also deactivates the flag from the “MM” block that was activated to indicate the triggering of the event.

A Guidelines on SysML modeling

A.1 Naming convention

Except for a few cases, most of the modeling elements added to the model must be given a name. There are different ways to name a model element, but a simple one that is common to all of them is to access to its property page, and fill in the "name" property. However, at the time of choosing the right name, there are a series of constraints that must be satisfied:

- Valid names must be composed by a combination of alphabetic characters and numbers, the first one being a letter.
- Do not name two blocks with the same name.
- Do not name two properties, two operations, or two signal receptions in the same block with the same name. In general, avoid name duplication within the boundaries of a block.
- The naming system is case-insensitive.
- Name's length is limited to 1024 characters.

A.2 Data types

Some modeling elements, in order to be properly specified, must have a data type. The data types that can be used when specifying block properties, operations and signal receptions are:

- Real
- String
- Boolean
- Integer
- Any of the previous four stereotyped as «Matrix»
- A data block[19]

The first five data types in the list are considered simple data types.

[19] Avoid recursion when assigning data blocks as data types.

A.3 Block specification

Specifying a block implies giving it a name, specifying whether it is an abstract block, specifying its properties (if any), specifying its operations (if any), specifying its ports (if any), specifying its stereotypes and tagged definitions (if any), and specifying how the block must be linked to other blocks. In the rest of this section, we will provide guidelines for the specification of all these modeling elements, except for the specification of links between blocks, that will be covered later on.

A.3.1 Specification of block properties

When it comes to the specification of block properties, the following rules apply:

- Properties must be given a name.
- Properties must have a data type.
- Properties must have a multiplicity of one (1).

A.3.2 Specification of block ports

Block ports are used for the specification of the interactions between the SUT and its environment. At the time of specifying block ports, the following rules apply:

- Ports must be given a name.
- Do not use “Proxy” ports, “Full” ports or “Flow” ports. Only regular ports are supported.
- Multiplicity of ports must be one (1).
- Ports must have a data type.
- Block ports must be connected in pairs. No block ports must remain disconnected once the model is finished.
- Two block ports must be connected by means of a “Connector” link[20].
- Every link connecting two ports must have associated a data block through its “Item Flow” property. Flow direction must be specified as well.

[20]Do not confuse with association links, composition links or generalization links. This is an specific connector available when modeling iBDs.

A.3.3 Specification of block operations

When specifying block operations, the following rules apply:

- Operations must be given a name.
- Operations cannot be abstract unless they are affected by a generalization link (see the specification of generalization links to find out more about this).
- Operations have at most one "return" parameter.
- Operations cannot have "in", "inout", or "out" parameters.
- Multiplicity of parameters must be one (1).
- Operations can have associated at most one behavioral element (one activity or one state machine). This is done by setting the field "Method" to the name of the corresponding behavioral element.
- There cannot be more than one operation in the whole model whose behavioral element is a state machine.

A.4 Specification of links between blocks

There are a number of different links that can be used to connect two blocks, namely, association links, generalization links, and composition links. In the rest of this section, we will describe how to use each of them.

A.4.1 Specification of composition links

Composition links are used for:

- Connecting the SUT block to its subsystem blocks.
- Connecting a data block to another, second, data block describing a data type, that is necessary to complete the specification of the first data block.
- Connecting the subsystem block responsible for handling the states of the SUT to the generic state block .

In order to specify a composition link, the following rules apply:

- Both ends of the composition link must be given a name.
- Both ends of the composition link must be navigable.
- When connecting the SUT block to its subsystem blocks, the end with the solid diamond must be connected to the SUT block.

- When connecting a subsystem block to the generic state block, the end with the solid diamond must be connected to the subsystem block.
- The cardinality of the end with the solid diamond must be one (1).
- The cardinality of the end without the solid diamond must be one (1).

A.4.2 Specification of generalization links

Generalization links are used for connecting the generic state block to the state blocks. When doing so, the arrowhead must point to the generic state block.

The utilization of generalization links has an impact on how blocks are specified. In particular, on whether a block can be abstract or not, and on how properties and block operations must be specified:

- Blocks cannot be abstract unless they are affected by a generalization link.
- Only blocks pointed by the arrowhead of a generalization link can be abstract.
- Multiple inheritance is not allowed.
- Chaining generalization links is not allowed.
- One block cannot be affected by more than one generalization link (followed from the previous two).
- Properties specified in a block pointed by a generalization link cannot be duplicated in the block at the other end of the link.
- Operations specified in a block pointed by a generalization link can be redefined in the block at the other end of the link by creating an operation with the same name.

A.4.3 Specification of association links

Association links are used for establishing other types of dependencies between blocks. There are two main reasons to do this:

- Connecting the state blocks to the SUT block.
- Connecting data blocks to the SUT block, the subsystem blocks, the state blocks or the generic state block.

In order to specify a association link, the following rules apply:

- Both ends of the association link must be given a name.
- Only one end of the association link must be navigable.

- The cardinality of both association ends must remain unspecified.
- When connecting the state blocks to the SUT block, the association end connected to the state block must be navigable.
- When connecting a data block to any other block, the association end connected to the data block must be navigable.

A.5 Activity specification

Specifying an activity implies giving it a name, specifying its inputs and outputs, specifying its actions, and specifying its control and data flows.

A.5.1 Specification of inputs and outputs

An activity can have inputs and/or outputs in two cases:

- When the activity is linked to an operation which features a return value. In this case, the activity will feature one output parameter, with the same type as the value returned by the operation.
- When the activity is not linked to an operation. In this case the activity will feature as many input and output parameters as needed to specify its behavior.

A.5.2 Specification of actions

Actions are the fundamental units to describe behavior in an AD. At the time of specifying an action, it is necessary to select the type of the action, give the action a name[21], and specify their inputs and outputs. In order to accomplish this last step, actions feature pins to receive or return data, the number of pins and their kind depending on the semantics (type) of the action being specified. Therefore, to specify the inputs and outputs for an action, it is necessary to fully specify the corresponding pins. A pin is specified by giving it a name, and by setting its data type accordingly, that is, the modeler must indicate for each pin, the nature of the data the action expects to receive (or is going to return) throughout that pin.

Among all the actions described in the SysML specification, this modeling methodology supports the following ones:

- Read Self Action: To access the object containing the activity being executed (also known as the context for the activity). It features no input pins and one output pin, to return this object.

[21]Actions and action pins can receive duplicated names.

- Read Structural Feature Action: To get the value of a given property. It features one input pin to receive the instance of the object which contains the property, and one output pin to return the value of the property from the object passed as context. In order to complete the specification of this action, it is necessary to fill in the field “Structural Feature” with the name of the property that has to be accessed.

- Add Structural Feature Action: To change the value of a given property. It features two input pins, one to receive the instance of the object which contains the property to be modified, and one to receive the value that must be assigned to that property. As in the previous case, in order to complete the specification of this action, it is necessary to fill in the field “Structural Feature” with the name of the property that has to be accessed. Additionally, the field “Is Replace All” must be set to “true”.

- Call Operation Action: To execute the behavior of an operation. It features one input pin to receive the instance of the object in which the operation is defined. If the operation returns a value, then it also features an output pin of the corresponding type. The specification of the action is completed when the field “Operation” is assigned with the name of the operation being called.

- Call Behavior Action: To execute an activity or state machine not directly linked to an operation in a block. The process to specify this action is the same described for the case of a “Call Operation Action”.

- Value Specification Action: To specify constant values. It features one output pin to return the constant. To specify the constant, the modeler must fill in the field “Value” with a literal expression of any of the supported simple data types.

- Opaque Action: To execute a function model. It contains a block of text in the “Body” property specifying what is necessary to execute the function model. The text is structured in four sections:

  - Operation: It contains the type of operation being executed. Only “ExecuteExternalModel” is supported for now.
  - Model: It contains data about the model to be executed. Three lines expressed as assignment (=) statements. The left side of the first line is the fixed word “modelName”. The right side of this line is the name of the external model to be executed. The left side of the second line is the fixed word “modelType”. The right side of this line is the fixed word “Simulink”, since this is the type of function model supported for now. The left side of the third line is the fixed word “modelExecutionType”. The right side of this line is set as follows:

    * “NotInteractive” for Simulink models in which, the computation of the outputs only depends on the inputs passed onto the model.
    * “InteractiveAndSynchronized” for Simulink models in which, the computation of the outputs at a certain point during the simulation, depends on the previously computed outputs, and on the time step size, which, in its turn, represents a certain amount of time elapsed in the physical world.
    * “InteractiveAndNotSynchronized” for Simulink models not falling into the previous two groups.
  – Input: Input parameters for the model. One per line expressed as assignment statements. The left side represents the name of the parameter in the function model. The right side represents the name of the property in the SysML model from which the value is retrieved.
  – Output: Data returned. One per line expressed as assignment statements. The left side represents the name of the property in the SysML model where the value will be stored. The right side represents the name of the output parameter in the function model.

Each section starts with the name of the section between brackets in a separated line.

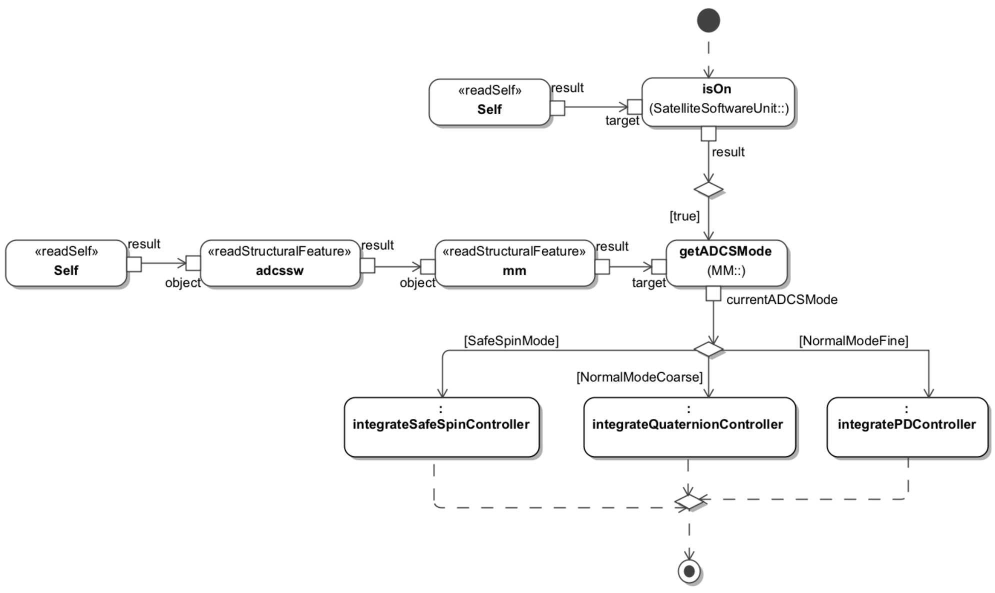

Figure 13: Behavior of the task “controlAttitude” in the “ACM” subsystem.

A.5.3 Specification of control and data flows

Apart from the actions described above, the following elements are supported at the time of specifying the data and control flow of an activity.

- Initial node
- Final node
- Decision node
- Merge node
- Fork node
- Join node
- Input parameter node
- Output parameter node
- Data flow link
- Control flow link

Decision nodes and merge nodes, as well as fork nodes and join nodes must be used in pairs. It is not necessary for an activity to feature an initial node or a final node, as long as it features input and output parameters. In general, either an initial node or an input parameter must exist, and the same applies to final nodes and output parameters.

Before specifying how to use control flow links and data flow links, it is necessary to make a remark on how actions can be used. As indicated above, actions serve the purpose of describing the behavior of activities. However, at the time of doing this, actions can be used in two different ways: to model the actual behavior of the activity (calling an operation, reading a value, modifying a value, etc.), or to conduct a series of auxiliary steps needed to provide the right context for the actions in the first group. As an example of this, figure 13 shows that the behavior of the "controlAttitude' task is to execute the operation "isOn" and then, if the returned value is "true", to execute the operation "getADCSMode". Then, depending on the returned value, one of the activities "integrateSafeSpinController", "integrateQuaternionController", or "integratePDController" is executed. However, in order to execute the operations "isOn" and "getADCSMode", it is necessary to provide the right context for them. This is done with the actions that can be seen at the left of these two operations. Whether the actions are used to provide context for other actions or not, has an impact on how data flow links and control flow links are used:

- When several actions are chained to define the context for another action, they all must be linked throughout their input and output pins by means of data flows.
- In general, when data must be sent from one action to another, regardless of their purpose, these two actions must be linked throughout the appropriate pins by means of a data flow.

- The flow outgoing from a “Value Specification Action” must always be a data flow.
- When an action provides the input for a decision node, the output pin of that action and the decision node must be linked by means of a data flow.
- A decision node must be connected to the first action of each outgoing branch by means of a data flow. In this case, that data flow must point to an input pin in the linked action, only if the linked action expects that input and features the necessary input pin. Otherwise, the data flow must point to the action itself.
- When an action provides the input data for a number of actions to be executed in parallel, the output pin of the action and the fork node must be linked by means of a data flow. In this case, each outgoing flow from the fork node must also be a data flow, and must point to the appropriate input pin in the action being linked.
- Decision nodes and fork nodes cannot feature more than one incoming flow link.
- All the branches outgoing from a decision node must be of the same type, and of the same type than the incoming flow.
- All the branches outgoing from a fork node must be of the same type, and of the same type than the incoming flow.
- Flows outgoing from an input parameter must be data flows.
- Incoming flows of an output parameter must be data flows.
- For the rest of cases not explicitly stated before, control flows must be used.

In general, when connecting actions to the elements used to specify how data and control will flow (initial or final nodes, decision or merge nodes, etc). Only the actions modeling the actual behavior of the activity should be considered (do not connect these elements to the actions used to provide context for other actions).

A.6 State machine specification

Specifying a state machine implies giving it a name, specifying its states, and specifying its transitions.

At the time of specifying states, the following apply:

- The SM must feature as many states as state blocks have been defined in the model.
- Each state in the SM must have the same name than the corresponding state block.

Additionally, the SM should feature an “Initial node” and, depending on the behavior being modeled, it might also feature a “Final node”.

At the time of specifying state transitions, each transition among two states, other than the initial node and the final node, must be linked to the boolean property triggering that transition. This is done by setting the “Event type” field to the value “ChangeEvent”, and by filling in the “ChangeExpression” field with an “ElementValue” expression corresponding to the property. For the transitions outgoing the initial node or reaching the final node, whether they must be linked to a property or not will depend on the behavior being modeled. Finally, the SM must be linked to the operation responsible for invoking its behavior.

B Model testing SysML profile

This SysML modeling methodology requires the presence of a SysML profile, in order to effectively create model testing-enabled models. In this Appendix we present this profile, along with some guidelines describing how to use it. The profile will help modelers to provide some specific data needed to steer the model execution process, as well as to define additional data types that are needed to interface with function models.

B.1 The «SUT» Stereotype

The «SUT» stereotype is used to identify the block representing the system under test. Only one block in the model must be stereotyped as «SUT». This stereotype includes two tagged definitions:

- taskSchedulerFunction: It sets the name for the routine that will contain the scheduler's code. That name should not collide with the names of any of the modeling elements defined in the stereotyped block, and must comply with the naming convention described in Appendix A.
- timeStepSize: It sets the scheduler time step size in milliseconds.

B.2 The «Schedulable» Stereotype

It is used to specify the SUT's tasks. This stereotype contains three tagged definitions:

- executionRateHz: It sets how many times per second the task must be scheduled. This value must be a multiple of "timeStepSize".
- executionOrder: It sets when the task must be scheduled in relation to the rest of tasks being scheduled. The order is global.
- estimatedCompletionTime: It sets the estimated duration of the execution of the task in milliseconds.

B.3 The «Initialization» Stereotype

It is used to specify auxiliary operations that must be executed only once, at the beginning of the co-simulation process. This stereotype contains one tagged definition:

- Order: It sets when the operation must be executed in relation to the rest of initialization operations. The order is global.

B.4 The «Background» Stereotype

It is used to specify the operations from the SUT environment blocks that must be executed alongside the SUT tasks, throughout the simulation time span. A typical example of background operation is an operation describing the behavior of the physical world block. This stereotype contains one tagged definition:

- Order: It sets when the operation must be executed in relation to the rest of background operations. The order is global.

B.5 The «Configuration» Stereotype

This stereotype is used to define a global configuration attribute needed for the co-simulation framework to work. Only the “model” block should be stereotyped as «Configuration». The tagged definition “functionalModelsPath” should reflect the location of the folder with all the function models the simulation depends on.

B.6 The «Matrix» Stereotype

This stereotype helps to define matrices, needed to interact with Simulink models. It can be applied to any property whose type is a primitive type with multiplicity 1. It defines three tagged values:

- numberOfRows: Number of rows of the matrix.
- numberOfColumns: Number of columns of the matrix.
- defaultValue: A comma-separated list of values enclosed between brackets.

B.7 The «Data» Stereotype

This stereotype is used to identify the data blocks in the model. It features no tagged values.

B.8 The «NotLoggable» Stereotype

This stereotype is used to control the verbosity of the execution trace produced by the co-simulation framework. By default, all the actions in ADs are logged. If an action is stereotyped as «NotLoggable» then its execution is not registered in the execution trace.

References

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

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

[3] OMG. OMG Systems Modeling Language (SysML) 1.4 Specification, 2015.