An Approach for Systematic In-the-Loop Simulations for Development and Test of a Complex Mechatronic Embedded System

Amir Soltani Nezhad(✉), Johan J. Lukkien, Rudolf H. Mak, Richard Verhoeven, and Martijn M.H.P. van den Heuvel

Mathematics and Computer Science Department,
Eindhoven University of Technology (TU/e), P.O. Box 513,
5600 MB Eindhoven, The Netherlands
{a.soltaninezhad,j.j.lukkien,R.H.Mak,P.H.F.M.Verhoeven,
m.m.h.p.v.d.heuvel}@tue.nl

Abstract. Simulations are widely used in the engineering workflow of complex mechatronic embedded systems in various domains, such as healthcare, railway, automotive and aerospace, for analyzing, testing and validating purposes. This paper focuses on the development and test of the control software of complex mechatronic embedded systems from the perspective of software interfaces (e.g., driver APIs) and presents a systematic approach for testing the control software during the various stages of an engineering process. Since we assume that the physical (hardware) components of an under-control plant could be replaced with simulation models, various kinds of in-the-loop simulations, ranging from MiL to HiL, can be consequently acquired. Additionally, we present a mathematical model of MESes required to formally describe the approach and also a healthcare case study to which our approach was applied.

Keywords: Development of mechatronic embedded systems · In-the-Loop simulations · Software-in-the-Loop (SiL) · Hardware-in-the-Loop (HiL) · Driver APIs

1 Introduction

A Mechatronic Embedded System (MES) typically consists of complex combinations of hardware and software components. Such a system is often software-intensive, and therefore the development and test or evolution of its control software is largely costly (time and money) over the development process. Due to this software complexity, manufactures and OEMs are interested in techniques and tools to shorten the development, verification and validation engineering workflow (typically V model-based), while maintaining or even improving the quality of the control software and in turn the entire product.

For instance, in recent years, simulations have been widely used during the development and test of MESes, especially for the control unit, in different domains, such as healthcare, railway [5], automotive [7], and aerospace.

© Springer International Publishing Switzerland 2016
A. Skavhaug et al. (Eds.): SAFECOMP 2016 Workshops, LNCS 9923, pp. 130–143, 2016.
DOI: 10.1007/978-3-319-45480-1_11

The key reason is to discover faults and inconsistencies with design in advance without postponing tests until the availability of implemented control software or of hardware components of the plant, which might be under parallel development. It can also be used for comparing different alternatives prior to taking a critical design decision. In complex embedded systems, there are well-know simulation techniques in which a control unit can be tested against its under-control plant, while different abstraction levels of the control and the components of the plant could participate in the simulations. Some examples of these techniques include Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Processor-in-the-Loop (PiL) and Hardware-in-the-Loop (HiL) simulations, which differ from each other based on the level of abstraction.

For instance, MiL is a technique, with the highest abstraction, in which there is no target hardware involved in the system and only simulation models are integrated for testing and validating purposes. On the other hand, HiL, among other definitions, is a simulation technique, with the lowest abstraction, in which a target control unit is tested while connected to an under-control, fully-simulated plant.

Deploying in-the-loop simulations, especially HiL, to accelerate the development and test of embedded systems in various domains and applications has been widely investigated [1–4]. However, these works mostly focus on in-the-loop simulations to validate different control algorithms in various domains, such as power electronic systems and mechanics.

In this paper, however, we are interested in looking at the problem of the development and test of the control software of a MES from the higher-level perspective of driver APIs of the plant. Through these APIs, a MES's control software controls the plant, typically via other sub-control systems. We consider these sub-control systems as part of the MES's plant. The in-the-loop simulation works in the literature mainly focus on validating these sub-control systems unlike this work that concentrates on the higher-level control software of a MES. To clarify more the research question, assuming that the under-development control software is going to be tested against the plant, given the driver APIs of the plant components, such as motors and sensors, we investigate a systematic way for testing such control software and its interaction with the plant through the APIs. It is essential to note that not all the components of the plant must be physically available. This means that we can arbitrarily replace any number of physical components with their simulated counterparts.

By solving this problem, we can consequently provide the engineers of a MES's control software with a mechanism by which various in-the-loop simulations in different abstractions could be realized.

Furthermore, observing the increasingly widespread adoption of the Model Based System Engineering (MBSE) paradigm, our solution could benefit from MBSE for the development of MESes. Because, since MBSE operates based on models throughout an entire engineering process, our approach could enjoy from MBSE by deploying, for example, early executable models of a plant for testing the early model of the control software.

A general idea of this paper was published in [6]. In this paper, we present the following contributions:

1. A mathematical model for a MES and running in-the-loop simulations, which can be found in Sect. 2.
2. A formal explanation of our methodology is addressed, which can be found in Sect. 3.
3. A detailed explanation of an industrial case study from the healthcare domain to which the methodology was applied, which can be found in Sect. 4.

Finally, Sect. 5 concludes this paper and presents the future directions of this work.

2 System Model

In this section, we introduce the key concepts of our system model (see Fig. 1 for an overview). The model considers that a MES is composed of components that interact solely via well-defined interfaces. For the scope of this paper, we assume that all systems are built from a known set of components taken from a fixed repository named $\mathcal{R}$.

Definition 1 (Repository). The repository $\mathcal{R}$ is a pair $(\mathcal{R}.\mathcal{I}, \mathcal{R}.\mathcal{C})$, where $\mathcal{R}.\mathcal{I}$ is a set of interfaces and $\mathcal{R}.\mathcal{C}$ is a set of components.

To enforce correct interaction patterns between components, some detail about their interfaces needs to be available.

Definition 2 (Interface). An interface $i \in \mathcal{R}.\mathcal{I}$ is a pair $(i.n, i.s)$ where $i.n$ is a unique name, and $i.s$ is a set of signatures (prototypes) of methods, i.e., method names with input/output parameters.

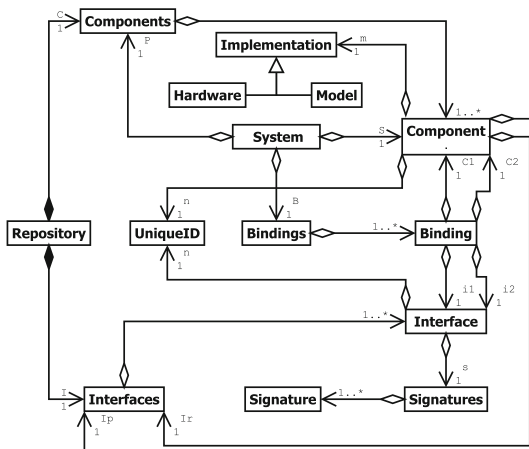

Fig. 1. System model overview

Occasionally, we need to express that sets of interfaces are identical apart from the names that make each interface unique. For this, we introduce the notion of signature equivalence.

Definition 3 (Signature Equivalence). Two sets of interfaces $I_1, I_2 \subseteq R.\mathcal{I}$ are *signature equivalent*, denoted by $I_1 \equiv_{sig} I_2$ when their bags of signatures are equal, i.e., when

$$\biguplus_{(n_1,s_1)\in I_1}\{s_1\} = \biguplus_{(n_2,s_2)\in I_2}\{s_2\}$$

A component uses interfaces both to provide functionality to its environment and to require functionality from other components. Thus, its provided functionality arises as a combination of its own implementation and the functionality obtained via the required interfaces.

Definition 4 (Component). A component $C \in \mathcal{R}.\mathcal{C}$ is a quadruple $(C.n, C.I_p, C.I_r, C.m)$, where $C.n$ is a unique name, $C.I_p, C.I_r \subseteq \mathcal{R}.I$ is the set of its provided and required interfaces, respectively, and $C.m$ is the C's implementation, which is a set of implemented artifacts. Moreover, we denote the set $(C.I_p \cup C.I_r)$ (i.e., all the interfaces) by $C.I$.

Based on the nature of its implementation, we distinguish two types of components.

Definition 5 (Component Type). A component is called a *hardware (physical) component* when its implementation is given by actual piece(s) of hardware, such as sensors or motors. It is, however, called a *simulated component*, if it is made of executable simulation model(s). A component whose implementation is of software but still need to be embedded in the targeted device is also called a simulated component.

The interaction patterns between the components of a system are captured by bindings. Each binding involves a pair of interfaces of opposite roles, such that all methods required by one component are provided by the other. However, within a binding, not all provided methods need to be required.

Definition 6 (Binding). A binding between components from $\mathcal{R}$ is a quadruple (C_1, i_1, C_2, i_2), where $C_1, C_2 \in \mathcal{R}.\mathcal{C}$, $i_1 \in C_1.I_p$, and $i_2 \in C_2.I_r$ such that $C_1.n \neq C_2.n$ and $i_2.s \subseteq i_1.s$.

For the purpose of an in-the-loop simulation, a MES is divided into two parts: control software[1] and a plant. In contrast to the control software component, which is of the simulated component type, the plant may consist of the both component types, simulated and hardware. Furthermore, in order to perform an in-the-loop simulation of a system, the bindings present between the control software and the plant must satisfy certain rules.

[1] In this work, our focus is on the software part of a MES's control than on the entire control unit.

Definition 7 (Valid System). A valid system is a triple (S, P, B), where $S \in \mathcal{R}.\mathcal{C}$ is a simulated component called the *control software*, and $P \subseteq \mathcal{R}.\mathcal{C} \setminus \{S\}$ is a set of components called the *plant*. In contrast to S, the components of P may be of either type, i.e., a plant may consist of both simulated and hardware components. Furthermore, B is a set of bindings such that:

1. $\forall_{i_r \in S.I_r} : \left[\exists_{C \in P, i_p \in P.I_p} : (C, i_p, S, i_r) \in B\right]$
2. $\forall_{i_r \in P.I_r} : \left[\exists_{C \in P, i_p \in S.I_p} : (S, i_p, C, i_r) \in B\right]$

where $P.I_p = \bigcup_{C \in P} C.I_p$ and $P.I_r = \bigcup_{C \in P} C.I_r$.

In terms of this system model, a simulation is an execution of a valid system in which one or more components of the plant are simulated components.

3 Methodology

In this section, we present our methodology that consists of the introduction of two (special) components to be inserted at predefined locations into the architecture of our approach. First, we state the objectives that must be achieved by these components. Next, we discuss the individual requirements imposed on these components, their locations in the system in the architecture, and typical usage in simulation.

The main consequence of our approach is to enable the executions of a variety of valid systems at various stages of a MES development, resulting in in-the-loop simulations at various levels of abstraction.

The first component we introduce is called the *Simulation Wrapper (SW)*. It will be inserted between the control software S and the plant P of a MES that has the capability of both tracing all traffic between S and P and redirecting or duplicating that traffic. Each system will contain a single SW, but its appearance will depend on the system in which it is inserted.

The second component we introduce is called the *Simulator Coordinator (SC)*. It bridges the gap between the executable simulation models (a.k.a., simulators) and interfaces of a component of the plant simulated by these models. Moreover, since most MESes are real-time systems, a simulator coordinator is also responsible for aligning the simulator's notion of time with the system time. In principle, we consider one SC per simulated component. Figure 2 depicts the general architecture of an in-the-loop system using this approach.

In the rest of this section, we discuss the components SW and SC in more detail.

3.1 Simulation Wrapper (SW)

As indicated above, the simulation wrapper is inserted between the control software S and the plant P. The rationale for this placement, as well as other possibilities for placement, has been discussed in [6]. Since the simulation wrapper is a component, i.e., $SW = (SW.n, SW.I_r, SW.I_p, SW.m)$, we need to specify both

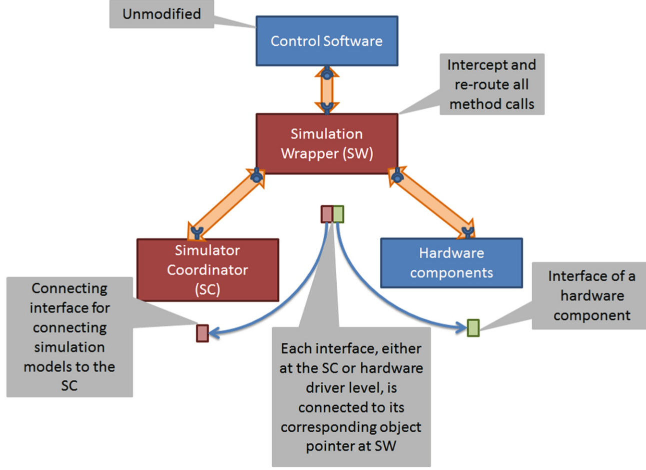

Fig. 2. Overall architecture. Our solution's components are highlighted in red. (Color figure online)

its interfaces and its implementation. Its interface sets are determined by the interface sets of S. For each required interface $i \in S.I_r$, it has three interfaces i_1, i_2, and i_3 that all have the same signature as i. Of these interfaces, i_1 is a provided interface that is intended to be bound to i. The other two are required interfaces that forward method invocations by S along with i to either a driver of a hardware component of the plant (i_2) or to the interface of the simulated version of that component (i_3). Similarly, there are three interfaces for each provided interface of S. Note that the provided interfaces of S correspond to call-backs from the plant. Thus, we have:

$$SW.I_p = SW.I_{p1} \cup SW.I_{p2} \cup SW.I_{p3}$$
$$SW.I_r = SW.I_{r1} \cup SW.I_{r2} \cup SW.I_{r3}$$

where

$$S.I_p \equiv_{sig} SW.I_{r1} \equiv_{sig} SW.I_{p2} \equiv_{sig} SW.I_{p3}$$
$$S.I_r \equiv_{sig} SW.I_{p1} \equiv_{sig} SW.I_{r2} \equiv_{sig} SW.I_{r3}$$

The implementation of SW is more difficult to specify in a formal manner. Therefore, we merely hint at the realization of its functional requirements. In addition, we mention non-functional requirements that such a realization should meet.

Functional Requirements:

– *Interception:* An important aspect of interception is that any invocation of a method from a provided interface $i_1 \in SW.I_{p1}$ needs to be forwarded to

its destination under simulation. For this, $SW.m$ must maintain references to its corresponding signature-equivalent interfaces $i_2 \in SW.I_{r2}$ and $i_3 \in SW.I_{r3}$. Listing 1.1 contains a code snippet that shows how this can be done by encapsulating such references in objects that possess the same interface. For the sake of simplicity, various details, such as how to deal with parameters that are themselves objects, or which object to return in case both the real and the simulated methods are invoked are left unspecified.
- *Traceability:* The same code snippet in Listing 1.1 illustrates how forwarding can be augmented with logging of invocation data.

Listing 1.1. Conceptual forwarding by the simulation wrapper

```
Interface1::MethodA(p) {
  if (Object2) { // forwards to i2
     RtnObj2 = Object2->MethodA(p);
  }
  if (Object3) { // forwards to i3
     RtnObj3 = Object3->MethodA(p);
  }
  if (Tracing_On) { // invoke custom logging
     LogInvocationData();
  }
  return ( ReturnFrom2() ? RtnObj2 : RtnObj3);
}
```

Non-functional Requirements:

- *Transparency:* To ensure that no modification in either plant or control software is needed, when running a new valid system, interception of method invocations needs to be transparent, i.e., neither the caller nor the callee should be aware of the existence of SW.
- *Small and Predictable Overhead:* To ensure reliable simulation results, SW must satisfy its functional requirements with small (preferably constant) and predictable overhead in terms of computing resources (e.g., CPU and memory), and must not cause unpredictable delay on method invocations between S and P.
- *Automatic Generation:* To support an efficient development process, generating and inserting the simulation wrapper into the architecture of a MES should be automated as much as possible. Because the interfaces of the components that make up a MES are available, and forwarding follows a standard pattern, this is, to a large extent, feasible.

3.2 Simulator Coordinator (SC)

As indicated, a simulator coordinator is responsible for connecting executable simulation models to the system. More specifically, it must take one or several simulation models as the implementation of a potential simulated component

from simulation tools such as Simulink, and attach interfaces to it. By doing so, a simulator coordinator, in fact, transforms one or multiple simulation models into a simulated component of P.

In a nutshell, any $SC = (SC.n, SC.I_p, SC.I_r, SC.m)$ is an access point for available executable simulation models that together simulate a component $C \in P$. Hence, it immediately follows that $SC.I_p \equiv_{sig} C.I_p$. For $SC.I_r$, the situation is more complex. In general, it consists of two sets of interfaces. One set assumes the role of the required interfaces of the plant's component under simulation. The other set of interfaces serves to connect the coordinator to the collection of models. If we assume that $M.I_p$ is the set of interfaces provided by the models, then $SC.I_r = SC.I_{r1} \cup SC.I_{r2}$, where $SC.I_{r1} \equiv_{sig} C.I_r$ and $SC.I_{r2} \subseteq M.I_p$. Figure 2 illustrates the overall architecture of an in-the-loop system in which simulators are connected to the simulation wrapper via a simulator coordinator.

To achieve its expected functionality, $SC.m$ must meet the following requirements:

- *Model Connectivity:* For each interface $i \in SC.I_p$ the methods of i must be implemented using the executable simulation models of $SC.m$ invoked through $SC.I_{r2}$.
- *Synchronization:* In general, a simulator simulates the behavior of a simulated component through a sequence of time-stamped state-transitions and associated events. For this, the simulator keeps track of a notion of logical time. In order to obtain a correct in-the-loop simulation, the logical clocks of the simulation models need to be synchronized with the real system-time. An example of how this can be done in practice is shown in the next section.

As with the simulation wrapper, the automatic generation of $SC.m$ is a desirable property. For the generation of the provided interface, this is, to a large extent, feasible. For the translation of interface methods into model methods, however, this is less likely, since it is highly dependent on the primitives of the simulation language and the plant component under simulation (Fig. 3).

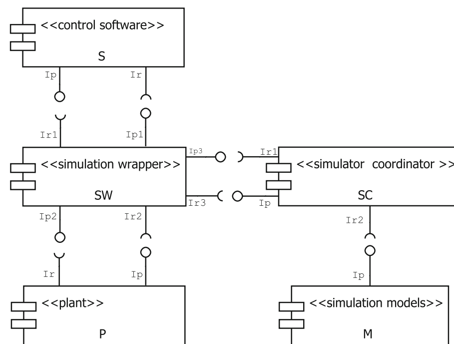

Fig. 3. All the Interfaces of the entire components of our solution. I in this figure actually represents any interface $i \in I$

4 Case Study

In this section, we illustrate the approach introduced in Sect. 3 on an industrial case study from the healthcare domain. More precisely, we explain the procedure required for configuring an SiL simulation for the case study using this approach.

We consider a safety-critical MES, viz., an *Interventional X-Ray (IXR)* machine (See Fig. 4). This is a case in which the control software manipulates quite heavy hardware components, such as a C-ARM and a patient table whose uncontrolled movements may harm patients or medical staff. In view of the cost of the machine and the mentioned safety aspects, there is a strong motivation not only to test the system with simulated hardware, but also to be selective on which hardware components are simulated.

To avoid being overwhelmed by unnecessary details of a complete IXR plant, we focus our attention on the simulation of a single component. In the sequel, we refer to the selected hardware component as C_2 and to its simulating counterpart, connected to the IXR system by means of a simulator coordinator SC and wrapper SW, as C_1.

For $C_2.I$, we selected an EtherCAT network driver, a third-party driver used by the IXR control software to control motors and sensors of an IXR over an EtherCAT network. Therefore, these motors and sensors are the implementation (i.e., $C_2.m$) of the component C_2. Since we intended to configure an SIL, C_2, as a hardware component, is absent.

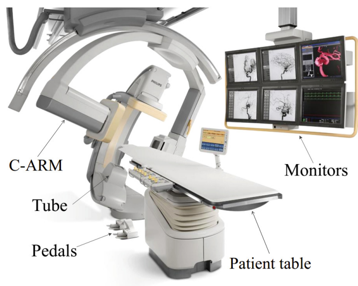

Fig. 4. An Interventional X-Ray (IXR) device

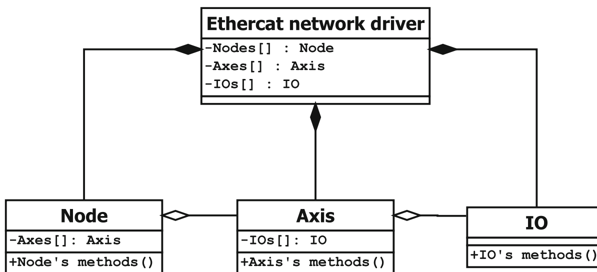

Fig. 5. Class diagram of the EtherCAT driver stub

Figure 5 depicts the structure of the EtherCAT driver's three main provided interfaces, namely, *Node*, *Axis*, *IO*. Hence, together they constitute the interface $C_1.I_p$ of the simulated component. For the latter's implementation $C_1.m$, we deployed a Matlab Simulink model that simulates a motor responsible for moving an IXR's patient table up and down and a sensor that measures the pressure on the patient table of an IXR. Both motor and sensor also simulate the original hardware controlled by IXR's S using the EtherCAT network driver.

To enable an SiL, the module of the IXR's control software S responsible for positioning the IXR's patient table, not yet embedded in its dedicated hardware, was used and the following objectives were pursued by this setup:

- To explore which type of motor is appropriate for the patient table, in terms of speed, power and other properties.
- To test whether the control software works properly.

Figure 6 shows the instantiated version of the generic architecture introduced in Fig. 2, for this particular case study.

SW: The main task of the simulation wrapper in this case study is forwarding method calls from S to C_1 through SC. Listing 1.2 shows the implementation of this task for a method named ProduceData() belonging to $C_1.I_p$.

The information determining which components are in P, introducing simulation models, if any, and their input and output parameters along with their critical functions, such as their step functions is obtained through a configuration file fed to each valid system. Using this file, prior to running a valid system, an initialization process is performed to bind relevant components, such as S to SW and SW to (simulated) components of P. For instance, the configuration file created for this case study causes the initialization process to bind C_1 to S via SC via SW. As a consequence, when receiving a method call from S, SW forwards the call to C_1 through SC.

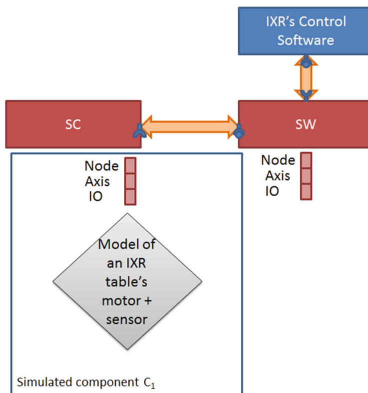

Fig. 6. Architecture solution for an IXR

As mentioned in Sect. 3, in addition to just forwarding calls, *SW* could also record these interactions for a later possible fault analysis (not yet implemented) and for testing purposes.

Listing 1.2. Main functionality of the simulation wrapper

```
if (C2_Object) {
// forwards to the physical component via the driver
RtnObj = C2_Object->ProduceData(p);
}
if (C1_Object) {
// forwards to the simulated component via SC
RtnObj = C1_Object->ProduceData(p);
}
```

SC: Recall that the purpose of the simulator coordinator is to solve two important generic problems: model connectivity, i.e., realizing simulated component interfaces using the methods offered by given simulation models' objects, and synchronization, i.e., aligning the passage of time in the control software with that in the simulation models. In addition, there may be other system specific issues. In this particular case, unit conversion is an example of the latter. We now discuss each of these in some detail.

Model Connectivity: As discussed earlier, the *SC* must expose the same interfaces as the Ethercat driver of the plant component whose simulation it coordinates. In our case, these interfaces are the Node, Axis and IO interfaces, each of which is implemented in *SC* by an object of a corresponding class. The actual implementation of the objects' methods is in terms of Simulink models. For this case study, the entire implementation has been done by hand, but using the configuration file described earlier in the context of the initialization process, a large part of the implementation of these classes, is skeleton code that can be generated automatically.

Furthermore, to have a structured mechanism to import the required Simulink simulation models, we made a design choice to encapsulate them in objects of a single class named *Model*. This class contains attributes and methods of a typical simulation model, like input and output variables, step size and step function. Thus, for every simulation model in $C_1.m$, an object of class *Model* is instantiated in the *SC* to be used by the Node, Axis and IO interface methods for implementation of the simulation proper. The resulting class structure of *SC* is shown in Fig. 7.

Synchronization: This issue addresses the difference in the handling of time between S and simulated components of P (here only C_1). The situation in our case study is as follows. The control software periodically loops through a sequence of control statements, whereas each simulation model steps through a sequence of states. For this, the models provide a function *stepFunc* that determines the next state and a parameter *timeStep* that indicates the advance in time associated with each step. These time steps are much smaller than the period of the control loop and, for the sake of simplicity, we assume in the sequel

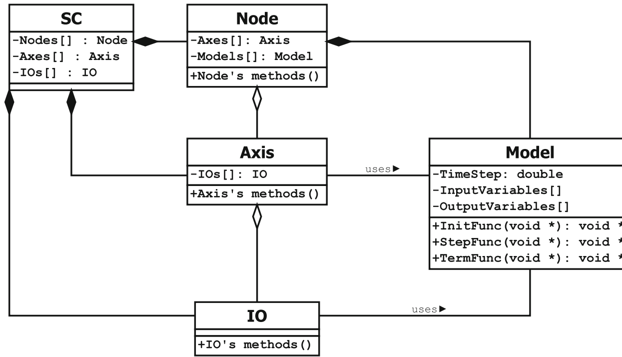

Fig. 7. SC class diagram for this case study

that the period of the control loop T_{ctrl} is a multiple of the time step of every model in $C_1.m$.

Listing 1.3 displays a variant of the actual control loop that captures the essential features, but uses fictional methods to simplify the explanation.

Listing 1.3. Basic control loop with fictional methods and period T_{ctrl}

```
control software loop{
    every (T_ctrl time units){
            readStatus();
            analyze();
            writeStatus();
    }
}
```

Listing 1.4 displays an idealized implementation of method *writeStatus()* $\in C_1.I_p$ whose intended effect is that all models in $C_1.m$ synchronize their state to the moment in time implicitly maintained by the control software state as it iterates through its loop.

Listing 1.4. Synchronization by writeStatus

```
writeStatus(){
for each (model m of C1){
        var dt = 0;
        while (dt  < T_crtl) {
                m.stepFunc();
                dt = dt + m.timeStep;
    }
 }
}
```

Unit Conversion: Besides its standard responsibilities *model connectivity* and *synchronization*, in this case study, the *SC* was also responsible for *conversion of values* exchanged between C_1 and S. This responsibility arose because the

simulation models in $C_1.m$ only work with the SI base units, whereas the control software S also works with SI derived units, such as volt. To overcome this difference, information about units occurring in both the simulation models and the system was gathered from the configuration file and used to add conversion functionality to the SC, e.g. $V = W/A = m^2 \cdot kg \cdot s^{-3} \cdot A^{-1}$.

5 Conclusion and Future Work

In this paper, we focused on the problem of the development and test of the control software of MESes from a high level perspective of software interfaces such as driver APIs of the under-control plant. The idea is that given the interfaces of the components of a plant, transparently to the control software, engineers have freedom to provide these interfaces using either hardware components of the plant or their simulated counterparts. As a consequence of solving this problem, various in-the-loop simulations, such as SiL and HiL, required for verification and validation of MESes are realized. Note that this paper does not fully solve the problem of mixed simulations (a special interpretation of HiL) where a plant consists of simultaneously hardware components and simulated ones.

It is important to note that our approach's components, especially SW, does not only play an essential role in the development and test of a MES, but it may also exist in the final product for serving different purposes, for example for logging interactions between control software and its plant for fault analysis.

The work in this paper could be extended in three directions. First, more industrial case studies from different domains to be studied for ensuring the applicability of this approach in other domains. Second, investigating the problem of the mixed simulations where there is the freedom of integrating hardware components and simulated ones as a plant using this approach. Third, adding a domain-specific language for this approach in order to extend it and make it as a comprehensive framework with high amount of code generation, especially on the SC side. This is because we believe that the more automated this code generation is, the more valuable the solution is, and it can be easier integrated into current development workflows of MESes.

Acknowledgments. This work was supported in part by the European Union's ARTEMIS Joint Undertaking for CRYSTAL (Critical System Engineering Acceleration) under grant agreement No. 332830.

References

1. Choi, S.B., Young, T.C., Park, D.W.: A sliding mode control of a full-car electrorheological suspension system via hardware in-the-loop simulationg. Dyn. Syst. Meas. **122**(1), 114–121 (2000)
2. Faruque, M., Dinavahi, V.: Hardware-in-the-loop simulation of power electronic systems using adaptive discretization. IEEE Trans. Ind. Electron. **57**(4), 1146–1158 (2010)

3. Hui, L., Steurer, M., Shi, K.L., Woodruff, S., Zhang, D.: Development of a unified design, test, and research platform for wind energy systems based on hardware-in-the-loop real-time simulation. IEEE Trans. Ind. Electron. **53**(4), 1144–1151 (2006)
4. Isermann, R., Schaffnit, J., Sinsel, S.: Hardware-in-the-loop simulation for the design and testing of engine-control systems. Control Eng. Pract. **7**(5), 643–653 (1999)
5. Scippacercola, F., Pietrantuono, R., Russo, S., Zentai, A.: Model-driven engineering of a railway interlocking system. In: Proceedings of the Third IEEE International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 509–519, February 2015
6. Soltani Nezhad, A., Ferreira, L.F.B., van den Heuvel, M.M.H.P., Verhoeven, R., Lukkien, J.J., Mak, R.H., Korff de Gidts, E.: Towards an interoperable framework for mixed real-time simulations of industrial embedded systems. In: Proceedings of IEEE International Conference on Emerging Technology and Factory Automation (ETFA) (2014)
7. Sung Chul, O.: Evaluation of motor characteristics for hybrid electric vehicles using the hardware-in-the-loop concept. IEEE Trans. Veh. Technol. **54**(3), 817–824 (2005)