Formalization of Cyber-Physical System Interface using Discrete Event System Specifications

by

Rishabh Sudhir Jiresal, B.Eng.

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Cyber-Physical systems are complex engineered systems that integrate embedded computing into the physical environment. These systems are considered to be “safety-critical” due to their application in the field of medical, transportation, or building control. CPS is composed of tight coupling of the physical and cyber worlds, and the interfaces are interactions that are defined as a bridge between these worlds. The interface is a fundamental characteristic of a CPS and CPS cannot function without it.

In this thesis, we propose an architecture of the CPS interface in Discrete EVent System Specification (DEVS) that mimics the functionality of CPS interactions. DEVS provides a formal platform for M&S of discrete event dynamic systems. We propose a DEVS simulation model named DCIF (DEVS CPS Interface Framework) that portrays the complete working of the CPS interface. Later, we also implement and evaluate this interface on real-time hardware. The architecture is verified by creating a synthetic environment that includes multiple test cases in the simulation as well as real-time implementation. Additionally, we apply this framework to a practical case study in the field of building information modeling.

Acknowledgments

Foremost, I would like to thank my family: my parents Rajashree and Sudhir, my brother Rahul and his wife Rayna, and my sister Radhika for continuous spiritual support and unparalleled love. I am forever indebted to my parents and siblings for giving me the opportunities and experiences that have made me who I am. This journey would not have been possible without them, and I dedicate this milestone to them.

I would like to express my sincere gratitude to my supervisor, Dr. Gabriel Wainer for the continuous support of my M.A.Sc. study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me all the time during the research and the writing of this thesis. I could not have imagined having a better supervisor and mentor for my M.A.Sc. thesis. The past one and half years have been a challenging but rewarding experience.

I would also like to thank my colleague, Joseph Boi-Ukeme, for his assistance in the collaborative work presented in this thesis. He provided great support during this work. Also, my deepest thanks to my close and supportive friend, Ritika. Finally, I would like to express my gratitude to all the members at the Advanced Real-time Simulation Lab for being supportive.

Table of Content

List of Figures

List of Tables

List of Appendices

List of Acronyms

Acronyms	Full Description
ALRT	Action Level Real-Time
API	Application Programming Interface
BEM	Building Energy Modeling
BIM	Building Information Modeling
CAD	Computer Aided Design
CO	Carbon Monoxide
CPS	Cyber-Physical Systems
DCIF	DEVS CPS Interface Framework
DEMES	Discrete Event Methodology for Modeling and simulation of Embedded Systems
DEVS	Discrete Event System Specification
DEVSRT	Discrete Event System Specification Real-Time
ECU	Electronic Control Unit
EIC	External Input Coupling
EOC	External Output Coupling
FDD	Fault Detection and Diagnosis
GSL	GNU Standard Library
HILS	Hardware-in-Loop Simulation
HLA	High Level Architecture
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation and Air Conditioning
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IMU	Inertial Measurement Unit
LQFP	Low Profile Quad Flat Package
LVM	Large Volume Metrology
MEMS	Micro Electromechanical Systems
MISO	Master In Slave Out
MOSI	Master Out Slave In
OTG	On The Go
PCA	Principal Component Analysis
PDEVS	Parallel Discrete Event System Specification
PID	Proportional Integral Derivative
PIR	Passive Infrared
PPM	Parts Per Million

RGB	Red, Green Blue
ROS	Robot Operating System
RPM	Rotations Per Minute
RT	Real-Time
SI	International System of Units
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Chapter 1: Introduction

Cyber-physical systems (CPS) are next-generation engineered systems that integrate embedded computing into the physical environment. Cyber-physical systems are embedded systems where the importance is to be more on an intense link between the computational and physical elements. CPS has been defined by various researchers from different perspectives. For instance, (Marwedel 2006) defines CPS as "embedded systems together with their physical environment" while (Helen 2008) describes CPS as "physical, and engineered systems whose operations are integrated, monitored, and controlled by a computational core". CPSs are made up of the physical world, interfaces, and cyber world (Gunes 2014). Some examples of the application of CPS include air traffic control systems, medical devices, and equipment, railway cross control systems, automotive airbag, smart homes, building systems, etc. (Khaitan and McCalley 2015).

The physical components of a CPS are responsible to convert physical quantities from one form to other and interact with the cyber world for analog/digital signal transfers. The cyber-world, on the other hand, consists of embedded computational devices that process information and communication in a virtual environment. This world is often referred to as the computational core. The interfaces are interactions that are defined as a bridge between the cyber world and the physical world.

CPS consists of two interfaces – physical interfaces and cyber interfaces (Fromel 2016). The sensors and the actuators are physical devices that perform physical interfacing tasks. While the embedded computational devices interface at a cyber level with the exchange of information. These physical and cyber interactions are fundamental characteristics in a

CPS and without these interactions, the CPS cannot function. There is a variety of research on the development of sensors, actuators, and embedded computational systems; both as individual entities and as a whole. But it is also important to study the interfaces in the CPS.

CPSs are regarded as “safety-critical”. A “safety-critical” system consists of the hardware, software, and control required to perform one or more functions, the failure of which would result in a significant increase in the safety risk for the people or environment involved. (Kaur 2018) provided extensive literature on these safety-critical system applications. They submit that CPS are dependable systems, but the complete dependability analysis of such systems is complex. However, the dependability of a CPS relies on the coordination of the interactions in the system. For instance, the data gathered from the sensors, which is a cyber-physical interface, influence the decision-making related to its control actions. Improper interactions in the CPS could lead to life loss, health hazards, or environmental hazards (Fromel 2016).

The use of formalization approaches in software and hardware design in engineering is motivated by the expectation that performing the appropriate mathematical analysis can contribute to a design's reliability and robustness (Holloway 1997). Hence, we believe that the behavioral issues in the CPS can be mitigated if the system interactions/interfaces are formally defined.

Modern CPS employs a fairly large number of complex standards. The software and hardware platforms with well-defined and appropriate levels of abstractions and architecture are essential for the development of reliable, scalable, and evolvable cyber-physical systems (Kim and Kumar 2012). A traditional design methodology where the

system is developed at a low level directly on the hardware cannot be used in case of such complexity. Hence, there is a strong need to evaluate the performance of the system under various conditions before they can be implemented on the hardware.

Modeling and simulation (M&S) provide an approach to verify the design and implementation details of such complex embedded applications (Moallemi and Wainer, Modeling and simulation-driven development of embedded real-time systems 2013). M&S also provides a risk-free testing environment to verify different conditions before any hardware implementation is done. In addition to that, it also reduces the cost and time required to develop new standards.

One of the techniques that we are interested in studying is called Discrete Event System Specification (DEVS) (B. P. Zeigler 1976). DEVS is a type of M&S technique that is used to model systems in a wide range of scenarios. It allows the system to be modeled as a series of discrete events occurring over time. We also studied that the CPS interactions in the interface possess discrete-time properties and they are time-synchronous and hence, we have used Discrete Event System Specification (B. P. Zeigler 1984). In addition to that, it was found that there is no formal definition of the CPS interface in DEVS.

The goal of this thesis is to formally define an architecture for the CPS interface in DEVS. The intent of this architecture is to allow the DEVS model to mimic the functionality of a CPS interface. To do that, we propose a DEVS simulation model named DCIF (DEVS CPS Interface Framework) that portrays the complete working of the CPS interface. Later, we also implement and assess this interface on real-time hardware. The architecture is checked by creating a synthetic environment that includes multiple test cases in the simulation as well as real-time implementation.

The main contribution of this work is the definition of DCIF model which is a discrete-time model that simulates and implements the formal working of the CPS interface. In this model, we implement the basic functionality of a CPS interface to portray the interactions in the system. Moreover, the model supports some additional features to improve the dependability of the system. We also define a formalized message structure for cyber interactions in the system.

As discussed earlier, the interactions/interfaces are responsible for the behavioral changes in the CPS. The CPS interface is a fundamental part of the CPS. It would be difficult to enhance the performance of the entire system without a detailed study of the CPS interface. Therefore, we build this model to provide an experimentation platform for a thorough study. We define the architecture using the DEVS formalism. The defined architecture underwent rigorous testing to verify the model's robustness and dependability through a large number of experiments using the Cadmium simulation tool. Then, we defined a different version of the model on a target real-time embedded hardware using an extension of the Cadmium simulation tool – RT-Cadmium.

Another contribution is that DCIF also provides a mechanism to define sensor fusion algorithms into a DEVS platform. We experimented with this method and defined a competitive sensor fusion algorithm defined in (Hongyan 2009) that uses the concept of principal component analysis (PCA). We defined this algorithm in DEVS using multiple types of sensors. The authors of (Boi-Ukeme and Wainer, A Framework for the Extension of DEVS With Sensor Fusion Capabilities 2020) presented a sensor fusion framework that facilitates the implementation of the sensor fusion algorithms within the DEVS simulator. However, the framework defined was not implemented on the target hardware potentially

due to uncertain sensor input types. The introduction of DCIF allowed an efficient implementation due to a formal definition of the input/output messages leading to an extension of the sensor fusion framework.

We verified the proposed architecture of DCIF by implementing a practical application that adapts this architecture in the field of building systems, which include sensors, actuators, and control devices, including a tight coupling of hardware and software features. Advanced building systems collect data to improve building performance, operations, and maintenance. Using Building Information Models (BIM) a large amount of historical data or simulated data can be stored. The introduction of DCIF contributes towards enhancing the measured data reliability through sensor fusion, data storage and analysis, and data visualization on BIM. We also show a prototype application using a digital twin of the Carleton University campus which has been built for virtual experience and building performance analysis where we use DCIF for data acquisition and storage.

However, the introduction of DCIF is not limited to the field of building modeling. The proposed DCIF model provides the same features – data reliability, sensor fusion, data storage, analysis, etc. if applied to other applications such as autonomous cars or multisensory robotics systems. From a developer's perspective, DCIF provides a platform to test the control algorithm by simulations before it can be implemented on the control hardware. Also, it provides a comparatively easy way to implement the model on the real-time hardware since we simplify how the model accesses the hardware – through DCIF Drivers discussed in Chapter 3.

1.1 Thesis Organization

The rest of the thesis is organized as follows. In chapter 2, a brief background of the CPS is provided. We discuss the CPS with respect to the design approaches, reliability of the CPS, and some critical applications. In addition, we present previous work done in the field of simulation and modeling including the hardware-in-the-loop simulations. We also provide a background on Discrete Event System Specification (DEVS) and Real-Time Discrete Event System Specification (RT-DEVS) formalisms. We conclude this chapter with a description of how RT-DEVS can be used in building control applications. In chapter 3, we introduce and formally define the DEVS CPS Interface Framework. Chapter 4 consists of results and case study applications of the described model in chapter 3. Chapter 5 concludes the thesis and presents the future possible work.

Chapter 2: Background

In this chapter, we provide a background on Cyber-Physical Systems, Discrete Event System Specification (DEVS), Real-Time Discrete Event System Specification (RT-DEVS), and Hardware-in-Loop Simulation (HILS). We also present related previous work done and applications of RT-DEVS and HILS in the development of cyber-physical systems. We discuss how the development of the cyber-physical embedded systems was improved by using a model-based development approach. In addition to that, we provide a background on how RT-DEVS can be used to integrate cyber-physical systems with 2D and 3D environments using the concept of building information modeling (BIM).

2.1 Cyber-Physical Systems

Cyber-physical systems are physical and engineering systems that are monitored, coordinated, controlled and integrated by a computing and communication core. It is a system that bridges the cyber-world of computing and communications with the physical world (Rajkumar, et al. 2010). Engineered systems are used in many societally critical application domains such as medical, transportation, and building control. These systems can be designed and developed to be much more smart, secure, dependable, and robust by using real-time embedded systems for distributed sensing, computation, and control over wired or wireless communication networks, multi objective optimization, high-level decision-making algorithms, and formal verification technologies (Kim and Kumar 2012). Cyber-physical systems are embedded systems where the importance is to be more on an intense link between the computational and physical elements. As a research challenge, the

authors of (Kim and Kumar 2012) mention that the cyber-physical systems are complex. The software and hardware platforms with well-defined and appropriate levels of abstractions and architecture are essential for the development of reliable, scalable, and evolvable cyber-physical systems. Some examples of cyber-physical system applications include automotive systems, manufacturing, medical devices, military systems, assisted living, traffic control and safety, process control, power generation, and distribution, energy conservation, HVAC, aircraft, instrumentation, water management systems, trains, physical security, asset management or distributed robotics.

Modern cyber-physical systems vary in their characteristics, applications, and levels of operations. To address these diverse requirements, several researchers have proposed different methods and design architecture. (Khaitan and McCalley 2015) discusses the classification of the modern cyber-physical systems and surveys their design techniques and applications. According to the authors of (Khaitan and McCalley 2015), the current research work in the field of cyber-physical systems can be classified into three categories: Design, Aspects/Issues, Applications. That is, these are the motivations for the research in the field of CPS improvements. Since a cyber-physical system is a "system of systems" in which complex and heterogeneous systems interact in a continuous manner. Hence, the overall architecture of the cyber-physical systems must be carefully designed. For instance, to address this need, the authors of (Y. Tan 2009) presented an architecture for CPS based on the temporal and spatial properties of the events. The authors represented an event as a function of attribute-based, spatial and temporal event conditions. Their framework allows conducting formal spatial and temporal analysis of the CPS. Also, in (E. Lee 2010) the authors presented two complementary approaches for the design of CPSs – “Cyberizing

the physical" and "physicalizing the cyber". As the complexity increases, the difficulty in specifying the interaction of cyber and physical systems also increases, and hence, using these approaches helps in bridging their differences for achieving more realistic modeling of CPSs. In addition to that, more researchers have proposed several designing techniques, semantics, and programming tools to design the CPS.

Due to the features provided by the CPS, there is multiple application of CPS in various fields. To mention an example of an application in the field of smart homes and buildings, the authors of (X. Li 2011) discussed smart community architecture. In smart homes, the sensors and actuators are configured in such a way that they can be remotely accessed or controlled through the internet. This forms an Internet-of-Things. Through this, the activities in the area can be monitored. This close surveillance provides improvements in community safety, health care quality, and home security. The authors of (A. S. al. 2011) discussed that the application of CPS in smart buildings provides rapid access to information which leads to a reliable interaction with the activities in the building.

CPSs are typically expected to be available at all times, to display appropriate behavior, and to survive failures. CPS that can achieve this, is known to be reliable. But upgrading the CPSs or correcting their faults is challenging. Reliability of the sensors has become not only the main focus of ongoing research but also the priority of research and innovation actions (RIA) supported by Horizon 2020 program (Castaño, et al. 2019).

Within a CPS, the data gathered from the sensors influence the decision-making related to its control actions. The resulting decisions are materialized using actuators and thus the loop between the physical and cyber components is closed. Given the importance of the data at this level, it must be accurate, accessible, timely, and consistent. These four aspects

make the data reliable (Sanislav, et al. 2019). The research conducted on the reliability of CPS can be classified into three aspects: data reliability, reliability assurance, and reliability analysis. For instance, the authors of (Abbas W 2016) highlighted their work on data reliability. According to the authors, any malicious cyberattack on the sensors can have a severe impact on the physical processes in the CPS leading to a negative impact on the CPS reliability. Hence, they formalized the methodology for attack resiliency, and they presented three approaches to achieve it: reliable sensor placement, reliable sensor network topology, and reliable data aggregation algorithm. The authors of (K. Xiao 2008) presented their work on reliability assurance by proposing a new layer which is called the "coordination layer" for CPS. This coordination layer provided management of fault-tolerant schemes of critical services in supervisory control and data acquisition system using a model. This model coordinates different critical services when faults caused by a cyber-attack occur in the system.

The authors of (Katipamula and Brambley 2005) presented their work on reliability assurance and analysis using fault detection and diagnosis (FDD) and applied their idea in the implementation of building systems. The primary objective of an FDD system is early detection of faults and diagnosis of their causes along with correction of the faults before additional damage to the system or loss of the service. FDD is described as consisting of three key processes: fault detection, fault identification, and fault isolation. To detect a fault, the physical system or a device should be closely monitored. When a fault is detected, fault diagnosis is used to evaluate faults and their causes. Fault diagnosis can be overly complicated in the CPS since it is difficult to distinguish faults from uncertainty. Hence, to maintain the system stability, we need to have a concrete study of the differences between

the faults and uncertainties. In (L. T. al. 2010), the researchers conducted a trustworthiness analysis of the sensors in CPSs. Their method helps in filtering out the noise from the input to CPS and hence reducing the false alarm issues caused by the uncertain faults. They introduced a method that implements a score-based trustworthiness analysis on the sensor data. Based on the fault diagnosis, the identification of the fault is conducted. It consists of a decision on how we can respond to the faulty scenario. When the fault identification is complete, an isolation procedure can be implemented in the system to tolerate future faults. This three-step process is together known as fault diagnosis. The authors of (Boi-Ukeme, Ruiz-Martin and Wainer 2020) presented a new scheme to detect and diagnose these CPS faults. This scheme was based on the combination of knowledge-based and model-driven FDD that proved to be an effective way to study the faults in CPS.

As discussed previously, the data reliability in the CPS is important since the data gathered from the sensors influences the overall behavior of the system. Hence, to make the data more reliable, data fusion or sensor fusion improves the data authenticity and availability. The data fusion or sensor fusion process involves association, correlation, estimation, and combination of the data and the sensor information from several sources (Khaleghi, et al. 2013). Performing data fusion or sensor fusion has several advantages as mentioned by the authors of (D.L. Hall 1997) that involve data authenticity and availability. For instance, improved detection, confidence, reliability as well as a reduction in data ambiguity. Data fusion also helps in the reduction of redundant data leading to energy consumption reduction in the system. For instance, in multi-sensor CPS with a substantial number of nodes connected in the sensor network, transmitting all the data from sensors can lead to data collisions and more energy consumption due to the redundant size of the data. This

leads to the implementation of an unreliable system. However, if data fusion is performed during the transmission process, the sensor data is fused and only the results are sent reducing the amount of data traffic in the system and leading to energy saving. The authors of (D.L. Hall 1997) also mention problems in multisensory data fusion.

The sensor fusion methods can be divided into three categories:

1) Competitive fusion: Multiple sensors of the same or different types are used to measure the same physical quantity.
2) Complementary fusion: Each sensor is used to measure different properties of the same object.
3) Cooperative fusion: The operation of a single sensor is dependent on the results of some other sensor.

In (Koval 2001), the researcher discusses a competitive sensor fusion algorithm and implements and evaluates it using MATLAB Image Processing Toolbox. The findings show that fusing the sensors' measurement can provide more accurate measurement results than that the results provided by a single sensor. Article (Raol 2010) introduces a sensor fusion toolbox to implement the sensor fusion algorithms on MATLAB which is a useful tool, but it can be complex to use. Article (Noordin 2018) discusses a complementary sensor fusion algorithm that outputs smooth roll, pitch, and yaw altitude angles for a quadrotor. This complementary sensor fusion algorithm fuses the values from the gyroscope, accelerometer, and magnetometer to provide a dependable output. The authors of (Galetto 2015) presented a cooperative sensor fusion algorithm where they find out the 3D position measurements using cooperative sensor fusion in a Large Volume Metrology (LVM) i.e., in the measurement of large objects such as cars, planes, or ship parts.

According to the authors, cooperative fusion is difficult to implement compared to competitive sensor fusion however cooperative fusion makes efficient use of the available information resulting in improved metrological performance.

In (Boi-Ukeme, Ruiz-Martin and Wainer 2020) the authors have developed a generic FDD framework for building CPS using Discrete Event Methodologies that is capable of detecting and isolating faults in real-time. And (Boi-Ukeme and Wainer 2020) discusses a framework that implements sensor fusion capabilities to DEVS. To implement the interface framework developed using DEVS in this thesis, we use these references as a good starting point, and we extend it to a significant margin.

Embedded system design is continuously undergoing improvements to meet performance, quality, safety, cost, and time-to-market. According to the authors of (Ferrari and Sangiovanni-Vincentelli 2001), the overall goal of an electronic system design or embedded system development is to minimize the production cost, development time, and cost subject to constraints on the performance and functionality of the system. (Gajski and Vahid 1995) discusses different system design techniques and proposes a new methodology for embedded system design based on the hierarchy of models at various levels of abstraction.

There are several traditional approaches used for the development of embedded system design. For instance, (Sangiovanni-Vincentelli and Martin 2001) discuss a platform-based development of embedded systems. The authors developed a design methodology to push away the company manufacturers from designing custom ICs directly and encourage software and hardware reuse. (Subramanian and Chung 2000) discusses an out-in development methodology in which the core functionality is developed together with

interface software that is used specifically for testing the core functionality. The authors of (Smith, Miller and Daeninck 2009) introduced an agile test-driven approach in the development of embedded systems where they implemented an Embedded xUnit testing framework to test the system. And (Nebut, et al. 2006) used the same test-driven paradigm but with generating the test cases according to the use case scenarios of the system.

Embedded systems development should have low development time, reduced use of resources, and low cost. It is difficult for traditional approaches to achieve all of these goals. In addition, the complexity of large-scale embedded systems is high and traditional approaches make the implementation complicated.

Complex embedded systems must perform specific tasks under timing constraints. These tasks need to be scheduled to guarantee real-time performance. To do so, we need to perform an analysis of the temporal behavior in the system to verify safe and predictable run-time system operation. As the entire system can fail due to missed timing deadlines or delayed responses, research has developed designs to analyze the timing behavior. For instance, using automata theory, a timed automaton is a finite automaton that is extended with a finite value of a real-valued clock. It can be used to model and analyze the timing behavior of an embedded computer system. Timed input/output automata are a class of timed automata with deterministic behavior, the separation between inputs and outputs, and input enabling. In (Giambiasi, Paillet and Chane 2003) the authors have presented a formal transformation of timed input/output automata into a discrete event model to verify using simulation. To add, the authors of (Springintveld, Vaandrager and R. D'Argenio 2001) provide a test suite derivation algorithm for a black-box conformance testing of timed input/output automata.

2.2 Hardware-In-Loop Simulation

Hardware-in-the-loop simulation (HILS) is a type of real-time simulation. HILS is a method to test a control algorithm of the system while it is functioning on the intended target hardware by creating a virtual real-time environment that replicates the physical system to be controlled (Bin Lu 2007). HILS is used to verify the correct implementation of the system design, performance forecasting, and to validate the experimental model. Hence, during the system design using HILS, we include all the possible actual parts of the system in the loop (Carrijo, Oliva and Leite Filho 2002). In HILS the digital program is broken into two parts, onboard computer with control laws and simulation of the rest of the system. Once the simulation is successful, the hardware interfaces are included (Köhler 2011). A typical example of HILS is a connection between a complete ECU (including a microcontroller) and a simulated building control environment on a building information modeling software. The reasons for the development of HILS systems are:

- Safety-critical scenarios can be assessed without compromising hardware or people.
- Test automation can be done.
- HIL tests are cost-efficient.
- A virtual environment can be easily modified, so a real control can be tested in different control loops.
- Tests are immediately repeatable.
- Hardware can be tested in parallel.

- Embedded real hardware can accelerate the whole simulation and implementation process.
- An early-stage development can be done and tested before implementation

HILS and discrete event modeling are co-related to each other since the discrete event modeling provides a formalism and communication between two atomic models or coupled models. The desire to have a seamless migration from initial development simulations to deployed systems in the field is referred to as 'model continuity' (Hosking and Sahin 2009). A four-step process for general model continuity: conventional simulation, real-time simulation, HILS, real implementation, and execution (Hu and Zeigler 2005). The HILS step allows part of the final system to be implemented and debugged before more resources are directed towards a flawed product.

There are numerous applications of HILS; for instance, the use of LabVIEW as a HILS platform was introduced in (A. M. Astorga 2011) to test the control algorithms for the function of an autonomous marine vehicle. The use of this tool helped in avoiding placing human lives at risk. In (Irwanto and Artono 2018), the authors discussed the development of a real flight vehicle using the HILS approach. They found that this approach provided convenience (time, cost, labor, etc.) in several procedures and settings in the real flight test. Also, in (Pratt, et al. 2017) and (Barreras, et al. 2016) HILS has proven to be critical. The development of the HILS system has been conducted to reduce the cost and impacts of evaluating the effect of advanced controllers such as Air Conditioners Model Predictive Control (Pratt, et al. 2017) or Battery management system (Barreras, et al. 2016).

For the development of simple embedded systems, we can use a traditional design approach where the systems are typically designed directly at low-level hardware. However, it

becomes difficult to use these methods for complex and highly integrated systems. Therefore, due to a higher level of integration, a model-based approach is highly recommended to develop more complex systems easily (Hong, et al. 1997).

2.3 DEVS and RT-DEVS

According to the authors of (Hong, et al. 1997), traditional real-time embedded software development methods have a significant limitation because it uses different tools and environments for each process step. Therefore, they introduced a formal definition for the design of embedded hardware and software using a model-based approach. This formal definition is an extension to Discrete Event System Specification (DEVS) formalism, which was developed by Zeigler, B. P. in 1984. They also proposed a basis for a real-time software development framework that bridges the gaps between analysis and implementation, namely RT-DEVS.

A DEVS atomic model is formally described by:

$$AM = <X, S, Y, \delta int, \delta ext, \lambda, ta>$$

where X is a set of input values; S is the set of sequential states; Y is the set of output values; and δint, δext, λ and ta are the internal transition, external transition, output, and time advance functions, respectively.

The definition of DEVS formalism allows the clear separation between the simulator and the model. This separation is important when designing models, which only requires providing an implementation of the δint, δext, λ, and ta functions, without worrying about invoking them or advancing the simulation time. These tasks are conducted by the simulator. At any given point during the simulation, each DEVS model is associated with

a state s, where s ∈ S. When the model receives an input x (x ∈X), the simulator invokes the external transition δext on the model and uses the time advance function ta to determine the time until the next internal transition. After that time elapses, the output function λ is used to obtain an output value y (y ∈Y), and the internal transition function δint generates the new state of the model.

As previously mentioned, a DEVS coupled model can be composed of one or more atomic or coupled models. A coupled model is formally defined as:

$$CM = <X,Y,D,\{Mi\},EIC,EOC,IC,Select>$$

where X is a set of input events; Y is the set of output events; D is the set of indices that refer to the submodel components that make up the coupled model. For each i in D, MI is a component of the coupled model. EIC refers to the list of external input couplings, the links between the input ports of the coupled model and the submodels. EOC refers to the list of external output couplings or the links between the submodels and the output ports of the coupled model. IC refers to the list of internal couplings, the links interconnecting the submodels. Finally, the Select function is used to determine the order in which the components undergo an internal transition in the event of a tie.

The real-time DEVS formalism extends the DEVS formalism in three perspectives: The real-time execution of models, the addition of time interval functions, and the activity specification for each state. Using this extension, a system can be specified, verified, simulated, and executed on real-time hardware-based on DEVS formalism.

In (Moallemi and Wainer 2013), the authors presented a modeling and simulation-based method which they refer to as DEVSRT. This helps in achieving model continuity between the simulation models and the final embedded application. They submit that M&S

(Modeling and Simulation) provides a cost-effective, dynamic, and risk-free-testing environment approach to verify the design and implementation of complex real-time applications. They extend the CD++ tool developed for DEVS simulation and modeling to form Embedded CD++ (E-CD++). They implemented the proposed DEVSRT framework for the development of embedded real-time applications in E-CD++. In (Wainer 2015), the author introduced a model-driven framework to develop cyber-physical systems based on DEVS formalism. This brought in a reliable way to incrementally develop embedded applications by integrating simulation with the hardware components.

In the development of some control system applications, it is important to meet strict real-time deadlines to complete tasks. Some of the examples of these control system applications are chemical and nuclear plant control, robotics, transportation control system applications such as railway and flight controls, etc. These applications are considered "safety-critical" real-time control systems. The authors of (Darragi, El-Koursi and Collart-Dutilleul 2014) focus on using the RT-DEVS formalism in these safety-critical real-time control systems, primarily in the transportation domain. They compare an implementation of an automated traffic control system using the general approach and RT-DEVS approach. Through the case study of air traffic control system, they show that model checking is enabled by a mapping of RT-DEVS behavioral models onto timed automata of UPPAAL (Furfaro and Nigro 2008).

Another key area of research is developing scheduling policies for computing platforms. This is where simulating safety-critical systems can be performed as close as possible to physical time. Hence, the authors of (Zhang, et al. 2018) proposed a frame-based real-time scheduling strategy and multi-threaded real-time event scheduling algorithm in RT-DEVS

simulation. Their real-time scheduling component could synchronize the simulation process with the system clock time, verify the feasibility of the real-time event scheduling strategy and scheduling algorithms. In addition to that, the authors of (Cho and Kim 2001) proposed a scheduling policy and a feasibility analysis method for RT-DEVS, and they conclude that RT-DEVS formalism provides a uniform and flexible description of real-time systems.

The formal methods for real-time systems development have advanced. However, they are sometimes difficult to apply in practical applications. The scalability is compromised as the complexity of the system scales up. Hence, the authors of (Moallemi and Wainer 2020) introduce a new theoretical framework called I-DEVS (Imprecise Discrete Event Systems Specification). This framework aimed to guarantee responses to inputs within specified time constraints under transient overloading conditions. The main computations in the RT-DEVS runtime engine occur in state transition functions as well as in scheduling and message transfers. So, if a system receives many inputs, due to the overloading, the outputs can be delayed exceeding their deadlines. Hence, the authors of (Moallemi and Wainer 2020) also introduced an overload management policy. The policy provides an early reaction mechanism to transient overrun situations, saving critical outputs from lateness, preventing catastrophic results in the system by prioritizing the model behavior.

The major aim of designing systems using RT-DEVS is that the system can be simulated and tested before it can be implemented, and this is carried out in the same work-frame. However, in the design of more complex systems, the simulation models are often discarded as unusable by the implementation stage. In some cases, the implementation is not derived from the simulation model but added independently during the development of

a system. This discontinuity is a common deficiency of this design method. Hence, the authors of (Hu and Zeigler 2005) the concept of “model continuity”, which to the ability to transition as much as possible a model specification through the stages of a development process. In (Hu and Zeigler 2005) a step-wise simulation-based test process was developed so that the control modes of an embedded system can be incrementally tested. This process included four steps namely central simulation, distributed simulation, HILS, and real system test.

For some classes of systems, it is advantageous to develop real-time models instead of stepwise or logical-time models. Toward this goal, the action-level real-time (ALRT) discrete-event system specification (DEVS) modeling and simulation approach was proposed in (Sarjoughian and Gholami 2015) by the authors. In the tests conducted by the authors, it was observed that Parallel DEVS (PDEVS), RT-DEVS, and real-time state charts are not well-suited in certain models. ALRT is an extension of RT-DEVS in those models which refers to as hard real-time simulation. ALRT-DEVS approach targeted for network processors with high precision clocks, specialized hardware, and real-time operating systems.

(Boukerche, Shadid and Zhang 2007) implemented a formal approach to design Real-Time Run-Time Infrastructure (RT-RTI) using RT-DEVS. This helped in the efficient design of RT-RTI which is a crucial part of the development of High-Level Architecture (HLA) based simulation systems. By using the RT-DEVS model-based formal approach to design HLA systems, model continuity helped in solving the complex dynamic system design problems and issues quickly and effectively.

The existing toolchain used to implement Discrete-Event Modeling of Embedded Systems, ECD++, had some shortcomings that were addressed and improved in Real-Time Cadmium (RT-Cadmium). RT-Cadmium is a Real-Time (RT) DEVS kernel developed on top of the Cadmium DEVS Simulator: (Earle, Bjornson and Ruiz-Martin, et al. 2020). RT-Cadmium allows users to switch between simulating and deploying their models with ease. RT-Cadmium is portable between target platforms, and it already supports MBed Enabled ARM microprocessors and Linux-based systems: (Belloli, et al. 2019).

Cadmium is a C++17 header-only simulator that also has an implementation of an RT-Kernel. Cadmium and RT-Cadmium use the same structure for implementation. The implementation includes two important directories – *atomics* and *top_model*. The atomics directory contains the definition of the DEVS atomic models and the *top_model* directory consists of the top model implementation of the model along with the inputs and outputs directories for the simulation. The *atomics* folder consists of all the DEVS atomic models defined in the previous chapter and it uses the following structure. The atomic models are defined as *.hpp* files. The .hpp file for an atomic model definition in Cadmium contains a C++ structure that defines the input and the output ports of the model. The atomic model defines a class of its name which consists of these functions: Internal transition function, external transition function, confluence transition function, output function, and time advance function. The class also consists of a structure definition that holds the states of the model.

Once all the atomic models are defined, they are then instantiated, and implementation of the coupled model is done in the main.cpp file. The definition of the coupled model consists of defining the external and internal couplings along with defining the ports for the coupled

model. The logger is then defined using the name of the output. The logger definition includes defining the output file for all the simulation logs so that it is easier to understand the errors. The time class is instantiated. And a call for the simulation runner is then made with the time class and the logger as the parameter. The main function contains the logger definition, atomic model instantiations, an implementation of the coupled model, and call for the runner object that is responsible to run the simulation.

In (Zeigler and Kim 1993), the authors discuss the application of RT-DEVS in modeling and simulating an autonomous oxygen extraction plant. They use the Extended DEVS scheme in implementing a prototype using the real-time simulation and control facilities. RT-DEVS framework provides a basis for the modeling of discrete event systems on which logical analysis, performance evaluation, and implementation can be performed based on only one framework. The DEVS specification language or DEVS formalism is used in the separate phases of system design and implementation. The authors of (Song and Kim 2005) used a set of tools that include DEVSim++ and RT-DEVSim++ for performance analysis and real-time simulation, respectively. They also developed a case study of the railroad crossing system which shows the implementation of RT-DEVS formalism in designing a system.

The authors of (Ruiz-Martin, et al. 2019) implemented a design of control of a quadcopter based on DEMES. They implemented a Stabilizer, PID Controller and Command, and an IMU. They found that the modeling-based approach for designing control for quadcopter makes it flexible to easily port the model to different embedded target platforms.

PowerDEVS, in (Bergero and Kofman 2010), allows atomic DEVS models to be defined in C++ language that can then be coupled graphically in hierarchical block diagrams to

create more complex systems. PowerDEVS automatically translates the graphically coupled models into a C++ code which executes the simulation. Combined with its continuous system simulation library, PowerDEVS is also an efficient tool for real-time simulation of physical systems. The authors of (Bergero and Kofman 2010) presented an example of real-time control of a small DC motor using PowerDEVS. The pulse signal was sent to the motor by using the mouse scroll wheel and hence actuating the motor using PowerDEVS.

DEVS-over-ROS relies on the DEVS framework for robust modeling and real-time simulation of hybrid controllers. The ROS middleware is used for the flexible abstraction of software/hardware interfaces for sensors and actuators (Marcosig, Giribet and Castro 2018). The authors established a connection between DEVS and ROS using the PowerDEVS simulation toolkit discussed previously. The ROS messages played a role in external events in the DEVS framework which consisted of value and the timestamp arriving at an atomic model.

In (Earle, Bjornson and Boi-Ukeme, et al. 2019), the authors presented a building controller built using the E-CD Boost library, which uses a DEVS model to control the lights and emergency systems of a building based on room occupancy and other inputs. These model-based control systems were developed to optimize energy consumption in buildings.

The applications of RT-DEVS show that it has a potential and a safety-critical factor. The study shows that RT-DEVS is a reliable option in applications like building control and management where fault tolerance is prioritized

2.4 Building Information Modeling (BIM)

BIM is a loose term introduced in the early 2000s. BIM incorporates a wide range of activities in object-oriented Computer-Aided Design (CAD). It facilitates the representation of architectural elements in terms of their geometric and non-geometric (functional) properties and interactions in 3-D (Ghaffarianhoseini, et al. 2017). BIM is suitable for larger and more complex designs. It is applied in the maintenance of commercial, residential, educational, and many other building types. In a building life cycle, BIM in a broader sense helps in solving the functional issues, informational issues, organizational issues, and in a narrower sense, it helps in solving technical issues: (Volk, Stengel and Schultmann 2014).

The authors of (Bryde, Broquetas and Volm 2013) provided a survey of the benefit of BIM in the construction and maintenance of buildings. The data obtained from the case studies they conducted suggest that BIM is an effective tool in improving certain aspects of the delivery of construction and maintenance projects. The cost was one of the most positively influenced by the implementation of BIM followed by time consumption, communication, coordination improvement, and quality. They found that the negative aspects of BIM implementation are low and most of them were hardware or software issues. This says that there is huge room for improvement in the hardware and software of the systems used.

One of the effective ways to achieve building energy efficiency is to use the technology of Building Energy Modelling (BEM). BEM aims at evaluating alternative designs; comparing and selecting systems and subsystems; allocating annual energy budgets; achieving compliance with energy standards; and economic optimization, during the building design process. However, BEM is not integrated into digital planning and process

and hence energy-efficient implementation is not well done in the initial stages (Gao, Koch and Wu 2019). Hence, BIM can be used to feed the BEM with the information of architectural design, mechanical loads, material properties, or HVAC systems. This drastically improves the implementation of design and development of energy-efficient construction and manages the energy consumption in real-time. In addition to that, the authors of (Chong, Lee and Wang 2017) provide extensive research on BIM development for sustainability. According to them, the planning, design, construction, energy consumption, operation, and maintenance categories are primarily addressed in BIM standards and guidelines, although research papers have shown a strong and growing interest in the design and energy consumption categories.

In (Gourlis and Kovacic 2017), the researchers applied BIM software and modeling in the process for developing two case studies and provided a comparative study on using BIM software in the application of BEM.

In the application of BIM in the field of building maintenance, a knowledge base is required to obtain the status of physical aspects in the building. Hence, an implementation of a reliable embedded system with the deployment of sensors is required for monitoring. This is how BIMs can be used to link embedded systems to architecture. The authors of (Motawa and Almarshad 2013) developed an integrated system to capture information and knowledge of building maintenance operations when/after maintenance is conducted to understand how a building is deteriorating and to support preventive/corrective maintenance decisions. Also, in (Lee, Cha and Park 2016), the authors constructed a data acquisition system of building energy consumption and a web browser using BIM to monitor the building's energy consumption. They performed a case study on an elementary

school in Sejong where they collected data using sensors for the status of energy consumption. The researchers then provided a 3D web browser of the test-bed. This thesis uses this state-of-the-art review as a base for the implementation of cyber-physical system interface in DEVS, since there is not much research done in combining RT-DEVS, sensor fusion, and BIM, and portrays an application in BIM and building maintenance.

To summarize, CPSs are regarded as safety-critical and dependable systems. The dependability of the system relies on the physical and cyber interactions in the system. We discussed how sensor reliability is an important aspect of the dependability of CPS along with some discussion on types of sensor fusion algorithms. We provided some background on HILS and how HILS is an effective way for the development of embedded systems. In addition to that, we explained what DEVS and RT-DEVS M&S techniques are and how HILS and DEVS are co-related to each other. We also discussed some of the critical applications of DEVS and RT-DEVS. Due to the discrete-time nature of the CPS interface, we believe that DEVS and RT-DEVS M&S technique can be a good fit for its formalization. The formalization incorporating the sensor fusion has been explained in Chapter 3. Furthermore, from an application point of view, we provided a brief overlook of the BIM and their applications with respect to real-time systems. We plan to apply the proposed formalized interface framework in the field of building management. The implementation, simulation results, and the building application case study, incorporating the sensor fusion, have been discussed in detail in Chapter 4 of this thesis.

Chapter 3: Interface Framework

As discussed in the previous chapters, physical and software components in the cyber-physical system are deeply intertwined and can interact with each other in ways that change with context. These interactions between the cyber and physical worlds in a cyber-physical system are done through an interface. There are two types of these interactions, physical/stigmergic and message-based.

Physical interaction is a connection and coordination through the environment between the same or different agents. These interactions in a cyber-physical system are managed by electronic or mechanical hardware in which there is a conversion of energies. The energies present in the environment in the form of physical quantities that can be measured are known as environmental variables. These environmental variables include physical quantities like force, torque, light, motion, position, heat, temperature, pressure, etc. Transducers, normally electronic or mechanical hardware, are the devices that conduct the conversion of the environmental variables to and from electrical signals.

Transducers are categorized as *sensors* or *actuators*. A sensor receives a stimulus from the environment, and it responds in the form of electrical signals according to the received stimulus. For instance, microelectromechanical system (MEMS) gyroscopes are small sensors that measure angular velocity. When it is rotated, a small resonating mass is shifted as the angular velocity changes. This movement is converted into very low-current electrical signals that can be amplified and read. In the MEMS gyroscopes, there is a conversion of mechanical wave energy to electrical signals and this conversion takes place in the physical interface.

An actuator is a transducer responsible for the moving or controlling mechanisms, typically the conversion of the electrical signals to another form of physical measure. Actuators use energy from a source upon receipt of an electrical signal to bring about change in the current physical state. For instance, a simple DC electric motor is an electrical machine that converts electrical energy to mechanical motion. It consists of a stationary set of magnets in the stator, and a coil of wire to generate an electromagnetic field around it when there is electric current passing through the wire. When the DC electric current is passed through the coil in an on and off sequence, a rotating magnetic field is created that interacts with the differing fields of the stationary magnets in the stator, which causes it to rotate. Therefore, in the DC electric motor, there is a conversion of electric energy to a magnetic and mechanical rotational motion and this conversion takes place in the physical interface. All the sensors and the actuators are a part of the physical world in the CPS, and they conduct the conversion of energies to and from electrical signals in the physical interface. These electrical signals can be seen in the cyber world as “messages”, and these messages are exchanged in the cyber world between the components, known as message-based interactions.

Message-based interactions in the CPS are crucial since they carry information from the physical world to the cyber world. The message-based interactions in the CPS are carried out through the CPS interface. These messages passed through the CPS interface normally follow an organization and a set of rules. If we formalize/standardize the behavior of the CPS interface in the system, can make the behavior of the cyber-physical system more predictable, manageable, and consistent. Therefore, this thesis proposes a formalization of

these interactions in the CPS interface. We call it the DEVS CPS Interface Framework (DCIF).

As discussed earlier, the DEVS formalism, a modular and hierarchical method for modeling and analyzing discrete event systems is well fitted to define the DCIF. In this thesis, we use DEVS to define the DCIF, but the formalization can be adapted to other modeling and simulation methods easily.

3.1 Design Considerations

In order to proceed with the formalization of the CPS interface, there are some important design considerations that have been made –

1) The sensors and the actuators being used in a cyber-physical system can be connected to the microcontroller or a central computing unit through different protocols such as SPI, I2C, UART, or something as simple as Digital/Analog I/O. This formalization does not consider any communication protocol as its part, for the sensors and actuator connections. The formalization of the interface is completely hardware independent.
2) Some sensors may take time for an initialization process to be able to use them. These delays are supposed to be small and can be discarded. Hence, they have not been taken into consideration in the design of the interface framework.
3) The proposed interface framework is completely time-synchronous. Managing asynchronous inputs to this model is not defined and hence any interrupt-based inputs are not taken into consideration as of now. Interrupt-based inputs can be

resolved by adding some additional handlers, but this is beyond the scope of this thesis.

3.2 High-Level Design

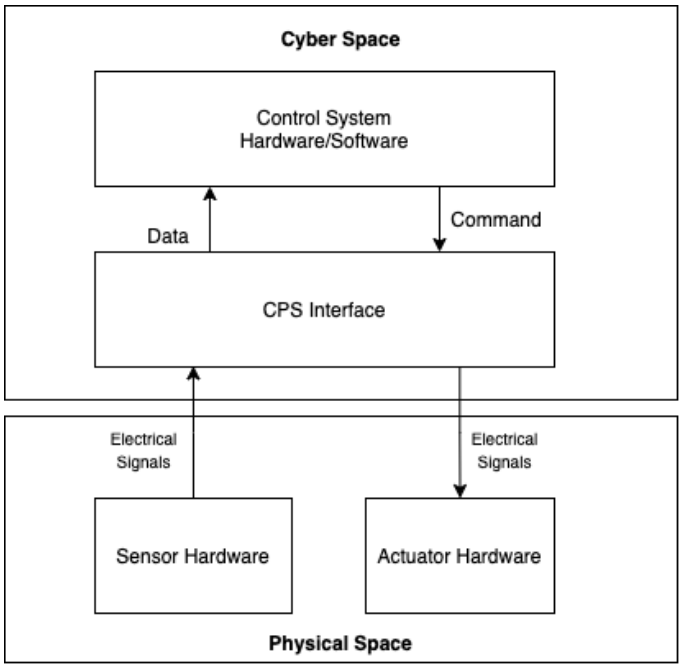

Figure 1. High-Level Design of Cyber-Physical System

In a CPS, we have sensors and actuators connected to a control system as discussed earlier. Figure 1 shows the high-level design of a cyber-physical system with both worlds connected through an interface. As shown in the figure, the physical space and the cyber space are connected to each other. There is an exchange of information that takes place between the two spaces of the CPS. The sensor and the actuator hardware are a part of the physical space. And the control system and the CPS interface are part of cyber space. The working principle of the CPS is that the control system receives the data from the sensors at a specific time interval. Once the data is received by the control system, it decides on the actuation and sends an actuation command to the actuator hardware. The transmission

and the reception of the data and the commands are done through the CPS interface framework. The job of an interface is to fetch the data in the form of electrical signals from the sensor hardware at a specific time stamp, process it to make it reliable, and give it to the control system to decide on the actuation.

In parallel to that, the interface communicates with the control system on the actuation command and once it is received, the interface is responsible for passing it to a specific actuator to operate it according to the received command. There are complicated tasks involved in the interface framework that it needs to conduct before sending out any data to/from the control system.

The CPS interface is made up of components or layers that are responsible for specific unique tasks performed on the data. Each layer has several unique functions to perform according to its functional property. There are three functional layers in a CPS interface – The cyber-physical layer, the Informational layer, and the Service layer. In Figure 2, the three layers of the CPS interface are shown. The layers are made up of DEVS atomic and coupled models. As shown in the figure, they are connected to each other, and they perform message-based interactions to communicate. It is two-way communication between the layers through the same or different DEVS models.

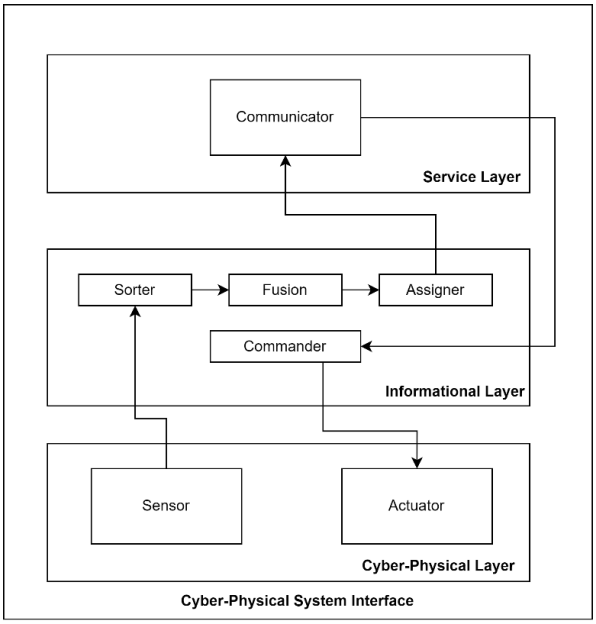

Figure 2. Cyber-Physical System Interface

The message-based interactions involve data correction, manipulation, and processing which is conducted in the DEVS models in each layer. These messages exchanged between the models have a unique structure for each model. We will also propose a concrete structure of messages used to pass information between the layers. These messages play an important part in the CPS interface framework as they pass information throughout the interface, therefore making it important to study them.

The cyber-physical layer, informational layer, and service layer form the DEVS Cyber-physical System Interface Framework (DCIF). Figure 3 shows the top model of DCIF which is an extended version of Figure 2.

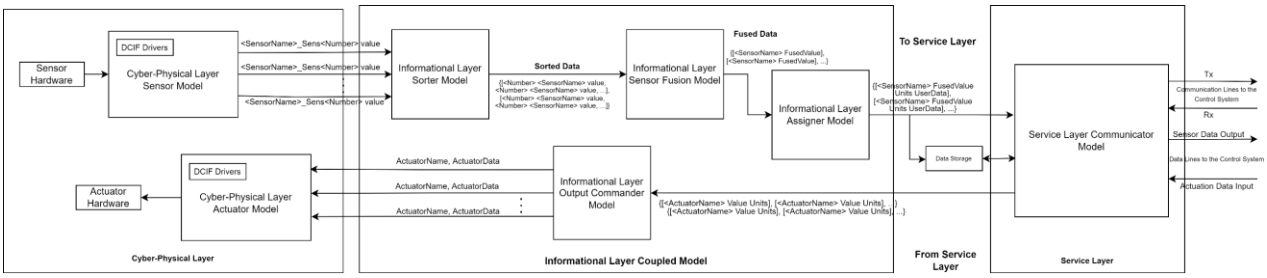

Figure 3. DEVS Cyber-Physical System Interface Framework (DCIF) Top Model

In the figure above, we can see that the Cyber-Physical layer, Informational layer, and the Service layer are connected to each other as discussed earlier. The Cyber-physical layer has two DEVS atomic models: Sensor Model and Actuator Model. The informational layer in the DCIF is a coupled model with 4 DEVS atomic models: Sorter, Sensor Fusion, Assigner, and Output Commander. The service layer consists of one DEVS atomic model: Communicator. The DCIF works for both cases - a single input/output or multiple inputs/outputs. However, it is considered that the sensor and actuator models have "multiple instances" with single output and input respectively, and not "one instance" with multiple outputs and inputs. In the case of a single sensor/actuator, the model will have only one instance of the cyber-physical layer sensor/actuator model. For multiple sensors and actuators, there will be multiple instances of the cyber-physical layer sensor and actuator models. A detailed explanation of the components of DCIF is provided in the following section.

3.3 Components of the DEVS CPS Interface Framework

As discussed previously, the three layers of DCIF – Cyber-Physical, Informational, and Service are its components. The layers are made up of atomic and coupled DEVS models. In this section, we will discuss all the DEVS models involved in the formation of DCIF.

3.3.1 Cyber-Physical Layer

The cyber-physical layer component of the interface establishes a connection between the hardware and the cyber world. At the cyber-physical layer, the information is represented by bit patterns and is exchanged between this layer and the informational layer. The bit pattern is a combination of binary digits 1s and 0s arranged in a sequence. We can obtain a bit pattern representing some data by using the digital electrical signals and finding their ON or OFF state at a particular interval in time. This conversion of the electrical signals to bit patterns and further to a message that can be transferred between the cyber-physical layer and the informational layer is carried out in the cyber-physical layer. The detailed working of the cyber-physical layer is explained later.

The cyber-physical layer gets the data from the sensors at a time interval in the form of electrical signals and converts them into a bit pattern. The sensors can be connected to the processing hardware using any communication protocol – for example SPI, I2C, or UART - but since the interface framework is independent of the communication method or the hardware used, a bit pattern of the data is obtained regardless in all cases. Once the bit pattern is received by the cyber-physical layer, it sends it to the informational layer for processing. In parallel to that, it receives actuation commands from the control system through service and informational layers. It converts the data into a bit pattern and sends it out to the required actuator to perform an action.

Some important properties at the cyber-physical layer include bit rates, energy levels, transmission medium, if they are signals or bit pattern (prearranged representation of the data according to the communication protocol with the sensors/actuators), etc. The cyber-

physical layer has all these “lower-level details” defined for the CPS to run on specific hardware. For instance, if a sensor is connected to a microcontroller through the SPI communication protocol, it is important to specify the communication pins used, initialize the SPI device to be a controller or peripheral, set a correct data rate, define if the bit pattern, in the form of data, is obtained by a rising or falling clock edge, etc. Once all the lower-level details are defined, the hardware becomes ready to use in the CPS. These details are only limited to and are the responsibility of the cyber-physical layer. Once the bit pattern is obtained, all the lower-level details are ignored and only the data is manipulated in the further layers.

We formally define the cyber-physical layer as a composite of two DEVS models: the Cyber-Physical Layer Sensor Model and Cyber-Physical Layer Actuator Model and the device drivers that are responsible to communicate with the hardware

Figure 4 shows the structure of the cyber-physical layer sensor model. The sensor hardware is connected to the DCIF through the cyber-physical layer sensor model. The sensor model deals with managing the connection to the sensor hardware and fetching the data using the DCIF drivers. In the case of multiple sensors connected in the system, we defined multiple instances of this model connected to those sensors' hardware.

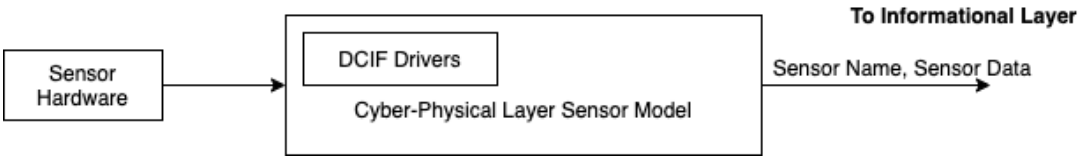

Figure 4. Cyber-Physical Layer Sensor Model

On the other hand, the actuator hardware is connected to the DCIF using the cyber-physical layer actuator model. Figure 5 shows the cyber-physical layer actuator model connected to

the hardware. The actuator model receives actuation data from the informational layer, and it deals with handling the actuation of the hardware using the DCIF drivers. The working of the sensor and the actuator models is discussed in detail later, but primarily it is essential to study the DCIF drivers to make it easier to understand the working of the models.

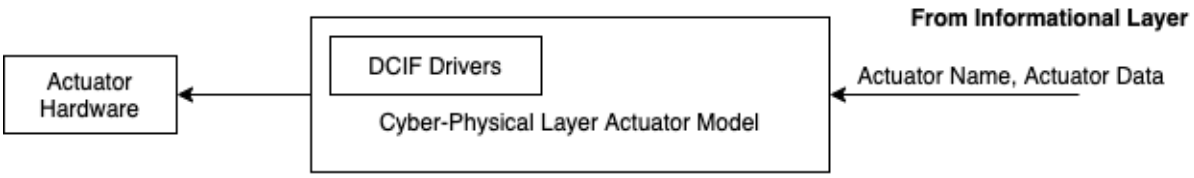

Figure 5. Cyber-Physical Layer Actuator Model

The DCIF drivers are programs that operate, and control sensors or actuators connected to the processing hardware i.e., a microcontroller. The drivers provide a software interface to the hardware devices to make the models in the cyber-physical layer access hardware functions without needing to know specific details about the hardware being used. In DCIF, the drivers are a unified wrapped single library of multiple hardware manufacturers' libraries, developed for the DEVS simulator. The drivers are divided into two categories – sensor or actuator drivers. They read the data from the sensors and write the data to the actuators respectively using the controller hardware. Figure 6 shows the basic structure of the drivers in the DCIF. As shown in the figure, DCIF drivers are a parent class and there are two subclasses of the DCIF drivers. The sensor drivers are further classified into the type of sensors. There are different manufacturers designing the same type of sensor that provides the same functionality with different or same working principles and with added or reduced precision. All the manufacturers' sensor programs, to operate and control the sensors, are defined here. The actuator drivers are classified into analog or digital since

most of the actuators are either analog or digital devices. The programs for the manufacturer-specific actuators are defined in the actuator drivers.

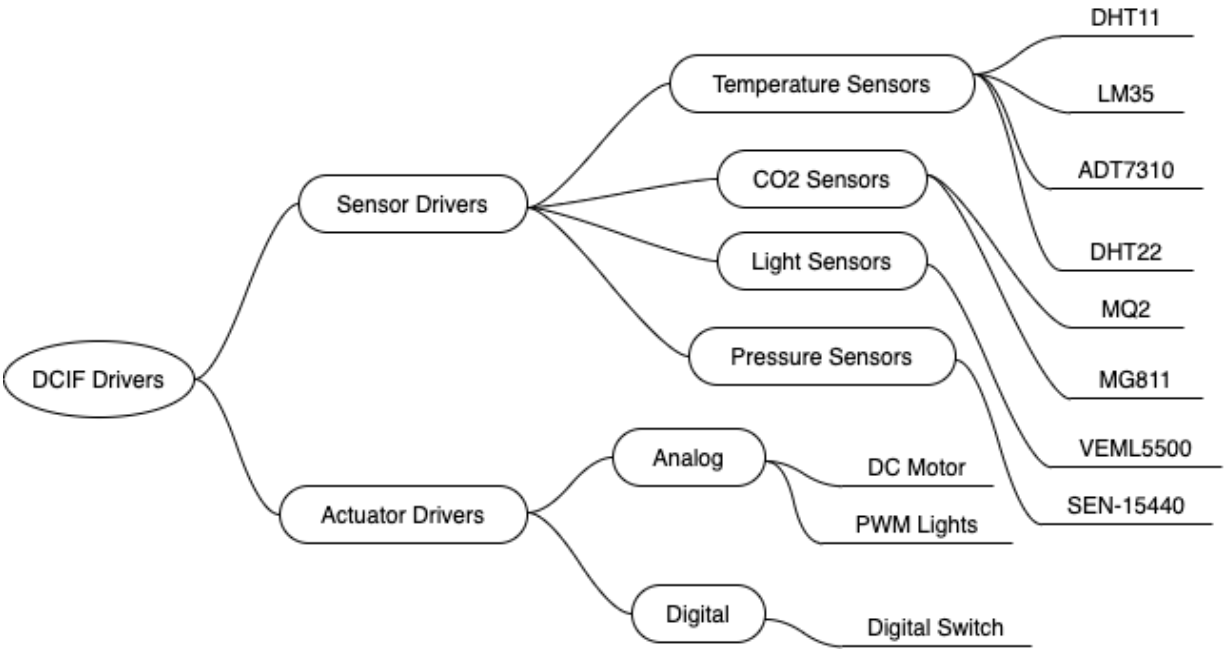

Figure 6. DCIF Drivers Structure

Reducing all the manufacturers' libraries to a single library (DCIF drivers), enables us to switch sensor models easily and effectively with no impact on the rest of the system. Also, these drivers return a value with specific standardized SI units for each sensor family which makes it comparable with any other similar sensors. These specific standardized SI units are retained throughout the DCIF and are later used in the informational layer to make the data presentable. Table 1 shows the default units used in the DCIF for the sensor readings. The DCIF drivers help reduce the problems of sensor availability and help reuse the code. In addition to that, the DCIF drivers also provide a skeleton for introducing the management of hardware-related issues such as buffering, exception handling, overflow of data, etc.

Sensor type	Units
Temperature	Degree Centigrade (Celsius)
Humidity	Relative percentage
Light	Luminosity (lux)
CO2	Parts per million (PPM)
Gyroscope	Rad/s
Distance	Centimeters
Voltage	Volts
Current	milliamps
Color	RGB component values

Table 1. Standardized Units in DCIF

The DCIF drivers are called in the cyber-physical layer sensor and the actuator models. The job of the cyber-physical layer sensor model is to fetch the bit pattern using the DCIF drivers from the sensor hardware and to send it to the informational layer in the form of a message. In parallel to that, when there is an actuation command received from the informational layer, the cyber-physical layer actuator model receives the messages, converts the information to a bit pattern or an analog signal, and sends it to the actuator hardware for the actuation using the DCIF drivers. For instance, we have one sensor and one actuator connected to the control system through the DCIF in a cyber-physical system. The sensor gives out the data to the cyber-physical layer sensor model. The sensor model converts the data to a message structure and sends it out to the informational layer. Concurrently, once the actuator model receives an actuation command in the form of a message from the informational layer, the actuator model converts the data to a bit pattern

and sends it out to the actuator hardware. If we have multiple sensors and actuators connected to the control system through DCIF, each sensor and the actuator has their own cyber-physical layer models but the output message from the model is different and unique to avoid confusion. The output and input messages of the cyber-physical layer models are discussed later.

The message passing between the models in the cyber-physical layer and the informational layer follows a structure, we call it the cyber-physical layer messages. The message structure is used to make it easier for the informational layer as well as the cyber-physical layer to understand the type of data, timestamp, the name of the sensor, and the value. It consists of two parts, a string with the name and a float with a value, as shown below –

From sensor model to the informational layer, we have

TIMESTAMP <SensorName>_Sens<Number> value

For example, if we are using a temperature sensor that collects a value of 23 degrees Celsius at a specific time stamp (say 00:00:02), then the structure of the message that will be passed from the sensor model to the informational layer will be

00:00:02 Temp_Sens1 23

From the informational layer to the actuator model, we have

TIMESTAMP <ActuatorName><Number> value

For instance, if we use a motor as an actuator and a message from the informational layer is obtained by the actuator model at a time stamp 00:00:05 to actuate it at the analog value of 240 in the range of 0-255, the message structure will be

00:00:05 Motor1 240

The cyber-physical layer consists of two components as mentioned before i.e., Sensor Model and Actuator Model. The Sensor model component is formally defined using DEVS as the following atomic model:

```
SM = <S, X, Y, δint, δext, δcon, λ, ta>
S = {}
X = {inputs from hardware is accessed through the DCIF drivers}
Y = {sensor_message}
δint (s) = {Call the DCIF drivers and obtain data, store in variable data.
Apply signal conditioning if needed.}
δext(s,e,x) = {N/A}
δcon = δint; δext;
λ () = {sensor_message = data}
ta () = pollingRate //i.e., data rate
```

According to the formal definition above, there are theoretically no inputs to the DEVS model but the input to the DEVS model is nothing but the sensor hardware. The hardware is accessed using the DCIF drivers. Since there are no inputs, there will not be any external transition and hence the external transition function is not applicable in this case. In the internal transition function, the model obtains the data from the hardware using the DCIF drivers and store it in a temporary variable named ***data***. Conceptually, the state of the sensors/actuators read by the DCIF drivers (Figure 3 and Figure 4) is now a part of the DEVS model's state. Here, any signal conditioning algorithms can be applied to make the sensor values reliable in case it is not. Once the data is obtained, it is sent to the output port ***sensor_message*** and the model advances with the time ***pollingRate***.

The actuator model component is formally defined using DEVS as the following atomic model:

```
AM = <S, X, Y, δint, δext, δcon, λ, ta>
S = {sendingDatatoActuator ∈ {true, false}}
```

```
X = {actuator_message}
Y = {outputs to the hardware is accessed through the DCIF drivers}
δint (s) = {if sendingDatatoActuator = true, then sendingDatatoActuator =
false}
δext (s,e,x) = {if sendingDatatoActuator = false, then
                actuator_name = actuator_message::string
                actuator_value = actuator_message::float
                sendingDatatoActuator = true}
δcon = δint; δext;
λ (sendingDatatoActuator=true) = {Call the DCIF drivers with the actuator
values
        as the parameters. Apply signal conditioning if applicable.}
ta (sendingDatatoActuator = true) = actuationTime
ta (sendingDatatoActuator = false) = INFINITY
```

Similar to the sensor model, the actuator model theoretically does not have outputs but the output to this model is the actuator hardware connected to the model. The output to the hardware is accessed through the DCIF drivers as mentioned earlier. In the output function of this model, the DCIF drivers are called with the actuator name and value (i.e., from the informational layer as the actuator message) as the parameters which actuate the hardware according to the values passed to this layer from the informational layer.

Effectively, the DCIF drivers access state variables in the external devices, which, in fact, function as state variables in the corresponding DEVS models.

3.3.2 Informational Layer

In DCIF, the informational layer is a coupled model that is responsible to make the data more reliable and fail-safe. Signal conditioning is applied to the sensor and the actuator models in the cyber-physical layer, but this is hardware-specific. This means that it does not deal with the information bound to the signals but deals with conditioning the

errors/jitters in the signals. There is a need to add an additional layer of reliability that deals with only the informational data obtained from the sensors or being passed to the actuator. Therefore, the implementation of the informational layer is focused more on making the data secure and reliable. In addition to that, it is also focused on optimizing and storing the data in such a way that it is in a presentable format.

Reliability refers to the consistency of the data throughout the cyber-physical system. Reliability of the data is important during data collection and data manipulation. Cyber-physical systems are complex, and they have safety-critical applications which make it crucial to keep the data fault-free and reliable. The informational layer in the DCIF deals with adding a security layer to the cyber-physical system by providing features such as sensor fusion, data sorting, data interpretation, and data assignment and storage for representation and reuse. A detailed explanation of each task performed by the informational layer is provided in the following sections.

The cyber-physical layer gets the data from the hardware as discussed earlier and sends it to the informational layer. The informational layer in the DCIF receives the data from the cyber-physical layer and sends it out to the service layer after processing it. It also receives the command from the service layer and sends it out to the cyber-physical layer after sorting the data according to their actuator names and types. The data processing includes the tasks mentioned earlier – data sorting according to the sensor name and type, sensor fusion to fix irregular data, data interpretation, and data assignment to make it presentable/understandable. These tasks are performed by different models in the informational layer. Every model in the informational layer of DCIF is responsible for its unique tasks performed. The models in the informational layers are – Sorter Model, Sensor

Fusion Model, Assigner Model, and Output Commander Model. These models are all hardware-independent, viz they do not deal with any lower-level details discussed in the cyber-physical layer. All the "lower-level details" in the cyber-physical system are managed by the cyber-physical layer therefore, the informational layer takes care of the "higher-level details".

Once the data is obtained by the informational layer from the cyber-physical layer, the DCIF does not have to deal with the requirements of the hardware. The informational layer focuses only on the higher-level details, i.e., the information packaged in the received data. This does not include anything related to the hardware (i.e, the bit pattern, data rate, or the communication protocol used). For instance, if the cyber-physical layer obtains data from the sensor connected to a microcontroller through the SPI communication protocol (using DCIF drivers and drivers details such as the communication pins used, SPI initialization, data rate, bit pattern, etc.), it is sent out as a message using the message structure mentioned earlier using multiple models explained below.

The informational layer is a coupled model that consists of 4 atomic models. The atomic models defined in the DCIF informational layer have unique functionality that deals with data manipulation and data fusion. The informational layer is a coupled model, and it contains all the atomic models presented in Figure 7.

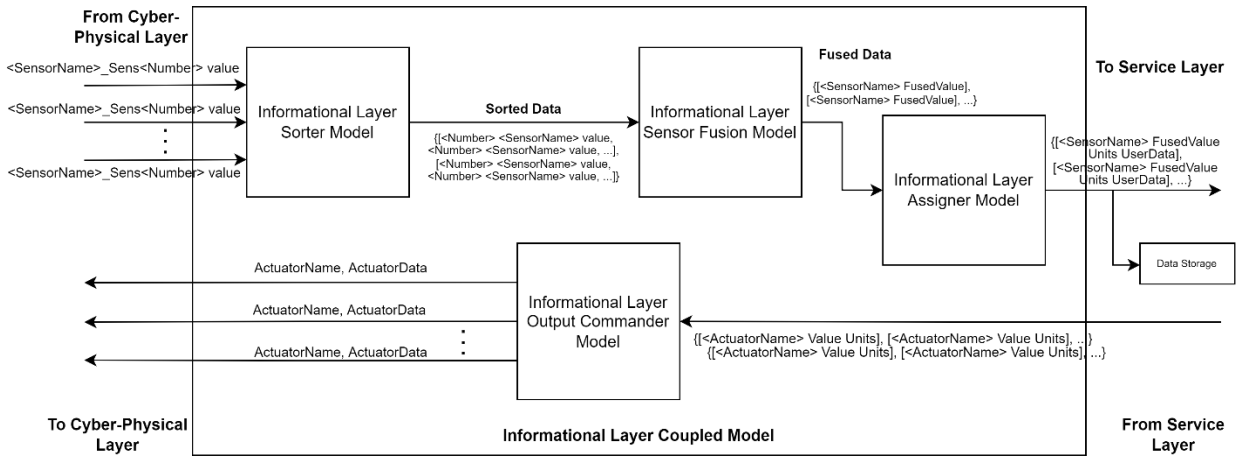

Figure 7. Informational Layer Coupled Model

The coupled model contains the four atomic models. These atomic models are connected to each other and there is a message-passing system between them. The data is processed at every stage and later it is stored in the data storage and also provided to the service layer. The number of connections from the cyber-physical layer depends on the number of sensors and the actuators used in the implemented cyber-physical system. As a coupled model, there are inputs from the cyber-physical layer sensor model and output to the service layer communicator model and the cyber-physical layer actuator model.

The inputs to the informational layer coupled model from the cyber-physical layer sensor model is given as follows:

TIMESTAMP <SensorName>_Sens<Number> value

Where TIMESTAMP is a time value at any point in time, <SensorName> is the name of a sensor (e.g., Temperature, CO2, etc.), <Number> is the sensor number that can be given to the sensor (e.g., 1, 2, 3, ...) and value is the sensor reading at that timestamp. There can be multiple instances of the sensor model in the case of multiple sensors.

The output to the service layer and the data storage from the information layer coupled model is given as follows:

{[OUT_TIMESTAMP <SensorName> FusedValue Units UserData], …}

Where OUT_TIMESTAMP is a time value at any point in time after the input arrives at the model, <SensorName> is the name of a sensor as mentioned earlier, *FusedValue* is the sensor reading obtained after performing a sensor fusion, *Units* is the standardized units for the sensor readings and the *UserData* is any additional information specified by the user to be included in the message (e.g., faulty sensors).

The inputs to the informational layer coupled model from the service layer communicator model are given as follows:

{[TIMESTAMP <ActuatorName> Value Units], …}

Where TIMESTAMP is the time value at any point in time, <ActuatorName> is the name of the actuator (e.g., Motor), *Value* is the standardized value of the actuator and the *Units* is the standardized units specified for the actuator (e.g., RPM).

The output to the cyber-physical layer from the information layer coupled model is given as follows:

OUT_TIMESTAMP ActuatorName ActuatorData

Where OUT_TIMESTAMP is the time value at any point in time after the input from the service layer arrives, *ActuatorName* is the name of the actuator as mentioned previously and *ActuatorData* is the lower level mapped/converted data for the specific microcontroller/microprocessor.

In the explanation given above, we discuss the inputs and outputs of the informational layer coupled model as a whole. It also explains the overall working of the informational layer

coupled model. However, the coupled model consists of multiple atomic models. Each atomic model in the informational layer has been discussed in the following section.

3.3.2.1 Sorter

The sorter model receives data from the cyber-physical layer sensor model and sorts the data if there are multiple sensors. The sorter model is not used if there is just one sensor connected. Otherwise, it is responsible for sorting the data according to their sensor types and providing them to the sensor fusion model for fusing the data in the informational layer. After the multiple sensor data is received by the sorter model at a specific time, the sorter model sorts the data in groups of their sensor types and arranges it into a sequence container. These containers of sensor data are then sent in the form of a message to the sensor fusion model to further apply sensor fusion algorithms for the reliability of the system. The detailed format of the messages has been explained later.

The data sent to the informational layer from the cyber-physical layer includes the sensor name and number along with the data provided by that particular sensor. This data in the form of a message, when received by the sorter model, is organized according to the sensor type name and number.

The major aim of the sorter model is to support the following connected sensor fusion model. At a specific timestamp, there is varied data arriving from multiple sensors to the DCIF. Hence, it is important to sort the data according to their sensor types. If the data is

sorted according to the sensor type and fed to the sensor fusion model, the sensor fusion model will be able to diagnose the faulty sensors efficiently without system confusion.

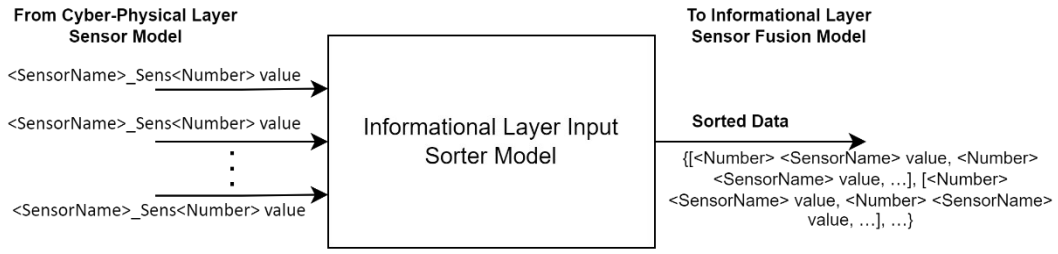

Figure 8. Informational Layer Input Sorter Model Atomic

Figure 8 shows the inputs and the outputs of the sorter model. As shown in the figure, there are multiple inputs and a single output. The input messages to the sorter model arrive from the connected cyber-physical layer sensor atomic model. There are multiple inputs from different instances of the sensor model. The format of the input messages to the sorter model is given below:

TIMESTAMP <SensorName>_Sens<Number> value

Since there are multiple sensors in the cyber-physical system, there will be multiple inputs in the same format, but with different <SensorName> or <Number> or both according to the type.

The sorter model consists of one output port. The output message from the sorter model consists of a sequence container with the sorted data. The sorting of the data is done alphabetically initially to separate the types and later rearranged according to their sensor number. The sorted data is added to a container and sent as a message in the given format below:

{[TIMESTAMP <Number> <SensorName> value, TIMESTAMP <Number> <SensorName> value, …], [TIMESTAMP <Number> <SensorName> value, TIMESTAMP <Number> <SensorName> value, …], …}

For instance, consider a system with four sensors each of type temperature and CO2. The received data in the form of a message from the temperature sensors and the CO2 sensors from the sensor models to the sorter model will be in a format,

TIMESTAMP Temperature_Sens1 value

TIMESTAMP Temperature_Sens2 value

TIMESTAMP Temperature_Sens3 value

TIMESTAMP Temperature_Sens4 value

and

TIMESTAMP CO2_Sens1 value

TIMESTAMP CO2_Sens2 value

TIMESTAMP CO2_Sens3 value

TIMESTAMP CO2_Sens4 value

where TIMESTAMP is the time value at any specific point in time and 'value' is the sensor reading. Once the sorter model receives these messages, they are stored in a one-dimensional sequence container as they arrive. The storage of these messages can be random at this point in time. That is since these messages arrive at the same point in time, the system fails to understand which sensor is named as the first sensor or what is the type of sensor. Hence, the complete container containing these messages is sorted according to their names (alphabetically) and numbers (in ascending order). Here in this case, after the sorting is performed, the container will look like this:

{[OUT_TIMESTAMP 1 CO2 value, OUT_TIMESTAMP 2 CO2 value, OUT_TIMESTAMP 3 CO2 value, OUT_TIMESTAMP 4 CO2 value], [OUT_TIMESTAMP 1 Temperature value, OUT_TIMESTAMP 2 Temperature value, OUT_TIMESTAMP 3 Temperature value, OUT_TIMESTAMP 4 Temperature value]}

where OUT_TIMESTAMP is any time value but after the time when the input is received. As we can see that the container looks much presentable and easier to perform operations on. The CO2 sensors are grouped as the initial index of the container and the temperature sensors are grouped as the following index of the same container due to an alphabetical arrangement. Once the sorting is done, the container is sent to the output port of the model as a message.

The sorter model is formally defined in DEVS as the following atomic model:

```
ISM = <S, X, Y, δint, δext, δcon, λ, ta>
S = {DataSorted ∈ {true, false}, Active ∈{true, false}}
X = {s1, s2, s3, …} According to the number of sensors in the system and
thereby the number of cyber-physical layer sensor model instances
connected.
Y = {Sorter_Out}
δint (s) = {if DataSorted = true, then DataSorted = false and if Active =
true, then Active = false}
δext (s,e,x) = {if DataSorted = false and Active = false, then
              Values_from_sensor[0] = s1,
              Values_from_sensor[1] = s2, …,
              perform sort(Values_from_sensor),
              rearranged_sensor_messages = rearrange(Values_from_sensor),
              send_to_fusion =
              values_to_sorter_message(rearranged_sensor_message),
              DataSorted = true, Active = true}
              Note: Values_from_sensor, rearranged_sensor_messages, and
              send_to_fusion are the variables that are being used in the
```

external transition function. The sort () function performs an alphabetical sorting of the sensor data according to their names, the rearrange() function performs a rearrangement of the sorted data according to the number of sensors, and the values_to_sorter_message() function adds the processed data to a sorter message structure, i.e., a sequence container.

δcon = δint; δext;

λ (Active = true and DataSorted = true) = {Sorter_Out = *send_to_fusion*}

ta (Active = true) = SendingTime

ta (Active = false) = INFINITY

According to the formal definition above, the inputs of the sorter atomic depend on how many sensors are connected in the system. The number of inputs to the sorter model is equal to the number of sensor models connected to it. And the number of sensor models is equal to the number of sensors connected in the cyber-physical system. There is one output port from the sorter model named *Sorter_Out*. At a timestamp, once all the cyber-physical layer sensor models send the data to the sorter, it checks if the model is passive. If the model is passive, the received sensor data is stored in a temporary container variable, *values_from_sensor***.** Initially, sorting is performed and all the data in *values_from_sensor* is sorted according to the sensor name alphabetically. Once it is sorted alphabetically, a rearrangement operation is performed on the same variable to sort the alphabetically sorted data according to their sensor numbers as well. The rearranged data is called *rearranged_sensor_messages*. This rearranged data is added in a special sequence container and sent out to the output port *Sorter_Out.*

3.3.2.2 Sensor Fusion

The implementation of the informational layer in the DCIF focuses on data reliability as discussed earlier, and the sensor fusion model in the information layer maintains the system stability and has been defined to add a reliability layer.

Sensor fusion is a process of combining the sensor data or the data derived from different sources such that the resulting information has less uncertainty than when these sensors are individually used. The term “less uncertainty” in this case means making the information more accurate, more complete, and more dependable. For instance, we can potentially obtain more accurate information of the current indoor weather condition in a particular room by combining multiple data sources of the same or different types such as fusing the readings from multiple temperature sensors. The sensor fusion model provides this sensor fusion framework that can be implemented in the DCIF. The implementation of the sensor fusion model is fairly simple as the model uses software sensor fusion algorithms to fuse the sensor data from various sensors to output a piece of more robust information that can then be used to take decisions by the control system. The fusion model takes the sensor data as an input from the sorter model. Once the data is received, it performs a sensor fusion using a software sensor fusion algorithm and outputs the fused data along with the sensor type.

The sensor fusion model in the informational layer deals with combining i.e., fusing the sensor values to obtain a higher quality output. Along with the fusion, the sensor model can identify the faulty sensors and with that information, the system stability can be maintained. Cyber-physical systems can be used in safety-critical applications, and it is important to have a security layer in the DCIF to maintain that level of reliability. This

model helps dramatically improve the reliability of the system and hence this model is a necessary part of the DCIF.

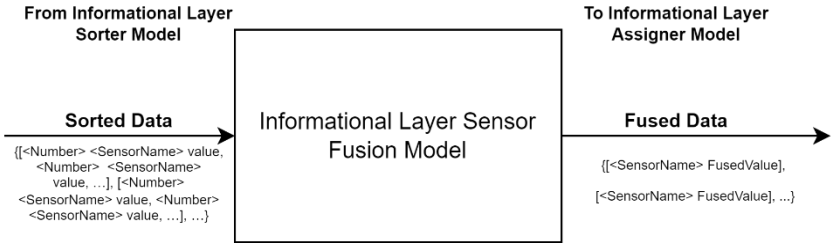

Figure 9. Informational Layer Sensor Fusion Model Atomic

The sensor fusion model has one input and one output port as shown in Figure 9. The output from the sorter model is given as an input to the sensor fusion model. The fusion model receives a sequence container message with all the sensory data from the sorter model. Once received, the sensor fusion model performs sensor fusion on the received data type by type. The algorithm implemented in the DCIF is a multi-sensor data fusion algorithm based on principal component analysis and can be replaced with any other sensor fusion algorithm easily. The implementation of the principal component analysis in DCIF has been explained in the implementation section of this thesis. After performing the sensor fusion, the obtained fused data for all types of sensors is sent in a sequence container to the assigner model in the informational layer.

The format of input message to the sensor fusion model is given below:

{[TIMESTAMP <Number> <SensorName> value, TIMESTAMP <Number> <SensorName> value, …], [TIMESTAMP <Number> <SensorName> value, TIMESTAMP <Number> <SensorName> value, …], …}

The input of the sensor fusion model is connected to the output of the sorter model. And hence the input message structure will be the same as the output of the sorter model.

The format of the output message from the sensor fusion model is given below:

{[TIMESTAMP <SensorName> FusedValue], [TIMESTAMP <SensorName> FusedValue], …}

The sensor fusion model fuses the values of all the sensors in the received container using the sensor fusion algorithm according to their index numbers in the sequence container. This leads to the conversion of multiple sensor values to a single value called as FusedValue for each index in the container. The output message of the sensor fusion model will retain the sensor name along with its fused value.

For instance, consider a system with four sensors each of type temperature and CO2. The output of the sorter model that is given as an input to the sensor fusion model will be,

{[TIMESTAMP 1 CO2 value, TIMESTAMP 2 CO2 value, TIMESTAMP 3 CO2 value, TIMESTAMP 4 CO2 value],
[TIMESTAMP 1 Temperature value, TIMESTAMP 2 Temperature value, TIMESTAMP 3 Temperature value, TIMESTAMP 4 Temperature value]}

where TIMESTAMP is the time value at any specific point in time and ‘value’ is the sensor reading.

Here, the values are properly sorted before they are provided to the sensor fusion model. Therefore, the sensor fusion model will perform a fusion algorithm on the received message and the output message from the fusion model will be in the format,

{[OUT_TIMESTAMP CO2 FusedValue], [OUT_TIMESTAMP Temperature FusedValue]}

Where OUT_TIMESTAMP is the time value after the time at which the input is received and 'FusedValue' is the sensor fusion value for each type of sensor.

In the sensor fusion model, multiple sensor values are converted into a single value for each type of sensor which makes the system less complex keeping the information more dependable.

The sensor fusion model in the informational layer can be defined in DEVS as the following atomic model:

```
SFM = <S, X, Y, δint, δext, δcon, λ, ta>

S = {multiple_types ∈ {true, false}, Active ∈{true, false}}

X = {Sorter_Message_IN}

Y = {Fusion_Message_OUT}

δint (s) = {if Active = true, then Active = false}

δext (s,e,x) = {if Active = false, then

                Receive the messages from sorter

                If (message size > 1) then multiple_types = true.

                If (multiple_types = true) then for each index of the
message, perform sensor fusion and store the value in a sequence container ValueToSend.

                Else perform sensor fusion only once and store the value in
the sequence container ValueToSend.

                Active = true

δcon = δint; δext;

λ (Active = true) = {Fusion_Message_OUT = ValueToSend}

ta (Active = true) = SendingTime

ta (Active = false) = INFINITY
```

According to the formal definition, the model needs to check if there are multiple types of sensors sent by the sorter model in a sequence container. After the check is complete, the state *multiple_types* is set according to the status. If there are multiple types of sensors, the sensor fusion algorithm function is called with the parameter sent to it type by type and the returned value stored in a container named *ValueToSend.* If there is only one type of sensor,

the sensor fusion algorithm is called once with the parameter as the received container and the returned value is stored in a container named *ValueToSend.*

3.3.2.3 Assigner

Now that the reliability check has been conducted in the sensor fusion model, the informational layer assigner model makes the data more organized, and it is also responsible to store the data in a data storage. An IoT application such as a *smart home* will generate a large amount of data in real-time from temperature and humidity sensors alone. This data, in the need of further processing, should be stored using data management techniques. The data storage in the DCIF refers to the storage of the sensor data in real-time. This storage can be offline storage that occurs on the hardware such as an SD card connected to the microcontroller, or it can be online storage in the cloud. The DCIF defines a skeleton for storing the data in the data storage regardless it is online or offline.

The assigner model receives a message with the fused values from the sensor fusion model. As the message arrives, the assigner model encapsulates the information from the received message in a specific format and it gives a scope of addition of more information to the encapsulated message. This additional information includes the units for the sensor data according to the sensor types, in addition to that it can be user-defined details, for instance, the sensor fusion results with the number of faulty sensors or a sensor fusion parameter. This encapsulated message is then sent to the data storage using the predefined storage mechanism. The data storage is allowed to be accessed by the following service layer that is connected to the assigner model in the informational layer. The storage can be accessed

by the service layer whenever needed to be able to provide the required informational data to the control system as requested.

In DCIF, there is a large amount of data transferred and it is necessary to represent and store it in an accessible way. The assigner model fulfills this necessity. After fusing, by removing the informational errors in the sensor data, the assigner model helps in adding more information to the received data. For instance, the data flowing from the cyber-physical layer to the assigner model in the informational layer does not have the units defined for the type of sensors in the data message. Hence, the assigner model adds the units to the sensor data message according to the sensor type. The standard units for the sensor readings have been provided in Table 1 in the cyber-physical layer section. In addition to that, once the additional information is added, the assigner model is responsible to store the data in the data storage for further use by the service layer. The informational layer connects to the service layer through the assigner model. The assigner model keeps on feeding the received current data to the service layer and at the same time, it stores the data to the data storage. This constant feed of the data also makes it easier for the service layer to fetch the current status of the sensor system which we will discuss in the service layer section.

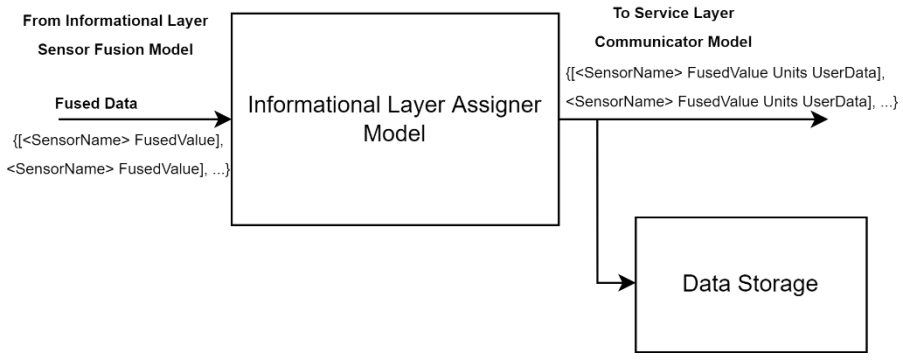

Figure 10. Informational Layer Assigner Model Atomic

As shown in Figure 10, the assigner model has one input and one output that goes to the service layer communicator model and the data storage. The data storage is not a DEVS model, but it is accessed using calls to the specific functions that access the storage and has the authority to read and write the storage. The assigner model receives the sequence container from the sensor fusion model and by default, adds more information to it such as the standardized units for the sensor data. If the user has any more information to add to the sequence, it can be added to the sequence here. Once the sequence is reformatted, it is sent to the output port which takes it to the service layer communicator model. The reformatted sequence is also sent to the data storage by calling the functions used to access the data storage using the sequence as the parameter. The exact format of the inputs and the outputs are provided in the following section.

The format of the input message to the informational layer assigner model is given below:

{[TIMESTAMP <SensorName> FusedValue], [TIMESTAMP <SensorName> FusedValue], …}

The input of the assigner model is connected to the output of the sensor fusion model.

The format of the output message from the assigner model is given below:

{[TIMESTAMP <SensorName> FusedValue Units UserDefinedDetails], [TIMESTAMP <SensorName> FusedValue Units UserDefinedDetails], …}

The assigner model assigns the units to the particular sensor data according to the sensor type. The UserDefinedDetails are the details that can be added to the output message by the model if defined by the user. These details can include anything like the number of faulty sensors identified by the sensor fusion model, or a sensor fusion parameter such as the criterion for the algorithm used. This helps the user to make it easier to understand the

data stored from the data storage through this assigner model and to identify any additional details the system provides as per the definition.

For instance, consider a system with four sensors each of type temperature and CO2. The output of the sensor fusion model after the fusion that is given as an input to the assigner model will be,

{[TIMESTAMP CO2 FusedValue], [TIMESTAMP Temperature FusedValue]}

where TIMESTAMP is the time value at any specific point in time and 'FusedValue' is the sensor reading.

Here, the assigner model receives the sequence container of the fused values. The assigner model adds the units to the sequence container and reformats it. The output message from the assigner model will be in the format,

{[OUT_TIMESTAMP CO2 FusedValue ppm], [OUT_TIMESTAMP Temperature FusedValue °C]}

Where OUT_TIMESTAMP is the time value after the time at which the input is received and 'FusedValue' is the sensor fusion value for each type of sensor received from the fusion model.

The output adds the units to the container according to the sensor type. Hence, the CO2 sensor will have the unit parts per million/ppm and the temperature will have degrees Celsius/°C. This reformatted sequence container is also stored in the data storage.

In the example given above, if the user defines that the output should also include the number of faulty sensors, it can be defined in the assigner model. For example, if there is 1 CO2 sensor that is faulty and 2 temperature faulty sensors, then the output will be in the format,

{[OUT_TIMESTAMP CO2 FusedValue ppm 1], [OUT_TIMESTAMP Temperature FusedValue °C 2]}

This denotes that the user can add more information to the output in the assigner model. This more information added will also be stored and that will help us analyze if the system is stable. One more function that can be provided is that the user can also use this user-defined field to define the system stability/hard fault state. The hard fault cannot be recovered from and hence requires an additional diagnosis. For instance, the sensor fusion algorithm can provide a reliability layer to the system ignoring the faulty sensors and fusing the running sensors. But if all the sensors turn out to be faulty that will be considered as a hard fault. These hard fault situations can be found out using algorithms that can be defined in the assigner model. If the complete system is faulty, this field can be used as a flag that denotes 0 as a non-faulty state and 1 as a faulty state.

The assigner model in the informational layer can be defined in DEVS as the following atomic model:

```
AM = <S, X, Y, δint, δext, δcon, λ, ta>
S = {Active ∈{true, false}}
X = {Fusion_Message_IN}
Y = {Assigner_Message_OUT}
δint (s) = {if Active = true, then Active = false}
δext (s,e,x) = {if Active = false, then
                Receive the sequence container message from fusion model.
                Reformat the sequence container with a function call:
add_values_to_assigner_message with the parameter as container message and store it in
the container named ValueToSend.
                Add additional user defined details to the container
ValueToSend.
                Active = true
```

```
δcon = δint; δext;
λ (Active = true) = {Assigner_Message_OUT = ValueToSend; Call to the
function StoreData with parameters ValueToSend (This sends the data to the data storage)}
ta (Active = true) = SendingTime
ta (Active = false) = INFINITY
```

According to the definition, when the model is passive, it waits for a message from the sensor fusion model. Once the message is received, it calls the function add_values_to_assigner_message and sends the received message as a parameter. The function adds the standardized units to the sequence container and reformats it. The reformatted container is stored in the variable sequence container *ValueToSend.* Additional user details, if applicable, as mentioned in the earlier parts are then added to *ValueToSend.* The variable sequence container is then sent out to the output port Assigner_Message_OUT. The model remains in the active state during these functions and then it passivates itself waiting for another input.

3.3.2.4 Output Commander

All the models from the informational layer discussed previously receive the data from the sensors and send it to the service layer and later to the control system. After the control system receives the data, it takes the decision on the commands for the actuators. Therefore, there is an implementation of the output commander model that deals with the commands from the control system. The output commander model is the model that works in the reverse direction and the implementation is only for the actuators. This model receives a sequence container (that may contain multiple actuator commands) from the control system through the service layer, breaks the container according to the actuator type, and sends it to the cyber-physical layer actuator model.

The output commander model is a simple model that is responsible for connecting the service layer to the cyber-physical layer. In parallel to the other models in the informational layer, the output commander model works only when there are commands from the control system for any actuation. The model receives a sequence container that contains the actuation commands. If a command sequence container is sent by the service layer to the output commander model, it separates the data according to the actuator type, removes high-level details and converts it into a lower-level detailed message, and sends it to their actuator models. This conversion includes the transformation of standardized units to machine-level units. For instance, if the received command says that the motor should be set to 200 RPM and the microcontroller being used is an 8-bit microcontroller (the resolution of the bus will have 2^8 = 256). If the maximum RPM the motor can get to is 1000 RPM, the conversion will be (200/1000) * 256 ≈ 51. We call this conversion "mapping". Hence, the output commander model will send this mapped output to the specific motor model connected in the cyber-physical layer as the actuator model. If there are multiple actuators, there will be multiple instances of the actuator model, in addition to that there will be multiple outputs to the output commander model that goes to multiple instances of those actuator models.

The output commander performs an important role in the DCIF to manage the actuation commands. In the case of multiple actuators, it can be a difficult task to manage all the actuation. The output commander provides an essential operation that manages the actuation.

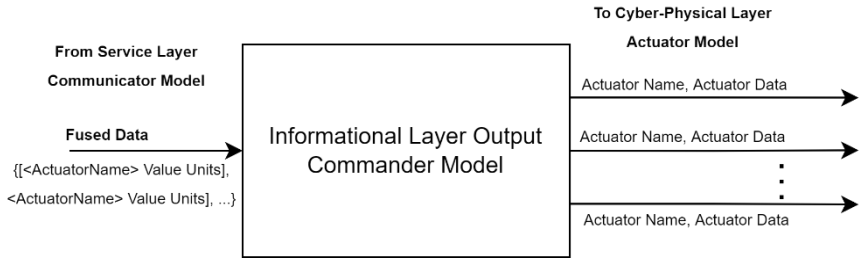

Figure 11. Informational Layer Output Commander Model Atomic

As shown in Figure 11, the output commander model has one input from the service layer communicator model and can have single or multiple outputs according to the number of actuators connected in the system. The output commander model receives a sequence container with all the actuation commands at a point in time, it sorts it according to the actuators and sends it to the actuator models. The connection to the actuators is not automatic in the defined DCIF, hence the programmer needs to specify the actuator output manually. The input and the output messages to the output commander model are defined in the following section.

The input message to the output commander model is in the format given as follows:

{[TIMESTAMP <ActuatorName> Value Units], [TIMESTAMP <ActuatorName> Value Units], …}

where TIMESTAMP is the time value at any specific point in time, ‘value’ is the actuation value to that particular actuator and ‘Units’ denotes the units for the actuation value.

The format of the output messages from the output commander model is in the format:

OUT_TIMESTAMP <ActuatorName> ActuatorData

OUT_TIMESTAMP <ActuatorName> ActuatorData

.

.

.

where OUT_TIMESTAMP is the time value at any specific point in time after the input arrives, ‘ActuatorData’ is the converted data by the output commander model according to the particular actuator.

The number of outputs will depend on the size of the input message sequence container. The size of the container will denote the number of actuators in the system and hence, the number of actuator models in the cyber-physical layer. These outputs will be given individually to the single input port of the instances of the cyber-physical layer actuator model to perform the actuation.

For instance, consider a system with 2 actuators, say Motor1 and Motor2. Both the motors are 1000RPM max speed and the microcontroller used is an 8-bit microcontroller. The speed will be mapped as 0RPM-1000RPM to 0-255. If the input from the service layer is as follows:

{[TIMESTAMP Motor1 300 RPM], [TIMESTAMP Motor2 800 RPM]}

Where TIMESTAMP is the time value at any specific point in time.

Since there are two actuators, there will be two outputs to the output commander model. The outputs from the commander model will be as follows:

TIMESTAMP Motor1 77 //at one output port connected to Motor1 actuator model.

TIMESTAMP Motor2 205 //at another output port connected to Motor2 actuator model.

The values are mapped and sent to the specific output port of the output commander model. Along with that, it can be seen that the input sequence container is sliced into parts and sent to the ports.

The output commander model in the informational layer can be defined in DEVS as the following atomic model:

OCM = <S, X, Y, δint, δext, δcon, λ, ta>

S = {Active ∈{true, false}}

X = {SL_Message_IN}

Y = {a1, a2, a3, …} *According to the number of actuators connected in the system and thereby the number of instances of the cyber-physical layer actuator model.*

δint (s) = {if Active = true, then Active = false}

δext (s,e,x) = {if Active = false, then

Receive the sequence container message from service layer communicator model.

Break the sequence container in pieces according to the number of elements in the container: *a1_message*, *a2_message*, …

Convert the messages' standardized units to lower-level details.

a1_MessageToSend = a1_message, *a2_MessageToSend = a2_message*, …

Active = true

δcon = δint; δext;

λ (Active = true) = {a1 = *a1_MessageToSend*, a2 = *a2_MessageToSend*, …;}

ta (Active = true) = SendingTime

ta (Active = false) = INFINITY

The output commander model is connected to the service layer communicator model to receive inputs and the cyber-physical layer actuator model to provide outputs. The number of instances of the actuator model depends on the number of actuators in the system, hence the number of outputs of the output commander model is dependent on the number of

actuators in the system. According to the formal definition, at an external event, i.e., an input from the service layer communicator model, the model checks its status. If it is in a passive state, the model receives and stores the receiving sequence container from the communicator model. The external transition function breaks the sequence container into pieces according to the number of elements in the container. For instance, if there are two elements in the received container, that means that there are two actuators in the system and the communicator model has a command to actuate them. Once the container is sliced, the standardized units from the container are mapped to the lower-level units. These messages are then sent to the different output ports according to the actuator. The model remains in an active state during the external event and passivates once the output has been sent.

3.3.3 Service Layer

In the previous section, we could see that the informational layer performs important tasks such as sorting the data, making the data reliable, and storing the data in the data storage along with many minor tasks. The service layer, instead, deals with the communication between the hardware and the control system. A control system is a logic unit that controls the cyber-physical system through the interface. According to the sensor inputs to the control system, it performs logical operations and responds back to the DCIF with a decisional command. The service layer performs a data/command transfer between the hardware and the control system. However, it does not directly communicate with the hardware, but it uses the informational layer and the cyber-physical layer to obtain secure data and transfer the commands as discussed before.

As discussed earlier, DCIF is completely time synchronized; therefore, it is important to keep the system in synchronization with the control system as shown in Figure 1. This is carried out by the service layer, which ensures that the communication between the control system and the hardware is live. The communication follows a protocol in the service layer that is independent of hardware, but it provides a set of important features that helps the cyber-physical system to work efficiently. The communication between the service layer and the control system is an abstraction over the hardware specifications which makes it independent of hardware.

The service layer is responsible to provide the following services to the control system that connects the hardware to the control system:

1) It provides an Application Programming Interface to connect the sensor/actuator to the control system through the informational and the cyber-physical layer.
2) It ensures that the hardware and the control system are connected and are transferring data.
3) It provides the data required by the control system on request by fetching it from the data storage.
4) It redirects the commands from the control system to the hardware using the informational and the cyber-physical layers.

All these tasks performed by the service layer follow a request-response communication protocol during the communication with the control system.

An Application Programming Interface (API) is a piece of software that helps in establishing a connection between two pieces of software. It is a type of software interface, offering a service to the other pieces of software. In DCIF, the service layer implements an

API to establish the connection between the service layer and the control system. That is, the service layer acts as a software interface between the informational layer and the control system. The informational layer, as discussed above, deals with the sensor data that is being sent out to its output port which is then received by the service layer and also stored in the data storage. The service layer contacts the control system and checks on what data does the control system need. After it receives a response from the control system, it fetches the data from the data storage or sends the current data to the control system according to the request. This is a request-response mechanism that works in the service layer.

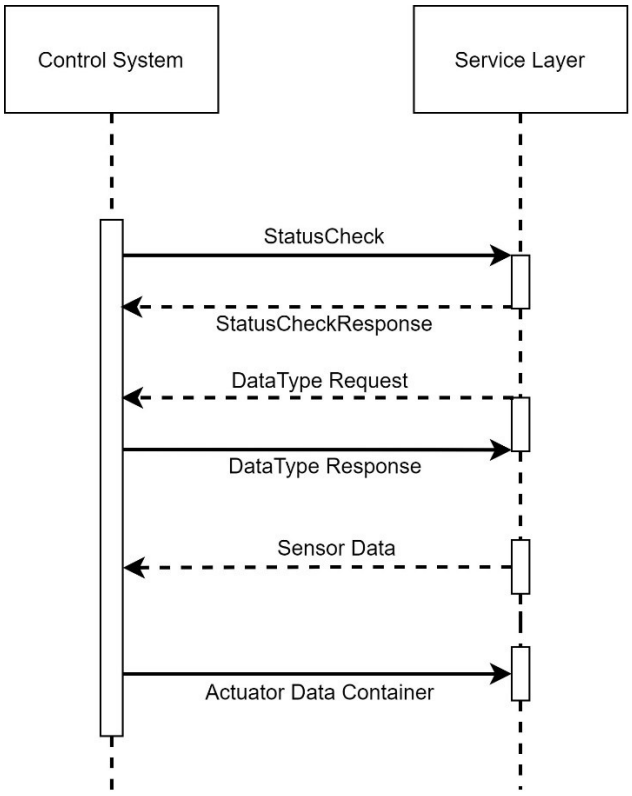

Figure 12. Service Layer API

Figure 12 shows how the request-response is conducted by the service layer. The control system sends a “StatusCheck” command to the service layer to check if the interface is alive. If the interface is alive and working, the service layer responds to the control system

with a positive response. Once the status is checked and is alive, the service layer requests the control system with the data type and the timestamp of the data that it requires. The control system responds with the required data type along with the timestamp. The service layer checks the timestamp, if the timestamp asks for current data, the service layer redirects the constant feed of the data from the informational layer to the control system. If the timestamp asks for older data, the service layer fetches the data from the data storage and sends it to the control system. However, the service layer remains always open for the actuator data from the control system, but the control system always needs to do the status check before sending the actuator data to the service layer. If any actuator data container is received at any point in time by the service layer, it transfers the container to the informational layer. The service layer uses this API to form a connection between the DCIF and the control system.

The service layer is made up of one DEVS model. We call it the communicator model. The communicator model is responsible for all the communication, synchronization, and data transfer between the service layer and the control system.

Due to the multiple tasks performed by one DEVS model, there are different states that the model goes through. The state changes for every action performed, and the following actions are dependent on this change. The communicator model has the following states: Passive, Active, ReadyToSend, CurrentData, and StorageData. These states hold a Boolean value and hence the value changes according to the inputs from the control system.

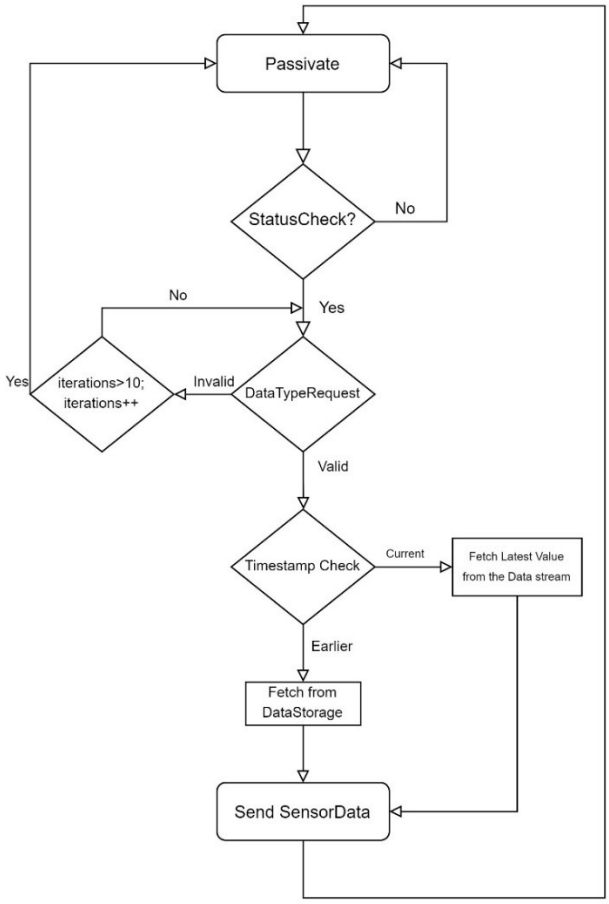

Figure 13. Communicator Model Flowchart

Figure 13 shows the flow chart of the working of the communicator model. According to the flow chart, the model remains in the passive state for a status check command from the control system. Once the status check command is received, the model reverts with the status and hence the state of the model becomes active. Only when the state is active, it can receive a data type request from the control system. If the data type request is received and the request is valid, the model changes its state to “ReadyToSend”. In the ReadyToSend state, the model decides on the type of data to be sent according to the received request. If the control system requests the current data, the model goes to a CurrentData state and

sends the data to the control system by fetching the latest value in the data stream received from the informational layer. And if the control system requests the StorageData, the model goes to a StorageData state and sends the data to the control system by searching the timestamp and the type of requested data from the storage. The model remains in an Active state during all stages and once the data has been sent, the model changes to a passive state. At any point in time, if there is an error, the model tries to passivate itself for that timestamp and checks for the StatusCheck request as soon as it is passive. In parallel to this and independently, the communicator model waits for any actuator commands from the control system and sends them to the informational layer as soon as it receives them.

Figure 14. Service Layer Communicator Model Atomic

To carry out the communication and the data transfer, the communicator model has separate communication I/O lines and data I/O lines as shown in Figure 14. The communication lines in the communicator model are only responsible for sending the state changes to the control system. These communication lines have a Boolean value i.e., true, and false. The data lines however are responsible for the data transfer between the control system and the service layer's communicator model. The data lines inputs/outputs carry

the sensor data or the actuator data only. The communicator model consists of two inputs, two outputs, and Tx Rx (1 input and 1 output) command lines. The data storage is accessed by the communicator model using the specific calls to the functions that are allowed to read and write the storage with the data as parameters. As shown in the figure, the format of the input message to the service layer communicator model is the same as the output of the informational layer assigner model,

{[TIMESTAMP <SensorName> FusedValue Units UserData], …}

where TIMESTAMP is a time value at any point in time, <SensorName> is the name of the sensor, FusedValue is the value obtained after the sensor fusion, Units is the standardized units for the sensor reading and UserData is any additional data specified by the user as explained in the informational layer assigner model.

The Tx line is the communication line and hence the format of the message going through the Tx line is in Boolean format. However, the Rx line acts as a Boolean for a status check but can receive the timestamp and the sensor type string once the status check is true. For example,

TIMESTAMP 0 //for the status check

TIMESTAMP 1

TIMESTAMP 00:10:00 Temperature //after status check

where TIMESTAMP is a time value at any point in time.

The format of the message of the actuation data input from the control system is as follows:

{[TIMESTAMP <ActuatorName> Value Units], …}

where TIMESTAMP is a time value at any point in time, <ActuatorName> is the name of the actuator (e.g., Motor), Value is the standardized value of the actuation and Units is the standardized unit for the actuation value (e.g., RPM).

The communicator model redirects this actuation message from the control system unchanged to the informational layer. The Communicator model in the service layer can be defined in DEVS as the following atomic model:

```
CM = <S, X, Y, δint, δext, δcon, λ, ta>
S = {Active ∈{true, false}, ReadyToSend ∈{true, false}, CurrentData ∈{true,
false}, StorageData ∈{true, false}}
X = {InformationalLayer_Message_IN, Rx, Rx_TypeAndTime,
Actuation_Command_IN}
Y = {CS_Message_OUT, Tx, Actuation_Command_OUT}
δint (s) = {if Active = true, then Active = false; if ReadyToSend = true,
then ReadyToSend = false; if CurrentData = true, then CurrentData = false;
if StorageData = true, then StorageData = false}
δext (s,e,x) = {
        if received StatusCheck at Rx
        if Active = false, then
        if Rx = true, then ReadyToSend = true, Active = true

        if Active = true, then
        if ReadyToSend = true, then if Rx_TypeAndTime = [“CUR” “Type”]
        ValueToSendCS = InformationalLayer_Message_IN
        CurrentData = true

        if Active = true, then
        if ReadyToSend = true, then if received at port Rx_TypeAndTime =
[“TIMESTAMP” “Type”]
        ValueToSendCS = ReadStorage ([“TIMESTAMP” “Type”])
        StorageData = true
```

```
                    if Active = false, then
                    if message received at port Actuation_Command_IN
                    ValueToSendIL = message at port Actuation_Command_IN
                    Active = true

            δcon = δint; δext;
            λ (Active = true && ReadyToSend = true) = {Tx = ReadyToSend}
            λ (Active = true && CurrentData = true && ReadyToSend = true) =
{CS_Message_OUT = ValueToSendCS}
            λ (Active = true && StorageData = true && ReadyToSend = true) =
{CS_Message_OUT = ValueToSendCS}
            λ (Active = true) = {Actuation_Command_OUT = ValueToSendIL}
            ta (Active = true && ReadyToSend = true) = SendingTime
            ta (Active = true && CurrentData = true && ReadyToSend = true) = SendingTime
            ta (Active = true && StorageData = true && ReadyToSend = true) = SendingTime
            ta (Active = true) = SendingTime
            ta (Active = false) = INFINITY
```

According to the formal definition, the input to the model triggers the external transition function. If the model receives a status check, it responds with the status ReadyToSend which changes to a Boolean value true. If the *ReadyToSend* changes to true, the model listens for the timestamp and the type of the sensor that needs to be sent to the *CS_Message_OUT* port. If the model receives "CUR" along with the sensor type then, the model receives the data from the Informational layer through the port *InformationalLayer_Message_IN* and sends it to the output port *CS_Message_OUT*. However, if it receives an older timestamp along with the sensor type, then the model searches the specific data at that time stamp in the data storage and sends it to the output port *CS_Message_OUT*. If the model receives input through the port *Actuation_Command_IN*, it redirects the same input to the output port

Actuation_Command_OUT. The time advance for all the states can be the same or different. After the output is sent to the output ports, the model passivates, changes all the states to false, and hence gets ready for the next set of inputs.

The communication of the service layer with the control system is hardware-independent and hence, the same formalization can be used with a different communication type. The output of the model depends on the state change. Therefore, if the output function contains calls to the functions that enable a different communication technique, it should work for this model. For instance, consider a system with a microcontroller that has an on-chip Wi-Fi transceiver and an implementation of DCIF. The control system in this particular cyber-physical system can be embedded in the same microcontroller or in a different microcontroller that has an on-chip Wi-Fi transceiver as well. In the latter case, the communication with the control system microcontroller will occur through the Wi-Fi connection with or without the internet but using the data packets. For example, it can use an HTTP request-response with or without the internet where the control system microcontroller will host the webserver. That is, the microcontroller with the implemented DCIF will communicate using the service layer with the control system microcontroller by calling the HTTP methods in the output function to send and receive the data. Due to this flexibility, the DCIF can be easily implemented on any hardware regardless of the communication type or the hardware used.

Chapter 4: Application: Case Study

In the previous chapter, we discussed the structure and functionality of the proposed framework. This chapter discusses the implementation of the framework in Cadmium and RT-Cadmium simulation tools to validate the proposed framework by simulating and implementing the model on a real-time hardware platform. The following section is divided into two parts, simulation, and real-time hardware implementation. This chapter will also portray the application point of view of the proposed framework along with a case study implementation to support it.

DCIF uses the same structure of the DEVS atomic models discussed in Chapter 2. The following section explains the implementation of the DCIF models in detail along with the explanation of some special functions that are being used in these models.

As explained in Chapter 3, DCIF includes 7 DEVS Atomic models distributed in three layers: cyber-physical, informational, and service. Figure 15 shows the DCIF framework top model. As previously explained, the cyber-physical layer consists of the sensor and actuator models, which have access to the hardware directly through drivers. They connect to the informational layer, a coupled model with 4 DEVS atomic components: Sorter, Sensor Fusion, Assigner, and Output Commander. The first three are responsible for making the sensor data obtained orderly and reliable by implementing the sensor fusion algorithm; the output commander model is responsible to provide the actuator model with commands. Lastly, the service layer consists of one DEVS atomic model: the communicator model, which provides an API for the communication between the DCIF and the control system.

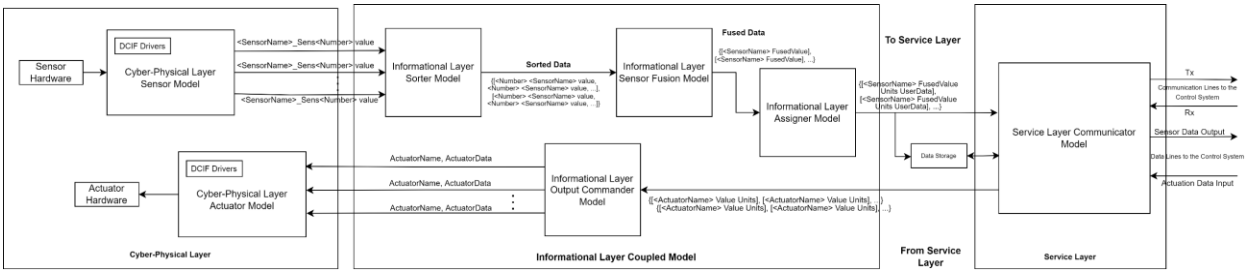

Figure 15. DEVS Cyber-Physical Interface Framework (DCIF) Model

We implemented the DCIF in Cadmium simulation tool. In the following section, we will go over a few of the model implementations in detail to portray how the models are built. The remaining models are provided in the Appendix for the readers' convenience.

4.1 DCIF Simulation

As discussed in section 3.3.1, the cyber-physical layer consists of sensor and actuator models. The models have been implemented according to the formal definitions provided in section 3.3.1 and implemented as in Figure 16, which shows a code snippet for the *sensor model* implementation, which polls the sensor values from the hardware using the DCIF drivers and provides it to the informational layer sorter model.

```
void internal_transition() {
    int upper = 30;
    int lower = 20;

    #ifdef RT_ARM_MBED
    state.Sensor->getData(state.output);
    #else

    state.output = (rand() % (upper - lower + 1)) + lower;
    state.message.name = state.name;
    state.message.value = state.output;
    }

void confluence_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
      internal_transition();
    }

typename make_message_bags<output_ports>::type output() const {
    typename make_message_bags<output_ports>::type bags;
      get_messages<typename defs::out>(bags).push_back(state.message);
      return bags;
    }
```

```
TIME time_advance() const {
      return pollingRate;
    }
```

Figure 16. Sensor Model Implementation

According to the formal definition, the sensor model accesses the sensor hardware using the drivers. Hence, there is just one output port to the model. Since there are no inputs to the model, the model does not anytime make an external transition. Therefore, we do not define an external transition function. In the internal transition function, we have two options depending on if the model is being compiled for simulation or real-time execution. The *getData()* function calls the sensor driver, which returns the value read by the sensor (the sensor drivers and their implementation will be discussed in detail in the real-time implementation section). When compiling the model for simulation, the driver functions are replaced by a random value generator that accepts a lower bound and an upper bound to generate a value. According to the DEVS formalism, after executing the internal and external transition functions, the model should produce an output. Hence, the *output()* function is called to send an output *message* to the output port *out*, which contains the name and the value of the sensor. According to the formal definition, the model advances its time as per the defined polling rate.

The output from the cyber-physical layer sensor model (as well as input messages) uses a message structure defined as follows:

```
struct Sensor_Message{
    Sensor_Message(){}
    Sensor_Message(string i_name, double i_value)
   :name(i_name), value(i_value){}

   string   name;
   double  value;
};

istream& operator>> (istream& is, Sensor_Message& msg);

ostream& operator<<(ostream& os, const Sensor_Message& msg);
```

Figure 17. Sensor Message Structure

As discussed in the previous chapter, the message from the sensor model has the following format, defined in Figure 17:

TIMESTAMP <SensorName>_Sens<Number> value

That is, the sensor message consists of a string for the sensor name and a float/double for the sensor value.

During the simulation of the sensor and the actuator model, the drivers are not used; instead, we generate values at random or provide inputs manually (as shown in Figure 18).

```
const char* D3 = "./inputs/Temperature_Sensor_Values1.txt";
AtomicModelPtr CPL_Sensor1 =
cadmium::dynamic::translate::make_dynamic_atomic_model<CPL_Sensor, TIME>("CPL_Sensor1",
D3);
```

Figure 18. Sensor Model instantiation with file path defined in main.cpp

The implementation of the remaining models in DCIF (found in the Appendix) overall follows the same methodology: we formally define the models; we define the message structure and additional implementation details. Following, we discuss some important aspects of some of these atomic models.

The actuator model is similar to the sensor model: when an input is received, the actuator model provides the output to the actuators using the drivers. For simulation, we use text-based outputs for the actuator model.

The *sorter model* sorts the sensor data and rearranges them into a vector to facilitate the work of the sensor fusion model. We convert the data to the *Sorter_Message* format that contains a number, sensor type, and value.

The sensor fusion model as discussed in section 3.3.2.2 implements the sensor fusion algorithm, and it performs sensor fusion on the data received from the sorter model.

The method used in the implementation of the DCIF is a fusion algorithm that determines the integrated support degree score of each sensor based on principal component analysis. The algorithm defines the fuzzy-index function as a support degree matrix of sensors. By principal component analysis, each sensor's integrated support degree score is obtained. According to their scores, valid observation values of sensors are determined and fused by allocating corresponding weight coefficients, so that the final expression of data fusion and estimation is obtained. The algorithm needs less calculation and can objectively reflect the mutual support degree of sensors without knowing any prior knowledge.

To find the integrated support degree score, we need to start with finding the support degree matrix. Assume that there are n sensors to measure the same target parameters. Let, x_i and x_j denote the sensor readings of the sensor i and j respectively. In order to express the error between x_i and x_j the fuzzy-index function d_{ij} :

$$d_{ij} = exp^{-|x_i - x_j|} \tag{1}$$

Where i = 1,2,…,n. and d_{ij} forms the support degree matrix D

The fuzzy-index function d_{ij} only denotes the support degree of sensor i by sensor j, but it does not reflect the integrated support degree score of sensor i by all the sensors of the same type.

Hence, adopting the principal component analysis to calculate the integrated support degree score of all the sensors. We need to calculate the eigenvalues and eigenvectors of the matrix D: Eigen Values = $\lambda_1, \lambda_2, \ldots, \lambda_n (\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n)$ and corresponding Eigen Vectors = $T_1, T_2, \ldots, T_n$ where $T_i = [t_{i1}, t_{i2}, \ldots, t_{in}]^T$ (i = 1, 2, …, n)

To calculate the principal component y_i (i = 1, 2,…,n):

$$y_1 = T_1^T D = t_{11}D_1 + t_{12}D_2 + \cdots + t_{1n}D_n$$

$$y_2 = T_2^T D = t_{21}D_1 + t_{22}D_2 + \cdots + t_{2n}D_n$$

…

$$y_n = T_n^T D = t_{n1}D_1 + t_{n2}D_2 + \cdots + t_{nn}D_n$$

Where $y_1, y_2, \ldots, y_n$ are the first principal component, the second principal component, …, and the n^{th} principal component, respectively. D_i is the i^{th} column of matrix D.

Now, we calculate the contribution rate of the kth principal component by,

$$\alpha_k = \frac{\lambda_k}{\sum_{i=1}^{n} \lambda_i} \tag{2}$$

Where k= 1,2,…,n

We calculate the accumulated contribution rate of the m principal components by,

$$\varphi_m = \sum_{k=1}^{m} \alpha_k \tag{3}$$

where ϕ_m indicates the ratio of the first m principal components over all information of the original variable. In practical application, the number of principal components is determined by criterion: ϕ_m>85%. Thus, the first m principal components are $y_1, y_2, \ldots, y_m$.

Now, we calculate the integrated support degree score of all the sensors by,

$$Z = \sum_{i=1}^{m} \alpha_i y_i \tag{4}$$

Where i = 1,2,…,m

If a sensor is not supported by more than 2/3 sensors, the sensor is regarded as a faulty sensor.

Therefore, only if $|z_j| < \left|\frac{1}{n}\sum_{i=1}^{n} z_i\right| * 0.7$, j = 1,2,…,n is satisfied, the sensor should be regarded as working and fault-free, otherwise, the sensor should be disregarded for the fusion.

For the working sensors, the weight coefficient should be calculated by,

$$\omega_i = \frac{z_i}{\sum_{i=1}^{n} z_i}, i \neq j \tag{5}$$

After finding the weight, the final fusion expression is obtained by,

$$\hat{x} = \sum_{i=1}^{n} \omega_i x_i, i \neq j \tag{6}$$

This is the fused value obtained after performing the sensor fusion algorithm. The implementation of the sensor fusion algorithm is done in Cadmium simulator and has been tested with different sensors and with different scenarios. The complete algorithm functions implementation can be found in the Appendix. As seen in the Appendix, the external transition function of the sensor fusion model calls these functions. The sensor data received by the sensor fusion model in the external transition function is given to the algorithm functions for calculation.

We built support functions (found in the Appendix), for instance, a method to find the support degree matrix of the sensors. Similarly, we implement functions to calculate the contribution rate and the accumulated contribution using the equations mentioned earlier. The function *compute_integrated_support_degree_score()* shown in Figure 19 performs the mathematical operations in the equation $Z = \sum_{i=1}^{m} \alpha_i y_i$, $i = 1,2,\ldots,m$ and returns the Z matrix that denotes the integrated support degree score.

```
MatrixXd compute_integrated_support_degree_score(vector<double> sensorinputs, MatrixXd
list_of_alphas, MatrixXd list_of_phi, MatrixXd dmatrix, int size){
    MatrixXd Z, Z_final;
    MatrixXd list_of_m_phi;
    int i;
    MatrixXd evec_i = eigen_vector_calculation(sdm_calculator(sensorinputs,size));
    MatrixXd y = dmatrix * evec_i;
    VectorXd alphas(Map<VectorXd>(list_of_alphas.data(),
list_of_alphas.cols()*list_of_alphas.rows()));
    Z = y.array().rowwise() * alphas.transpose().array();
    Z = (Z /100);
    Z_final = Z.rowwise().sum();
    return Z_final;
 }
```

Figure 19. Integrated Support Degree Score Calculation function

As seen in Figure 15, the sensor fusion model has one input and one output. The external transition function receives a vector of vectors of type *sorter message*. If the vector contains more than one element, then there are sensors of multiple types; else there is one type of sensor. For each type of sensor, we store the sensor readings. For all the values stored that corresponds to the same type of sensor, the sensor fusion algorithm function shown in Figure 20 returns the fused value. The returned fused value along with the sensor type name for each type of sensor has transferred a vector of type *Fused_Message* sent to the output port.

```
double faulty_sensor_and_sensor_fusion(MatrixXd Z, vector<double> inputsensors,
                            double criterion, int size){
    int i, tempfault=0,j=0;
    double *weight = (double *) malloc(sizeof(double)*(size));
    int *fault = (int *) malloc(sizeof(int)*(size));
    double average, sum=0,calculation=0,fusion_value=0;

    sum = Z.sum();
    average = fabs((sum/size))*criterion;
    double *Z_array = Z.data();
    for(i=0;i<size;i++){     //Identify faulty sensor and store index in array
  if(fabs(Z_array[i])<average){
    tempfault = i;
    fault[j]=tempfault;
    j++;
   }
    }
    //Making the reading and score of all faulty sensors to zero
    for(i=0;i<j;i++){
  Z_array[fault[i]]=0;
  inputsensors[fault[i]]=0;
    }
    //Assigning the sum of integrated support degree score of all
    //NON-FAULTY sensors to 'calculation' variable
    for(i=0;i<size;i++){
  calculation += Z_array[i];
    }
    //Creating an array of weight coefficients of each sensor by dividing a
    //sensor's integrated support degree score by sum of all scores
    for(i=0;i<size;i++){
  weight[i] = Z_array[i]/calculation;
    }
    //Calculating the fused value as a summation of product of
    //weight coefficient and sensor reading
    for(i=0;i<size;i++){
  fusion_value += weight[i] * inputsensors[i];
    }
    free(weight);
    free(fault);
    return fusion_value;
}
```

Figure 20. Sensor fusion and weight calculation

The sensor fusion algorithm function *faulty_sensor_and_sensor_fusion()* shown in Figure 20 performs the sensor fusion algorithm using the principal component analysis. This function takes the following parameters:

a. Sensor inputs: the inputs from the sensors provided by the cyber-physical layer are stored in a vector/array.
b. List of alpha: This is the list of contribution rates of each sensor.
c. Integrated Support Degree Score: This is calculated by multiplying the contribution rate of each sensor with the principal component.
d. List of phi: This is the accumulated contribution rate by certain principal components.
e. Support degree matrix: This is a matrix that consists of the score of support of one sensor to other.
f. Criterion: This is the faulty sensor criterion that helps in canceling out the faulty sensor.

In the *faulty_sensor_and_sensor_fusion()* function, the algorithm checks for the faulty sensors by finding out the average using the given criterion. The integrated support degree score is compared with the average and hence the faulty sensor readings are assigned a 0 value so that they would not affect the fusion. This function also finds out the weight coefficients for all the sensors by implementing the formula mentioned previously. The average of the weight multiplied by the input values of the sensor readings is known as the fused value that is then returned to the calling function.

The assigner model as discussed in section 3.3.2.3 implements the formal definition in Cadmium which receives the data from the sensor fusion model, makes it more orderly by adding the units and stores the data in a data storage.

```
void StoreData(Vector_Assigner_Message message) {
fstream fout;
time_t seconds = time(NULL);
fout.open("../DataStorage.csv", ios::out | ios::app);
for(auto i=0;i<message.message.size();++i){
    char buf[40];
    strftime(buf,40, "%Y-%m-%dT%H:%M:%SZ,", localtime(&seconds));
    fout<< buf;
    fout<<message.message[i].type << "," << message.message[i].value<< "," <<
message.message[i].units <<endl ;
    }
}
```

Figure 21. Implementation of the data storage for simulation

During the simulation of the model, the data storage is done in a file that can be read and written. Figure 21 shows a code snippet of the function *StoreData()*. Each element in the vector *message* is written to the file by using the *fout* command. The format of storing the data is yyyy-mm-ddThh:mm:ssZ. The simulation and the hardware implementation of the data storage follow the same format where it uses the real system time as a timestamp to store the data.

The output commander model as discussed in section 3.3.2.4 receives commands and sends them to the actuator models. The external transition function of the output commander model receives a vector of type *Actuator_Message*. The vector is checked to obtain the number of actuator commands present. For simulations, we use two output ports, and hence, there will be at most two values in the vector for the two actuator models connected to this model. Particular actuator messages are sent to their actuator models through the output ports.

The communicator model as discussed in section 3.3.3 plays an important role in keeping the hardware and the control system connected. The model uses 4 input ports and 3 output

ports. As discussed in the previous chapter, the communicator model contains command lines and data lines. Figure 22 shows a code snippet for the *communicator model* implementation.

```
void internal_transition (){
  state.busy = false;
  state.send_current_data = false;
  state.ready_to_send = false;
  state.send_actuator_command = false;
  state.status_check = false;
  state.to_IL.name = "\0";
  state.to_IL.value = 0;
  state.active = false;
}

void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs){
...

  status = get_messages<typename SL_Communicator_defs::command_line_rx_status>(mbs);
  message=get_messages<typename SL_Communicator_defs::command_line_rx_message>(mbs);
  if(status.size() >= 1){

  if(status[0].status_check==1 && state.status_check==false && state.woke_up==false)
    {
     if(!state.busy) {state.status_check = true;}
     state.tx.status_check_response = 1;
     state.tx.type_and_time_request = 1;
     state.woke_up = true;
  }
  }

...

  if(message.size()>=1) {
    if(!message[0].message.name.empty() && state.woke_up == true) {
    state.to_IL.name = message[0].message.name;
    state.to_IL.value = message[0].message.value;
    state.send_actuator_command = true;
    state.busy = true;
    }
    }
state.active = true;
}

typename make_message_bags<output_ports>::type output() const {

  typename make_message_bags<output_ports>::type bags;

  if(state.woke_up == true && !state.busy)
    get_messages<typename defs::command_line_tx>(bags).push_back(state.tx);

  if(state.woke_up == true && state.ready_to_send == true)
   get_messages<typename defs::sensor_data_out>(bags).push_back(state.send_to_CS);

  if(state.woke_up == true && state.send_actuator_command == true)
   get_messages<typename defs::actuator_data_out>(bags).push_back(state.to_IL);

  if(state.woke_up == true && state.send_current_data == true) {
   for(auto i=0;i<state.from_assigner.message.size();++i){
    get_messages<typename defs::sensor_data_out>(bags).
                  push_back(state.from_assigner.message[i]);
   }
 }
}
```

```
TIME time_advance() const {
 if(state.active && state.woke_up && !state.busy) { return TIME("00:00:00:200"); }
 if(state.active && state.woke_up && state.ready_to_send){return TIME("00:00:00:200");
   }
 if(state.active && state.woke_up && state.send_actuator_command) {
   return TIME("00:00:00:200");
   }
 if(state.active && state.woke_up && state.send_current_data) {
   return TIME("00:00:00:200");
   }
return std::numeric_limits<TIME>::infinity();
}
```

Figure 22. Implementation of Communicator Model

The communicator model interfaces with the control system to provide it with two services: 1. Provide the control system with the required data and 2. Transfer the actuation data to the informational layer.

The external transition function receives the status check at the input port *command_line_rx_status,* the type and time of the requested data at the input port *command_line_rx_type_and_time* and an actuation command at the input port *actuation_message*. When the status check is received, the *status_check_response*, *type_and_time_request* and *woke_up* are set to true. The next input received by the model is the type and the time of the sensor data. The data for the received time stamp is searched through the data storage and hence if the match is found, the state *ready_to_send* changes to true. If there is an actuator command received at the port *actuation_message* and the state *woke_up* is true, the state *send_actuator_command* is changed to true. The internal transition function resets all the states except *woke_up*. The *output()* function checks states *status_check_response* and *type_and_time_request;* if they are true, then Boolean value *true* is sent to the output port *command_line_tx*. If *send_current_data* state is true, then the model redirects the input from the input port *from_informational_layer* to the output port *sensor_data_out*. If the state ready_to_send is true, the searched data from the data storage is sent to the output port *sensor_data_out*. If the state *send_actuator_command* is true, the

received actuation command is sent out through the output port *actuator_data_out*. The model responds instantaneously and hence, the time advance is zero.

4.2 Real-Time Implementation

The models above can be used to study the behavior of the system using simulation, after which the model can be deployed to the target platform to test it in real-time. To do that we use the RT-Cadmium extension.

RT-Cadmium is a Real-Time DEVS kernel developed on the top of Cadmium DEVS simulator. RT-Cadmium drops the use of the simulated clock information and instead uses the hardware real-time clock. RT-Cadmium also allows users to seamlessly switch between the simulation and deployment of the model with ease. RT-Cadmium is portable, and it supports Mbed Enabled ARM microcontroller and microprocessors and Linux-based systems. In this implementation, we port the simulated implementation of the DCIF explained previously to RT-Cadmium with minimal changes that are only hardware-based. We use a NUCLEO-H743ZI Nucleo-144 development board for the deployment of the DCIF model. This is an STM32H7 ARM Cortex-M7 based microcontroller in LQFP144 package. This board supports on-board STLINK-V3 debugger/programmer with SWD connector with a USB re-enumeration capability. It has an onboard 32.768 kHz crystal oscillator, and it also supports Ethernet with IEEE-802.3-2002. This microcontroller provides both good performances along with lower power consumption and it does not require a separate probe for communication since it uses the ST-LINK programmer/debugger.

We used temperature and humidity sensors, CO2 sensors, and light intensity/ambient light sensors to build and test a model in real-time. Table 2 shows which sensors were used.

Sensor Type	Sensor Name
Temperature and Humidity Sensors	DHT11
CO2 sensors	MG811
Ambient Light Sensor	VEML7700
Motion/Occupancy detector	Generic PIR Sensor

Table 2. Sensors used for the deployment

We use 5 sensors of each type in the deployment. The sensors are connected to the NUCLEO-H743ZI microcontroller using I2C for VEML7700, digital I/O for the DHT11, and analog I/O for the MG811 sensors. Figure 23 shows the connections of sensors to the microcontroller. The sensors on the left side shown in grey color are the MG811 CO2 sensors connected to the microcontroller, the one on the top left and the top are the VEML7700 ambient light sensors and the one connected on the right side of the figure are the DHT11 temperature and humidity sensors.

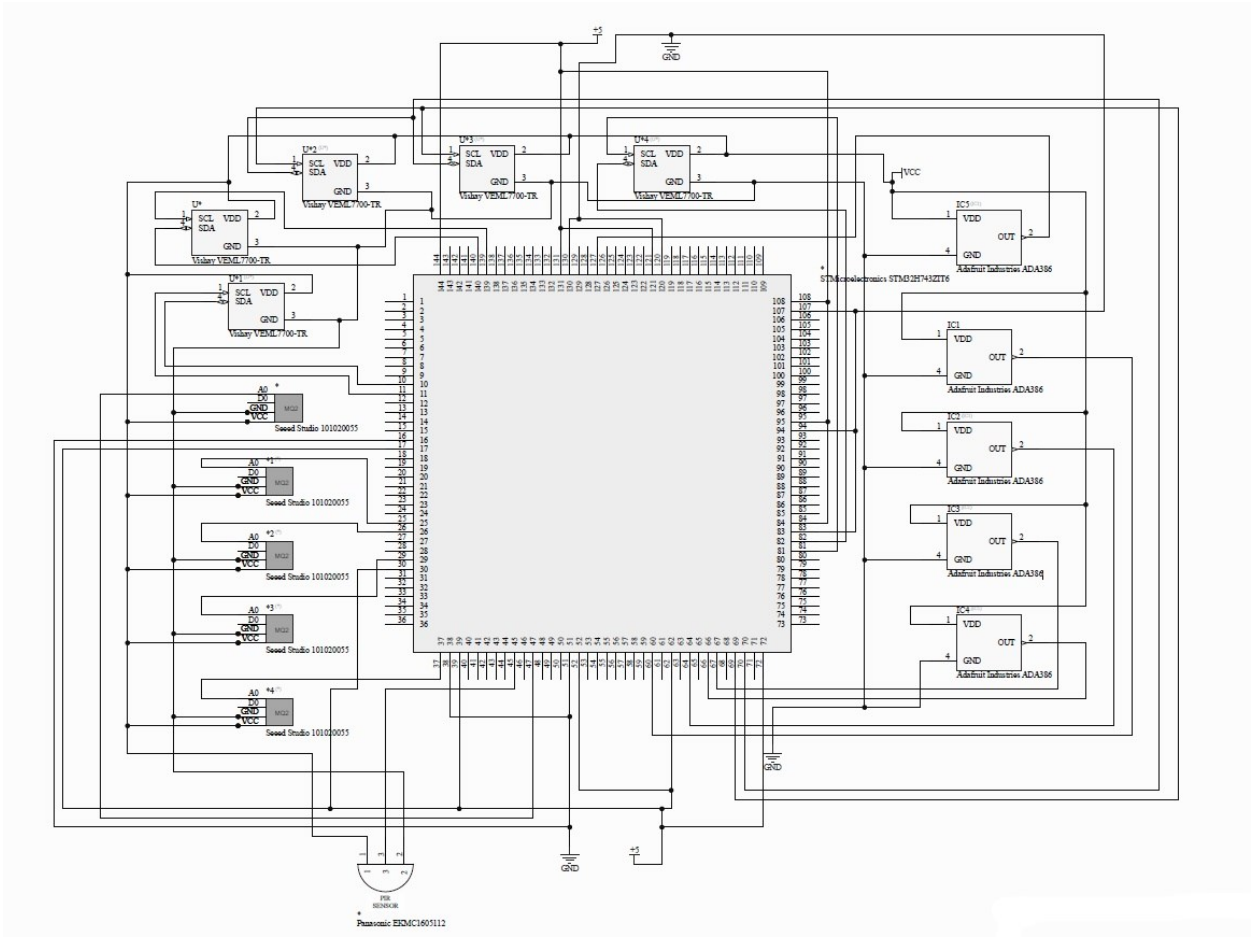

Figure 23. Circuit Schematic for the connections

The implementation of the DCIF model for the deployment has some changes in comparison to the simulation explained earlier. The changes or additions are seen in the cyber-physical layer sensor model, cyber-physical layer actuator model, and the informational layer assigner model. In the simulation of the model, we use a random value or manual text-based inputs and outputs for sensor and actuator models, respectively. Here, we define hardware driver classes that are dependent on the manufacturer's sensor libraries (which are specifically defined for particular hardware to control a sensor). The sensor library deals with reading and writing the hardware registers along with using the

hardware-specific resources such as the timers for the sensor to return the value obtained from the surrounding.

To simplify the design, we built a wrapper that hides the complexities of the sensor library. We create a namespace that includes a set of classes that implements an easy method of using the sensor libraries discussed above. We create classes of easier names for instance *TEMPERATURE* or *HUMIDITY*. In their classes, we initialize an object of the class that has been defined in their manufacturers' sensor library. A function *getData()* is defined that calls the functions from the manufacturers' sensor library class. When we want to use the required sensor in the implementation, we instantiate an object of the class we defined and call the function *getData()*, as shown in Figure 24.

```
drivers::TEMPERATURE* Sensor = new drivers::TEMPERATURE(Pin);
  //The Pin variable is the pin name for at which the sensor output is connected
Sensor->getData(output);
```

Figure 24. Wrapping drivers in the atomic model

We define the pin number of the sensor connection when we instantiate the atomic model. This method wraps all the sensor libraries into one file that defines functions that are consistent for all types of sensors.

The cyber-physical sensor model is designed in such a way that it supports the sensor connected to the microcontroller through a Digital I/O, Analog I/O, I2C communication, SPI Communication, or UART. To do that, the sensor model implements overloaded constructors which in common has *pollingRate* and the sensor name as arguments. This overloading allows us to instantiate the sensor model according to the sensor connected. For instance, if a sensor is connected through SPI communication we instantiate the class with the constructor that takes pins *MOSI*, *MISO,* and *SCLK*.

The cyber-physical sensor model when instantiated as an atomic model, calls the constructor through which the sensor connects to the microcontroller. For instance, the temperature sensor uses a digital I/O to connect to the microcontroller. If we use 5 temperature sensors connected to the microcontroller with a *pollingRate* of 5 seconds and the name of the sensors are given as "Temp_Sens1", "Temp_Sens2", "Temp_Sens3", "Temp_Sens4" and "Temp_Sens5" and they are connected to the pin numbers D3, D5, D7, D8, and D10 of the microcontroller. Then they should be instantiated as shown on Figure 25.

```
AtomicModelPtr CPL_Sensor1 = cadmium::dynamic::translate::make_dynamic_atomic_model
   <CPL_Sensor, TIME>("CPL_Sensor1", D3 ,TIME("00:00:05:000") , "Temp_Sens1");
AtomicModelPtr CPL_Sensor2 = cadmium::dynamic::translate::make_dynamic_atomic_model
   <CPL_Sensor, TIME>("CPL_Sensor2", D5,TIME("00:00:05:000"), "Temp_Sens2");
AtomicModelPtr CPL_Sensor3 = cadmium::dynamic::translate::make_dynamic_atomic_model
   <CPL_Sensor, TIME>("CPL_Sensor3", D7,TIME("00:00:05:000"), "Temp_Sens3");
AtomicModelPtr CPL_Sensor4 = cadmium::dynamic::translate::make_dynamic_atomic_model
   <CPL_Sensor, TIME>("CPL_Sensor4", D8,TIME("00:00:05:000"), "Temp_Sens4");
AtomicModelPtr CPL_Sensor5 = cadmium::dynamic::translate::make_dynamic_atomic_model
   <CPL_Sensor, TIME>("CPL_Sensor5", D10,TIME("00:00:05:000"), "Temp_Sens5");
```

Figure 25. Sensor Atomic Model instantiation for hardware deployment

For the actuator model, there are two ways actuation can be carried out.

1. We call the hardware-specific drivers (which are the same as the sensor drivers) to actuate in the actuator model output function.
2. We connect the output of the actuator model to one of the implemented digital or analog output RT-DEVS models.

We use the second method to call the hardware-specific actuation functions. The models *analogOutput* and *digitalOutput* are the DEVS models that receive an input and execute the call to the Mbed specific digital and analog output classes *DigitalOut* and *AnalogOut* in the output function. For the actuators, the implementation of DCIF only supports digital

and analog outputs at the moment but it can be extended for actuations with more complicated communication protocols.

During the deployment of the model on the target platform, the assigner model needs some change for storing the data in the data storage. While we compile the model for simulation, we do the data storage in a file locally. But in the hardware deployment we implement the data storage using three different methods:

1. The data can be printed onto the console of the computer that the hardware is connected to and storing the data in a file by transferring the console details.
2. The data can be stored in an SD card or an OTG storage connection since the hardware supports it.
3. The data can be stored in the cloud using Ethernet connectivity onboard the hardware.

The first two of the methods have been implemented in the implementation of the model. The third method can be implemented but it is outside the scope of this thesis.

During the implementation of DCIF on the target platform, we observed that using the GNU GSL library for eigenvalue and eigenvectors calculation is not possible for the choice of hardware we are using. As mentioned earlier, we use an ARM Cortex M7 based microcontroller, but the GSL library cannot be compiled for it as it depends on precompiled linker files that are not compiled for the ARM architecture. Hence, we only used the Eigen library

4.3 Case Studies

As discussed in the previous parts of this chapter, we implemented a simulation for the DCIF model and later we performed a hardware deployment on the hardware mentioned previously. This section of the chapter will discuss the results and the outputs we see for the simulation and deployment on the target platform. During the simulation, we test the DCIF model with multiple test cases. Here, we show three test cases:

1) A scenario where we have one sensor of one type. For instance, a temperature sensor.
2) A scenario where we have multiple sensors of one type. For instance, 8 temperature sensors.
3) A scenario where we have multiple sensors of multiple types. For instance, 4 temperature sensors and 4 CO2 sensors.

During the simulation with one sensor of one type connected in the system, we do not use the sensor fusion model since it is not required. Hence, the system inputs and outputs are fairly simple. We provide the model with the text-based inputs and the text-based outputs are obtained. For one sensor being used in the system, the outputs logs from the Cadmium simulator for DCIF look like as shown in Figure 26.

```
00:00:00:000
[CPL_Sensor_defs::out: {Hum_Sens1 75.3}] generated by model CPL_Sensor1
…
00:00:02:000
[] generated by model CPL_Sensor1
[IL_Sorter_defs::out: {1 Humidity 75.3}] generated by model IL_Sorter1
…
00:00:03:000
[] generated by model CPL_Sensor1
[] generated by model IL_Sorter1
[IL_Assigner_defs::out: {Humidity 75.3 RH}] generated by model IL_Assigner1
…
```

Figure 26. Simulation output for one sensor in the system

The sensor model receives the first value at the timestamp 00:00:00:000. The sorter model rearranges it according to the sorter model output message format. The assigner model receives the value at 00:00:03:000, and it adds the value to the assigner message and sends it to the data storage and let us assume that the data stored in the storage has a timestamp of 2021-06-08T11:44:30Z (using the timestamp format explained previously).

```
00:00:06:000
…
[ControlSystem_inputreader_status_defs::out: {1}] generated by model InputFromCS1_status
…
00:00:06:000
…
[SL_Communicator_defs::actuator_data_out: {}, SL_Communicator_defs::sensor_data_out: {},
SL_Communicator_defs::command_line_tx: {1 1}] generated by model SL_Communicator1
…
00:00:23:000
…
[ControlSystem_inputreader_typeandtime_defs::out: {Humidity 2021-06-28T11:44:30Z}]
generated by model InputFromCS1_typeandtime
…
00:00:23:000
…
[SL_Communicator_defs::actuator_data_out: {}, SL_Communicator_defs::sensor_data_out:
{Humidity 75.3 RH}, SL_Communicator_defs::command_line_tx: {}] generated by model
SL_Communicator1
…
```

Figure 27. Simulation output for dummy control system communication

As shown in Figure 27, at timestamp 00:00:06:000 the control system sends out a status check command. As soon as the communicator model receives it, it reverts with the status check the response and the type and time request i.e., {1 1}. As the responses are sent, the communicator waits for the control system to respond. At time stamp 00:00:23:000, the communicator model receives the type of the sensor and the time stamp for which it needs the data. The communicator model finds the required data in the data storage and provides it to the control system at the same time. This communication remains the same regardless of the number of sensors used and the types of sensors.

For multiple sensors of either the same type or multiple types, we use the fusion model to fuse the sensor data and discard the faulty sensors. Figure 28 shows the output cadmium logs if we simulate 8 humidity sensors.

```
00:00:00:000
[CPL_Sensor_defs::out: {Hum_Sens1 75.3}] generated by model CPL_Sensor1
[CPL_Sensor_defs::out: {Hum_Sens2 79.6}] generated by model CPL_Sensor2
[CPL_Sensor_defs::out: {Hum_Sens3 70.9}] generated by model CPL_Sensor3
[CPL_Sensor_defs::out: {Hum_Sens4 78.4}] generated by model CPL_Sensor4
[CPL_Sensor_defs::out: {Hum_Sens5 73.7}] generated by model CPL_Sensor5
[CPL_Sensor_defs::out: {Hum_Sens6 75}] generated by model CPL_Sensor6
[CPL_Sensor_defs::out: {Hum_Sens7 72.5}] generated by model CPL_Sensor7
[CPL_Sensor_defs::out: {Hum_Sens8 76.2}] generated by model CPL_Sensor8
…
00:00:02:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[IL_Sorter_defs::out: {1 Humidity 75.3, 2 Humidity 79.6, 3 Humidity 70.9, 4 Humidity
78.4, 5 Humidity 73.7, 6 Humidity 75, 7 Humidity 72.5, 8 Humidity 76.2 }] generated by
model IL_Sorter1
…
00:00:03:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[] generated by model IL_Sorter1
[IL_Fusion_defs::out: {Humidity 76.4283 }] generated by model IL_Fusion1
…
00:00:04:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[] generated by model IL_Sorter1
[] generated by model IL_Fusion1
[IL_Assigner_defs::out: {Humidity 76.4283 RH }] generated by model IL_Assigner1
…
```

Figure 28. Simulation output for multiple sensors of the same type in the system

The sensor models generate the humidity outputs that are given to the sorter model. The sorter model properly sorts the values, and numberings are provided. The output from the fusion model can be seen at time stamp 00:00:03:000 where it outputs one fused value for the 8 inputs. Lastly, the assigner model assigns the units and stores the data in the data storage.

During the simulation of the multiple types of multiple sensors, the sorter models sort the random data into groups, and hence the further processes are done. Figure 29 shows the output logs generated by the simulator when we simulate multiple types of sensors of

multiple types are connected. Here we use humidity and temperature as different types and 4 sensors each. We can see that the inputs arrive randomly at the DCIF, but the sorter model sorts the inputs according to the name and number. The fusion is performed for both types of sensor values independently but at the same time. The assigner checks for the type of the sensor according to the name and adds the units to the message at the following time stamp.

```
00:00:00:000
[CPL_Sensor_defs::out: {Hum_Sens1 75.3}] generated by model CPL_Sensor1
[CPL_Sensor_defs::out: {Hum_Sens4 78.4}] generated by model CPL_Sensor2
[CPL_Sensor_defs::out: {Hum_Sens3 70.9}] generated by model CPL_Sensor3
[CPL_Sensor_defs::out: {Hum_Sens2 79.6}] generated by model CPL_Sensor4
[CPL_Sensor_defs::out: {Temp_Sens3 23.5}] generated by model CPL_Sensor5
[CPL_Sensor_defs::out: {Temp_Sens2 24}] generated by model CPL_Sensor6
[CPL_Sensor_defs::out: {Temp_Sens1 23.2}] generated by model CPL_Sensor7
[CPL_Sensor_defs::out: {Temp_Sens4 23.8}] generated by model CPL_Sensor8
…
00:00:02:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[IL_Sorter_defs::out: {1 Humidity 75.3, 2 Humidity 79.6, 3 Humidity 70.9, 4 Humidity
78.4, 1 Temperature 23.2, 2 Temperature 24, 3 Temperature 23.5, 4 Temperature 23.8 }]
generated by model IL_Sorter1
…
00:00:03:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[] generated by model IL_Sorter1
[IL_Fusion_defs::out: {Humidity 74.8037, Temperature 23.7695 }] generated by model
IL_Fusion1
…
00:00:04:000
[] generated by model CPL_Sensor1
…
[] generated by model CPL_Sensor8
[] generated by model IL_Sorter1
[] generated by model IL_Fusion1
[IL_Assigner_defs::out: {Humidity 74.8037 RH, Temperature 23.7695 °C/°F }] generated by
model IL_Assigner1
…
```

Figure 29. Simulation Output for multiple sensors of multiple types in the system

In the real-time hardware deployment of the DCIF, we followed the same approach as we did during the simulation. We did experimentation with one sensor of a type, multiple temperature sensors, and multiple temperature and humidity sensors working at the same time.

The deployment of the DCIF for one sensor of a type was fairly simple and there was nothing different than that of what we have seen in the simulation. For the multiple temperature sensors deployment, we used 5 temperature sensors connected to the microcontroller NUCLEO development board. Initially, we used a NUCLEO F401RE development board, which is much lesser powerful than that of H743ZI but uses very low power. This board uses the same architecture and hence, the deployment does not make any difference. When needed (power, memory constraints), we used the NUCLEO H743ZI. Figure 30 shows the photos of the initial testing of the DCIF model with 5 temperature sensors.

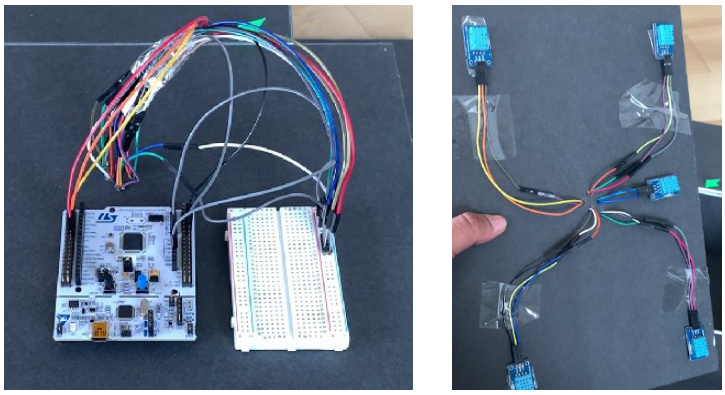

Figure 30. DCIF model on NUCLEO F401RE hardware and 5 temperature and humidity sensors

In Figure 30, the picture on the left shows the connections of the sensors to NUCLEO F401RE and the picture on the right shows the 5 temperature sensors arrangement. We generated data by polling the sensors at the rate of 30 seconds, 1 minute, 5 minutes, etc. However, for the building applications like this one, a polling rate of 30 seconds is too frequent, and we have used a polling rate of 1 minute and 5 minutes for the testing. Figure 31 shows the data storage file that was created by the hardware output on the computer console. This file was generated when the model with 5 temperature sensors was capturing

the real-time data at a polling rate of 5 minutes. The values stored in the data storage are the fused value of all the temperature sensors.

```
2021-08-11T23:16:08Z,Temperature,27.7695,°C
2021-08-11T23:21:12Z,Temperature,27.7523,°C
2021-08-11T23:26:14Z,Temperature,27.6544,°C
2021-08-11T23:31:16Z,Temperature,27.7522,°C
2021-08-11T23:36:20Z,Temperature,27.7766,°C
2021-08-11T23:41:24Z,Temperature,27.7854,°C
2021-08-11T23:46:26Z,Temperature,27.7121,°C
2021-08-11T23:51:22Z,Temperature,27.6870,°C
 2021-08-11T23:56:24Z,Temperature,27.5,°C
2021-08-11T23:16:28Z,Temperature,27.6534,°C
2021-08-11T00:01:28Z,Temperature,27.7443,°C
2021-08-11T00:06:26Z,Temperature,27.7553,°C
2021-08-11T00:11:25Z,Temperature,27.7234,°C
2021-08-11T00:16:28Z,Temperature,27.7432,°C
```

Figure 31. Hardware generated DataStorage.csv for multiple sensors of the same type

The DHT11 sensors are temperature and humidity sensors. That is, it can also sense the humidity along with the temperature. Similar to that of the previous experimentation with multiple temperature sensors, we also added the functionality to sense the ambient humidity to the model. The data stored in the data storage by the hardware is shown in Figure 32 where it can be seen that along with the temperature values shown in Figure 31, the humidity values were also stored.

```
2021-08-11T23:16:08Z,Temperature,27.25,°C
  2021-08-11T23:16:08Z,Humidity,65.5,RH
2021-08-11T23:21:12Z,Temperature,27.3,°C
   2021-08-11T23:21:12Z,Humidity,65,RH
2021-08-11T23:26:14Z,Temperature,27.5,°C
  2021-08-11T23:26:14Z,Humidity,65.3,RH
2021-08-11T23:31:16Z,Temperature,27.45,°C
  2021-08-11T23:31:16Z,Humidity,65.5,RH
2021-08-11T23:36:20Z,Temperature,27.24,°C
  2021-08-11T23:36:20Z,Humidity,65.3,RH
2021-08-11T23:41:24Z,Temperature,27.37,°C
  2021-08-11T23:41:24Z,Humidity,65.2,RH
2021-08-11T23:46:26Z,Temperature,27.33,°C
   2021-08-11T23:46:26Z,Humidity,65,RH
```

Figure 32. Hardware generated DataStorage.csv for multiple sensors of different types

Further, we printed the output of all the 5 sensors data to the console and processed the file generated to do a graph-based analysis. The generated graphs are shown and explained later.

We conducted a more exhaustive experiment where we connected 5 DHT11 sensors to determine the temperature and humidity in the environment with the NUCLEO-H743ZI development board. The frequency of the measurement, i.e., the polling rate was 1 minute. The period of the experiment was 24 hours. There were no external sources to affect the temperature and humidity readings drastically and the locations of the sensors were fixed. The faulty scenarios were created manually, and the results were analyzed. For instance, we disconnected one of the sensors to see if the system remains stable. Once we obtained the results, we plotted graphs. Figure 33 and Figure 34 show the graphs of temperature and humidity, respectively.

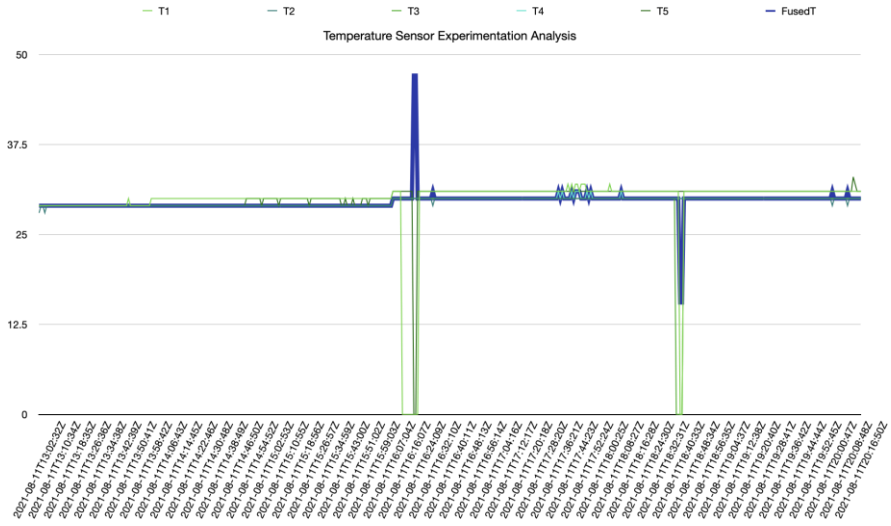

Figure 33. Graph representing the temperature sensor experimentation results

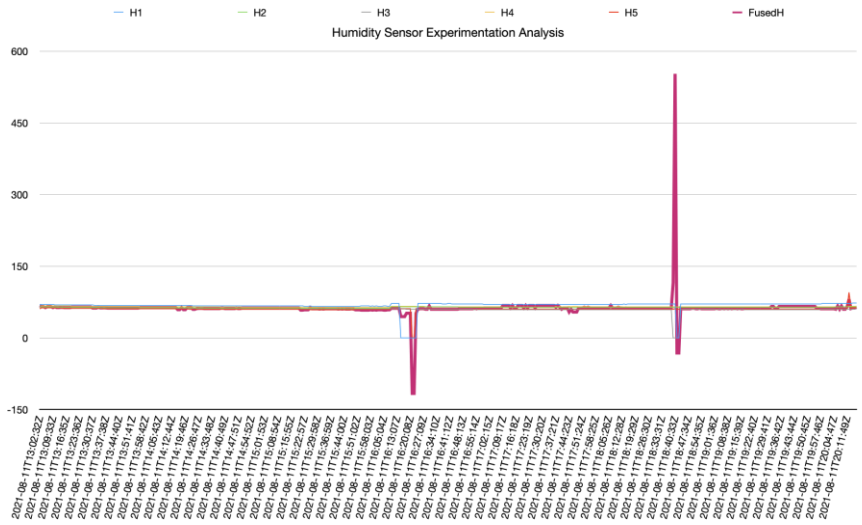

Figure 34. Graph representing the Humidity sensor experimentation results

We can see that for the most part the system remains stable. But there were occasional glitches in the output as seen in the graphs as the spikes. These spikes represented a hard fault situation.

The hard faults are the faults in the system that is responsible for the system to fail. For instance, in Figure 34 at the spike location, we can see that one of the sensors was disconnected (hence the 0 value for that sensor) but since our system should remain stable as the sensor fusion algorithm we use works even if we disconnect 1/3 sensors according to the criterion. These are the rare cases where the system fails regardless and hence they are known as hard faults, and this can create problems with the decision-making of the control system. Figure 35 shows the hard faults situations in the system.

2021-08-11T16:15:07Z	0	30	30	30	31	30	0	65	61	66	61	44.5406	TRUE
2021-08-11T16:16:07Z	0	30	30	30	31	30	0	65	61	66	61	44.5406	TRUE
2021-08-11T16:17:07Z	0	30	30	30	31	30	0	65	61	66	61	44.5406	TRUE
2021-08-11T16:18:08Z	0	30	30	30	31	30	0	65	60	66	61	51.6642	TRUE
2021-08-11T16:19:08Z	0	30	30	30	31	30	0	65	60	66	61	51.6642	TRUE
2021-08-11T16:20:08Z	0	30	30	30	31	30	0	65	60	66	61	51.6642	TRUE

Figure 35. Hard Fault Situations

Column 1 represents the timestamp, 2-6 represents the 5 temperature values obtained, 7 represents the fused temperature data, 8-12 represents the 5 humidity values obtained, 13 represents the fused humidity value and the last column represents the hard fault situations (TRUE if present). These values were obtained by using DCIF. As we can see all the temperature values are around 30 degrees, except sensor 1 since it is disconnected, and hence the sensor fusion algorithm determines the fused value to be 30 degrees. However, for the humidity values, we see that there are a lot of variations. For instance, two sensor values are around 61 %RH and the other two are around 65-66%RH. Sensor 1 is disconnected here as well. The fused value here comes out to be realistically incorrect. This is due to the sensor fusion algorithm used in the DCIF calculating the integrated support degree score, i.e., it checks if the sensors support each other. In this case, sensor 2 and sensor 4 are supported by each other and not supported by the other three. And sensor 3 and sensor 5 are supported by each other and not supported by the other three. Sensor 1 is not being supported by any other sensor. This forms a conflict situation and hence in this case the algorithm fails to understand which sensor denotes the correct value. This is a hard fault and hence the system becomes unstable. This can be fixed by implementing another layer of security to the framework that can resolve the conflict situations but is not covered in this thesis.

Another issue faced during the implementation of the real-time hardware is that the GSL library could not be used for the deployment since it cannot be compiled for ARM Cortex M processors. Hence, we use the Eigen library even though GSL produced good results due to a higher degree of floating-point value calculations.

4.4 Case Study: Building Control Application

After the real-time implementation and the simulation of the DCIF model, we carried out a case study that applies the DCIF model to a real-life application. We implemented a prototype application using a digital twin of Carleton University campus, the Carleton Digital Campus, which was built for virtual experience and the building performance analysis. In the implementation of the digital twin, we use DCIF to implement and execute the sensor network in the buildings. Since we know that the data reliability of the sensors in the building management is an important aspect of building maintenance. DCIF provides that reliability layer and hence it also provides a framework for implementing the sensors network and providing data to the BIM in the Carleton Digital Campus.

Carleton University comprises more than fifty buildings linked by five kilometers of underground tunnels and a network of roads and pathways. The Carleton Digital Campus includes a BIM of Carleton University that permits visualization of the complex data sets. Building information modeling (or BIM) is a process supported by various tools, technologies, and contracts involving the generation and management of digital representations of physical and functional characteristics of places. The DCIF provides the sensor data as it requires and hence helping the visualization of the data on the BIM. There are several types of data that can be used to evaluate the operation and performance of the buildings that includes simulated data, experimental data, historical data, or real-time data. The DCIF provides all these types of data by simulating the model as we discussed earlier or through experimentations we discussed previously. The simulated data is used to study various scenarios and to simulate the experimental conditions while the experimental data is directly obtained from measurements performed on the physical model of the building

in which real-world scenarios are replicated. We perform both, simulation and experimentation, in this case study where DCIF is being used.

The Carleton Digital Campus model is an accurate digital representation of the physical campus, created over an 8-year period using heterogeneous data sets. These digital assets were translated and incorporated into a single federated model in BIM software (Autodesk Revit).

Figure 36 shows the BIM of a building we use in this case study i.e., VSIM building. VSIM BIM is a replica of the actual building with all the building elements (windows, doors, desks, etc.) and building systems (sensors, HVAC, etc.).

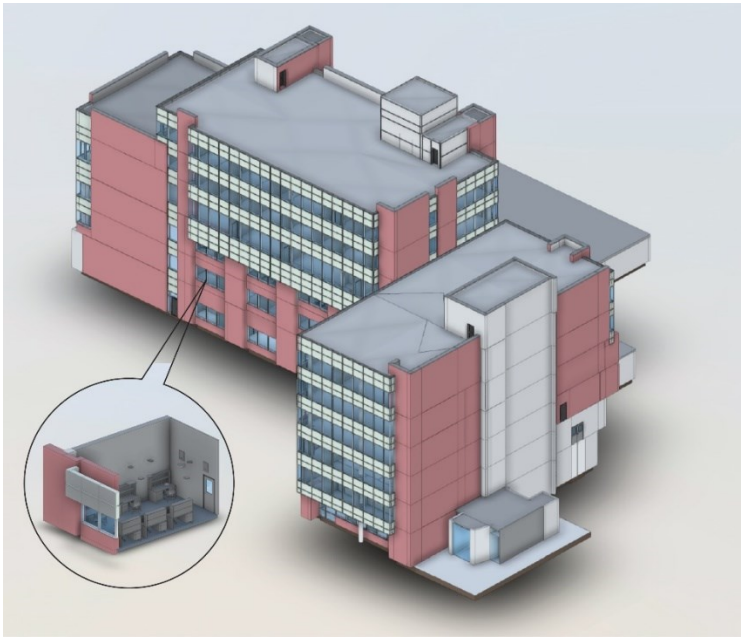

Figure 36. VSIM building model

We conducted simulation studies with different temperature readings within the measuring range using DCIF. To evaluate DCIF, the faults were deliberately injected in both simulation and experimentation using different scenarios we mentioned in the real-time implementation. The BIM of a lab in the VSIM building was used the execute the

experimentations. And hence, the lab was fabricated at a scale of 1:20 using laser cutting on acrylic sheets and adding control hardware as shown in Figure 37.

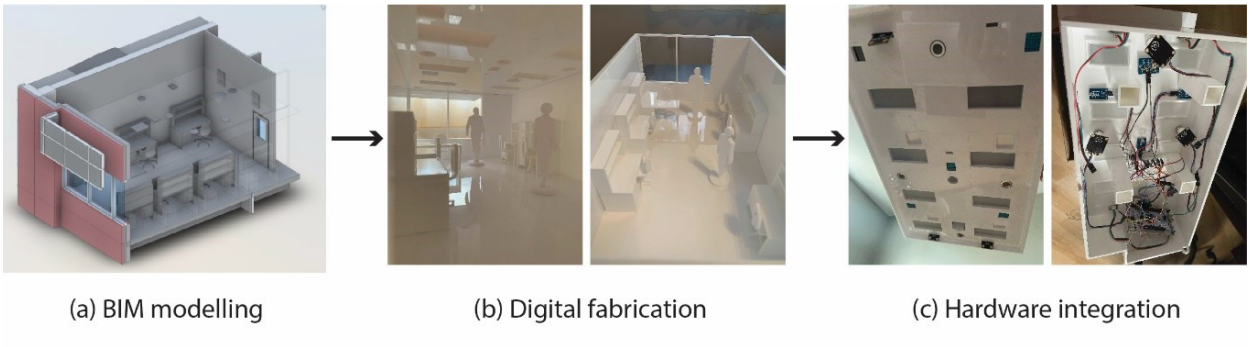

Figure 37. Physical Model for experimentation

Figure 37(a) shows the BIM of the lab that was used to fabricate a maquette, (b) shows the fabricated model using automated laser cutting from the BIM and (c) shows the hardware integration in the physical model. The computing hardware we used for the case study was an STM32 based Nucleo-144 microcontroller (NUCLEO-H743ZI). Multiple sensors were used: DHT11 (Temperature and Humidity Sensors) connected through digital input pins as discussed earlier, MG811 (CO2 sensors) connected through analog input pins, VEML7700 (Ambient Light Sensors) connected through the I2C protocol and PIR Motion Detection Sensors connected through a digital input pin. The composition of the physical model can be seen in Figure 38.

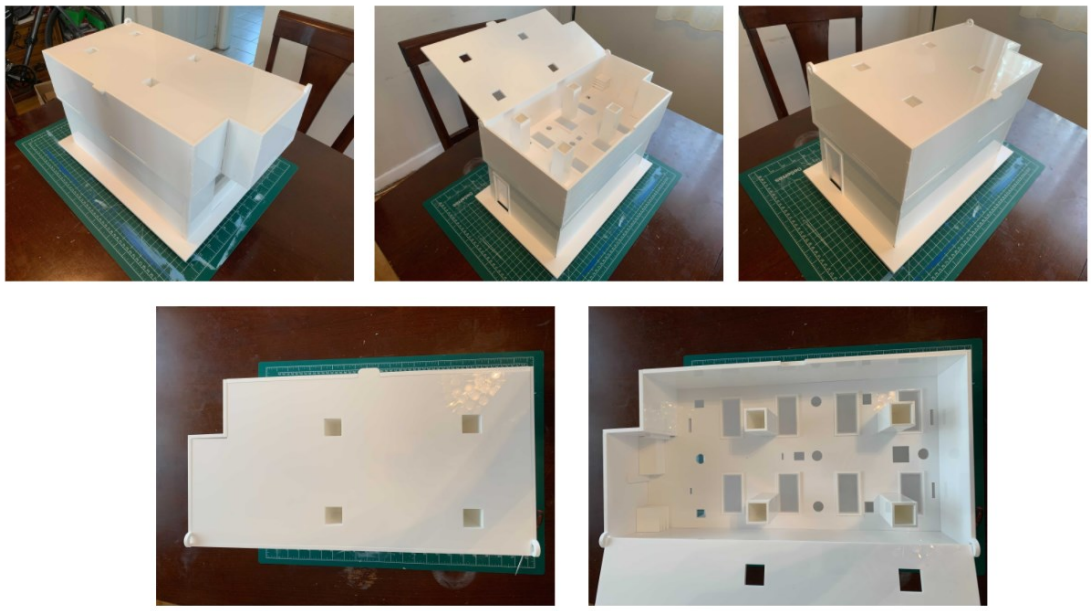

Figure 38. Physical model of the VSIM building room 360 view

The sensors were mounted to the physical model at specific locations on the top to mimic the actual building room. Figure 39 and Figure 40 show the mounting of the sensor on the physical model.

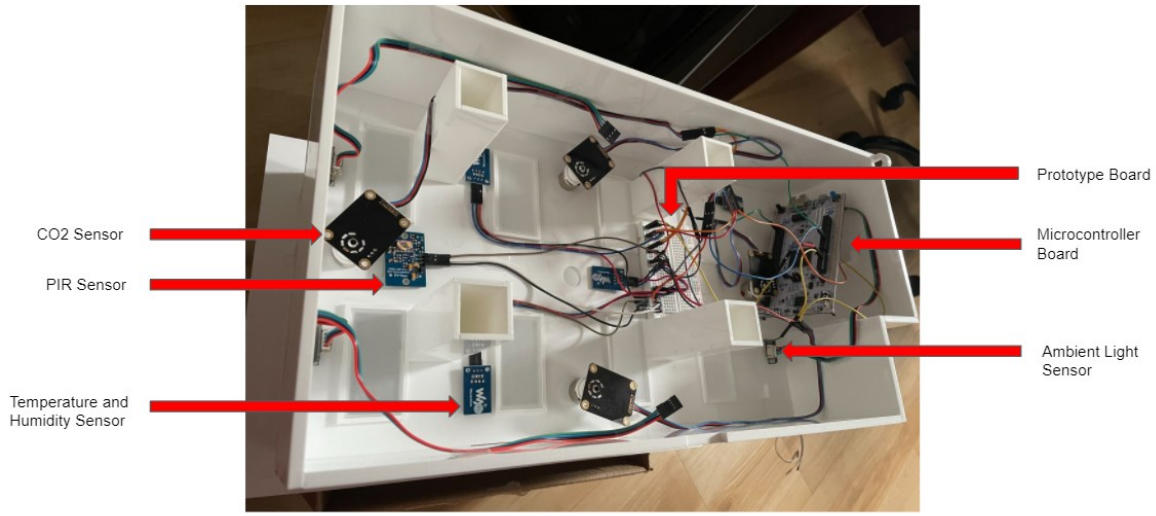

Figure 39. Top View of the roof of the room in the physical model

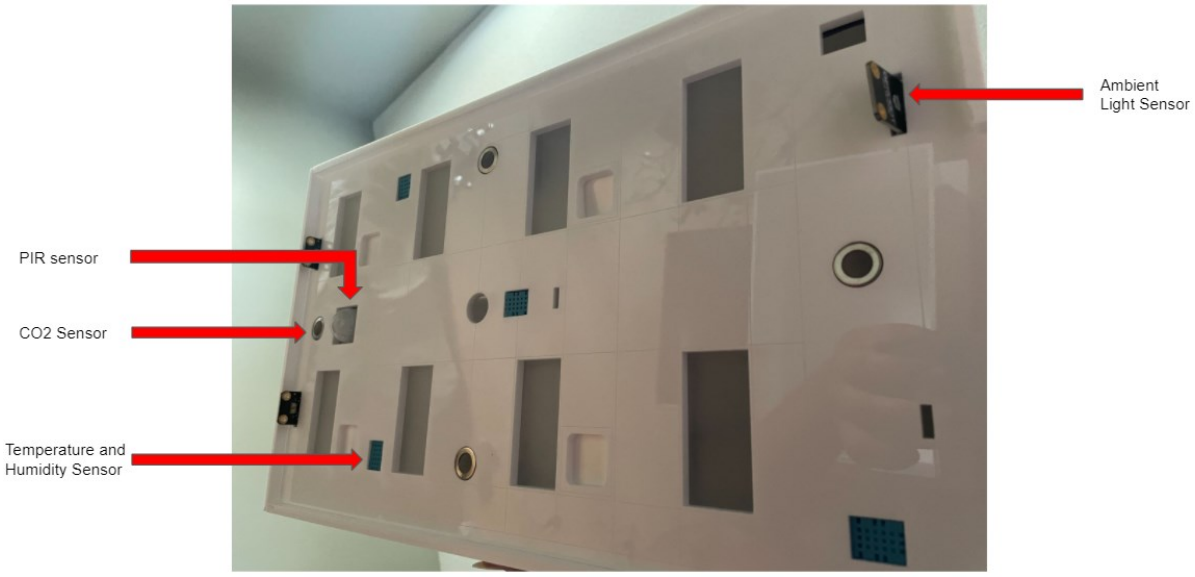

Figure 40. Bottom View of the roof of the room in the physical model

There were 5 temperature and humidity sensors mounted with 4 being on the four corners of the room and one being in the center of the room. There were 5 CO2 sensors mounted on 5 different places as shown in Figure 40. There were 4 ambient light sensors used as shown in the figure and one PIR sensor to check if there is a movement from the door when someone enters or exits the room.

The experimentation included the collection of data from the sensors using the DCIF and storing it in the data storage. The data storage was later sent to the cloud server and hence the data was visualized on the BIM. The sensor readings were perceived by the sensors at the polling rate of 5 minutes. That is, every 5 minutes, a new set of data was fetched from the sensors and hence stored in the data storage. The sensor output from the DCIF was then compared with the manually calculated average room temperature from the sensor readings and the comparison was studied.

The data collected by using DCIF and stored in the storage was used to display it onto a BIM using cloud services. The data storage was provided as an input to the Autodesk Forge software. Figure 41 shows the integration of the data collected from the sensors along with the fused value and manually calculated average value.

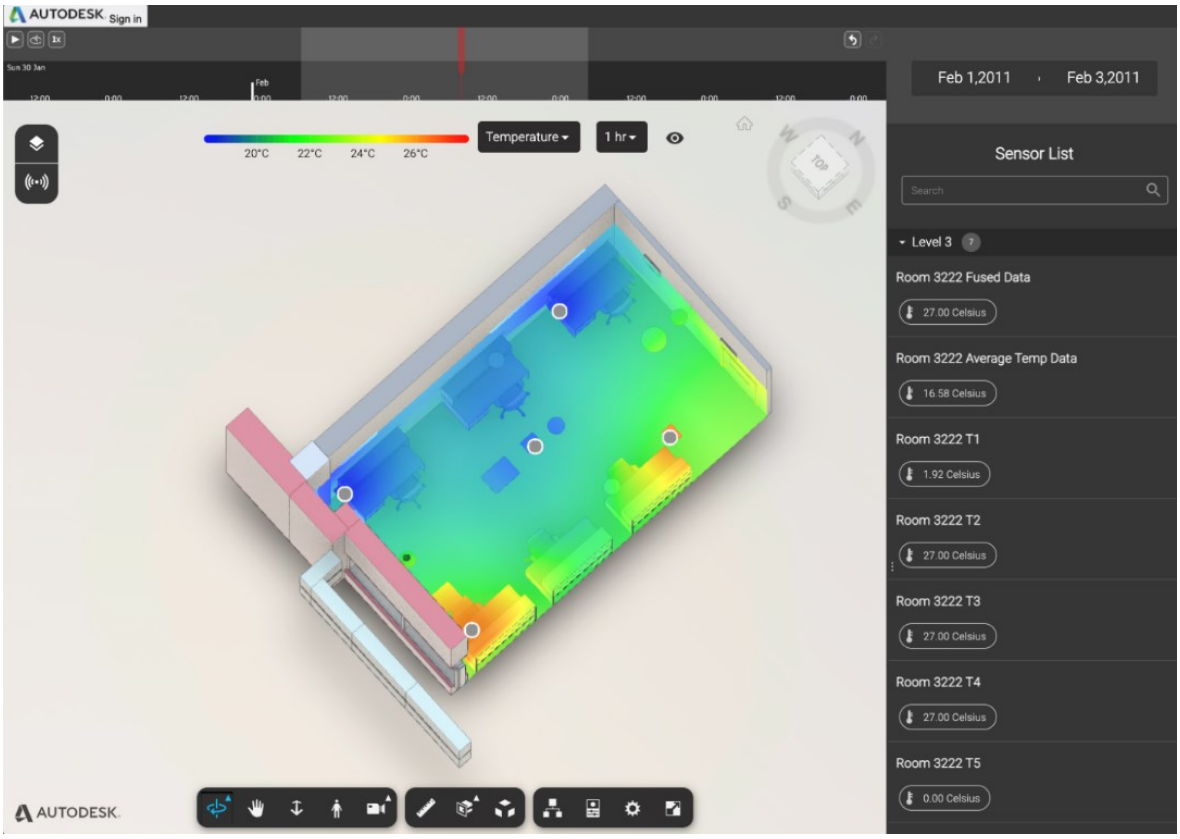

Figure 41. Integration of sensor data with BIM using Autodesk Forge

For the experimentation, the ideal room temperature at the given point in time was 27 degrees Celsius. As shown in the figure, the values T1, T2, T3, T4, and T5 denote the temperature values at a timestamp in Celsius. At the same timestamp, fused data is the data obtained by DCIF after processing it. The average data is the data that shows the average value of the temperature values at the timestamp. As mentioned previously, the faults were manually injected into the system, and hence we can see that the temperature T1 and T5

are providing 1.92 and 0.00 degrees Celsius, i.e., an erroneous value since they were tempered in the experimentation for some time to provide real-time errors injection. Regardless, the DCIF provided a consistent value of the fused data that is 27 degrees Celsius since the model uses a sensor fusion algorithm to provide a layer of security in case of failures. However, the average value calculated manually can be seen to be giving a value of 16.58 degrees Celsius which is not a correct value and if we did not implement a good sensor fusion algorithm in DCIF, the system would fail. The average temperature has many variations due to faulty readings, but the fused value obtained from the DCIF provided a steadier temperature value by finding out the integrated support degree score for all the sensors implemented. Figure 42 shows a comparison between the DCIF output to the manually calculated average of the sensors. It can be seen that the DCIF output provided very promising results.

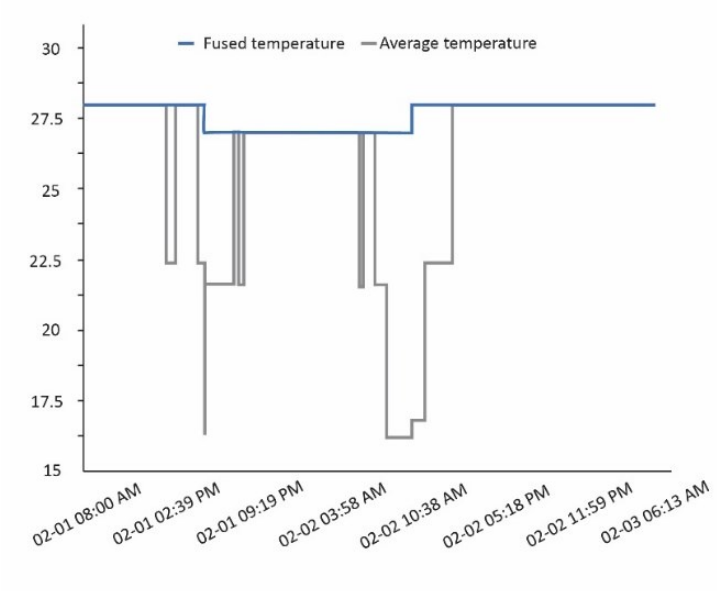

Figure 42. Comparison of DCIF Fused temperature output to the average temperature

The DEVS CPS Interface Framework helped the BIM software obtain the data. We used DCIF to poll the data from the sensors, store it in the data storage in a presentable format and provide it to the BIM for visualization. We could see that the visualization of the data was very intuitive and easy to understand which could help the researchers in the field of building modeling.

Chapter 5: Conclusion and Future Work

A cyber-physical system consists of the physical world, interfaces, and cyber-world. There is an intense link between the cyber and physical worlds and the interfaces in CPS are responsible to provide this important link. The interface is a fundamental characteristic of a CPS and without the interfacing, the CPS cannot provide any functionality. A variety of research shows that there is a huge development on the sensor, actuators, and embedded computational systems but it is equally important to study the interfaces of the cyber-physical systems.

We found that there is a strong need to evaluate the working of the CPS interfaces and standardize them. But due to CPS complexity, it is unsuitable to use traditional methodologies where the system is developed at a low level directly on the hardware. However, modeling and simulation (M&S) techniques allow researchers to model systems and perform simulations to evaluate the performance of the proposed design. This process reduces the cost and time required. Discrete Event System Specification (DEVS) is a general and formal framework for modeling and simulation that allows the researchers to model complex systems without worrying about the implementation details of the simulator.

In this dissertation, we formally defined an architecture for the CPS interface in the DEVS M&S technique. We called it DEVS CPS Interface Framework (DCIF). The developed DCIF model portrayed the complete working of the CPS interface. To support the cyber communication in the model, we used message-based transactions. The structure of the messages used for these transactions was also formalized in this thesis. The model was

designed to be extensible, flexible, and easily adaptable for simulation and real-time implementation on target embedded hardware.

The DCIF consists of three layers: Cyber-physical, Informational, and Service. We defined the behavior of these three layers in DEVS and portrayed their execution using multiple simulations in a simulation environment using the Cadmium simulation tool. Once the model was simulated and tested in the simulation environment, we implemented it on the target embedded hardware using the extension of the Cadmium – RT-Cadmium. In addition, we provided an easy way for the implementation of the sensor fusion algorithms in the interface framework. We used a principal component analysis-based competitive sensor fusion algorithm that provides a layer of security and reliability.

Additionally, we verified the proposed architecture of the DCIF by implementing a practical application that adapts this architecture. We also illustrated a prototype application using a digital twin of the Carleton University campus which has been built for virtual experience and building performance analysis where we use DCIF for data acquisition and storage. DCIF application in the field of building information modeling contributed towards enhancing the measured data reliability through sensor fusion, data storage and analysis, and data visualization on BIM.

The proposed framework DCIF in this thesis is completely time-synchronous. This framework perfectly fits the need of the non-interrupt-based sensors by using the polling method. But managing asynchronous inputs to this model is not defined and hence any interrupt-based inputs are not taken into consideration in this thesis. However, in the future, interrupt-based inputs can be added to this framework by adding an additional asynchronous event handler.

In addition to that, in this thesis, although we implement a sensor fusion model in the framework to improve the reliability, we discussed some hard fault situations that make the system unstable. Some competitive algorithms form conflict situations due to ambiguous sensor inputs. To mitigate this issue, in the future, an implementation of an additional layer of security that supports the sensor fusion can be done.

Appendices

Appendix A

A.1 Model implementation

This appendix includes the important code snippets from the implementation of the DCIF.

```
using input_ports=std::tuple<typename defs::inputFromIL>;
using output_ports=std::tuple<typename defs::out>;

void internal_transition() {
  state.active = false;
}

void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs)    {
  for(const auto &x : get_messages<typename defs::inputFromIL>(mbs)) {
    state.Name = x.name;
    state.Value = x.value;
   }
   state.active = true;
 }

void confluence_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
   internal_transition();
   external_transition(TIME(), std::move(mbs));
 }

typename make_message_bags<output_ports>::type output() const {
   typename make_message_bags<output_ports>::type bags;
   get_messages<typename defs::out>(bags).push_back(state.Value);
   return bags;
 }

TIME time_advance() const {
   if(state.active) {
      return TIME("00:00:00");
   }
   return std::numeric_limits<TIME>::infinity();
 }
```

Figure 43. Actuator Model Implementation

Figure 43 shows the implementation of the actuator model. The external transition function breaks down the message from the port *inputFromIL*. The name of the actuator is stored in the *Name* variable defined in the state structure and the value of the actuator is stored in the *Value* variable defined in the state structure. The state of the model changes to active. The *Value* variable is given to the output port *out*. In the output function, if we implement

the model for hardware deployment, we call the hardware driver functions with the variable *Value* as the parameter. When the state of the model is active, there is no delay in providing the output and hence the output is instantaneous i.e., the time advance is 0. If the model is passive, then the model passivates with the time advance changing to *infinity*.

```
struct Actuator_Message{
  Actuator_Message(){}
  Actuator_Message(string i_name, double i_value)
   :name(i_name), value(i_value){}

    string   name;
    double  value;
};

istream& operator>> (istream& is, Actuator_Message& msg);

ostream& operator<<(ostream& os, const Actuator_Message& msg);
```

Figure 44. Actuator Message Structure

Figure 44 shows the actuator message structure. The actuator message consists of a string for the actuator name and a float/double for the actuator value.

```
using input_ports=std::tuple<typename defs::s1, typename defs::s2, typename defs::s3,
typename defs::s4, typename defs::s5, typename defs::s6, typename defs::s7, typename
defs::s8>;
using output_ports=std::tuple<typename defs::out>;

   void internal_transition (){
    state.active = false;
   }

   void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs){
    state.send_to_fusion.message.clear();
    state.send_to_fusion.message.shrink_to_fit();
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s1>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s2>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s3>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s4>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s5>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s6>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s7>(mbs)[0]);
    state.values_from_sensors.push_back(get_messages<typename
IL_Sorter_defs::s8>(mbs)[0]);

    vector<vector<Sensor_Message>> rearranged_sensor_message;
    sort(state.values_from_sensors.begin(), state.values_from_sensors.end(), compare);
    rearranged_sensor_message = rearrange(state.values_from_sensors);

    state.send_to_fusion = add_values_to_sorter_message(rearranged_sensor_message);
    rearranged_sensor_message.clear();
```

```
    rearranged_sensor_message.shrink_to_fit();
    state.values_from_sensors.clear();
    state.values_from_sensors.shrink_to_fit();
    state.active = true;
  }

   void confluence_transition(TIME e, typename make_message_bags<input_ports>::type mbs)
{
  internal_transition();
  external_transition(TIME(), std::move(mbs));
   }

   typename make_message_bags<output_ports>::type output() const {
  typename make_message_bags<output_ports>::type bags;
    get_messages<typename defs::out>(bags).push_back(state.send_to_fusion);
  return bags;
   }

   TIME time_advance() const {
  if(state.active) {
    return TIME("00:00:02");
  }
  return std::numeric_limits<TIME>::infinity();

   }
```

Figure 45. Sorter Model implementation

Figure 45 shows the implementation of the sorter model. The external transition function stores the input from the input port in a vector of type *Sensor_Message.* A sorting operation is performed that sorts the sensor messages alphabetically. Once the sorting is performed, the *rearrange()* function is called with the sorted vector as the parameter. The *rearrange()* function groups the similar types of sensor readings and returns a vector of a vector of type *Sensor_Message* which is stored in the variable *rearranged_sensor_message.* The function *add_values_to_sorter_message()* is called which converts the sensor messages to sorter message and returns the sorted message of type *Sorter_Message.* This message is sent to the output port *out.*

```
Struct Sorter_Message{
  Sorter_Message(){}
  Sorter_Message(int i_number, string i_sensor_type, double i_value)
   :number(i_number), sensor_type(i_sensor_type), value(i_value){}

    int number;
    string   sensor_type;
    double  value;
};

struct Vector_Vector_Sorter_Message{
  Vector_Vector_Sorter_Message() {}
```

```
  Vector_Vector_Sorter_Message(vector<vector<Sorter_Message>> i_message) :
message(i_message) {}

  vector<vector<Sorter_Message>> message;
};

istream& operator>> (istream& is, Sorter_Message& msg);

ostream& operator<<(ostream& os, const Sorter_Message& msg);

ostream& operator<<(ostream& os, const Vector_Vector_Sorter_Message& msg);
```

Figure 46. Sorter Message Structure

Figure 46 shows the sorter message structure. It consists of an integer for the number of the current sensor, a string for the type of sensor and a float/double for sensor value. We create a vector of the sorter message to facilitate the storage of multiple sensor data.

```
Vector<vector<Sensor_Message>> rearrange(vector<Sensor_Message>& values_from_sensor) {
    string tempstring;
  vector<vector<Sensor_Message>> temp1;
  vector<Sensor_Message> temp = values_from_sensor;
  vector<Sensor_Message> temp2;
  int counter = 0;
  tempstring = values_from_sensor[0].name;
  tempstring.resize(3);
  for(auto i=0;i<temp.size();++i)
  {
  if(tempstring == temp[i].name.substr(0,3)){
      temp2.push_back(temp[i]);
  }
  else {
      counter++;
      tempstring = values_from_sensor[i].name;
      tempstring.resize(3);
      --I;
      }
  if(counter>0) {
      temp1.push_back(temp2);
      temp2.clear();
      counter = 0;
  }
  }
  if(counter == 0) {
      temp1.push_back(temp2);
      temp2.clear();
  }
  return temp1;
}
```

Figure 47. Rearrange Function for Sorter Model

Figure 47 shows the rearrange function in the sorter model. The *rearrange*() function takes the vector of type *Sensor_Message* as an argument and returns a vector of vectors of type *Sensor_Message.* It loops through the *values_from_sensor* vector. It compares *name* variable of the element in the *values_from_sensor* vector that denotes the sensor name to

the initial string value of the *name* variable. If they are the same, it stores the complete element that contains the *name* and *value* in temporary variable *temp2* which is a vector of type *Sensor_Message*. This is carried out until the *name* variable changes to the next value. Once the *name* string variable is changed in the vector, the counter is incremented that tells the program that there is a change in the type of sensor according to the *name* variable and it then pushes the vector *temp2* to the vector of vectors *temp1*. The loop is run until all the elements in the vector *values_from_sensor* are covered.

```
Vector<vector<Sorter_Message>>
add_values_to_sorter_message(vector<vector<Sensor_Message>>& sensor_message) {
    vector<vector<Sorter_Message>> sorter_message;
    vector<Sorter_Message> temperature, humidity, co2, lux, pir;
    int tempCounter=1, HumCounter=1, CO2Counter=1, LuxCounter=1, PIRCounter=1;
    for(int I = 0; i<sensor_message.size();i++) {
  for (int j = 0; j<sensor_message[i].size();j++) {

      if(sensor_message[i][j].name.substr(0,3) == "Tem") {
       Sorter_Message temp(tempCounter, "Temperature", sensor_message[i][j].value);
       temperature.push_back(temp);
       tempCounter++;
      }
      if(sensor_message[i][j].name.substr(0,3) == "Hum") {
       Sorter_Message temp(HumCounter, "Humidity", sensor_message[i][j].value);
       humidity.push_back(temp);
       HumCounter++;
      }
      if(sensor_message[i][j].name.substr(0,3) == "CO2") {
       Sorter_Message temp(CO2Counter, "CarbonDioxide", sensor_message[i][j].value);
       co2.push_back(temp);
       CO2Counter++;
      }
      if(sensor_message[i][j].name.substr(0,3) == "Lux") {
       Sorter_Message temp(LuxCounter, "Luminosity", sensor_message[i][j].value);
       lux.push_back(temp);
       LuxCounter++;
      }
      if(sensor_message[i][j].name.substr(0,3) == "PIR") {
       Sorter_Message temp(PIRCounter, "PIR", sensor_message[i][j].value);
       pir.push_back(temp);
       PIRCounter++;
      }
  }
  if(tempCounter>1) {sorter_message.push_back(temperature); tempCounter = 1;}
  if(HumCounter>1) {sorter_message.push_back(humidity);HumCounter = 1;}
  if(CO2Counter>1) {sorter_message.push_back(co2);CO2Counter = 1;}
  if(LuxCounter>1) {sorter_message.push_back(lux);LuxCounter = 1;}
  if(PIRCounter>1) {sorter_message.push_back(pir);PIRCounter = 1;}
    }
    return sorter_message;
}
```

Figure 48. Create Sorter Message function for Sorter Model

Figure 48 shows the *add_values_to_sorter_message()* function. This function loops through the vector of vectors of type *Sensor*_Message, checks for the sensor type by running the conditional statements, and adds the elements to the *sorter_message* which is a vector of vectors of type *Sorter_Message.* This vector of vectors is then returned.

```
struct Fused_Message{
  Fused_Message(){}
  Fused_Message(string i_type, double i_value)
   :type(i_type), value(i_value){}

    string   type;
    double  value;
};

struct Vector_Fused_Message {
  Vector_Fused_Message() {}
  Vector_Fused_Message(vector<Fused_Message> i_message) : message(i_message) {}

  vector<Fused_Message> message;
};

istream& operator>> (istream& is, Fused_Message& msg);

ostream& operator<<(ostream& os, const Fused_Message& msg);

ostream& operator<<(ostream& os, const Vector_Fused_Message& msg);
```

Figure 49. Fused Message Structure

Figure 49 shows the sensor fusion model message structure. It consists of a string for the type of sensor and a float/double for sensor value. We create a vector of the sorter message to facilitate the storage of multiple sensor data.

```
Using input_ports=std::tuple<typename defs::in>;
using output_ports=std::tuple<typename defs::out>;

void internal_transition (){
  state.active = false;
}

void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs){
  state.from_fusion = get_messages<typename IL_Assigner_defs::in>(mbs)[0];
  state.outputData = add_values_to_assigner_message(state.from_fusion.message);
  state.from_fusion.message.clear();
  state.from_fusion.message.shrink_to_fit();
  state.active = true;
}

void confluence_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
  internal_transition();
  external_transition(TIME(), std::move(mbs));
}

typename make_message_bags<output_ports>::type output() const {
  typename make_message_bags<output_ports>::type bags;
  get_messages<typename defs::out>(bags).push_back(state.outputData);
```

```
  StoreData(state.outputData);
  return bags;
 }

 TIME time_advance() const {
   if(state.active) {
    return TIME("00:00:01");
   }
   return std::numeric_limits<TIME>::infinity();
 }
```

Figure 50. Assigner Model Implementation

Figure 50 shows the implementation of assigner model. The external transition function receives the message and stores in a variable *from_fusion*. The function *add_values_to_assigner_message()* is called with the temporary variable as the only parameter. The function *add_values_to_assigner_message()* creates a vector of type *Assigner_Message* and adds the received message from the sensor fusion model to it along with adding some more information like the units and returns it. The returned vector of type *Assigner_Message* is then sent to the output port *out* of the model. In addition to that, the vector is also stored in the data storage by calling the function *StoreData()*.

```
Struct Assigner_Message{
  Assigner_Message(){}
  Assigner_Message(string i_type, double i_value, string i_units)
   :type(i_type), value(i_value), units(i_units){}

    string   type;
    double  value;
    string units;
    //more variables can be defined here and used as UserDefinedDetails
};

struct Vector_Assigner_Message {
  Vector_Assigner_Message() {}
  Vector_Assigner_Message(vector<Assigner_Message> i_message) : message(i_message) {}

  vector<Assigner_Message> message;
};

ostream& operator<<(ostream& os, const Assigner_Message& msg);

ostream& operator<<(ostream& os, const Vector_Assigner_Message& msg);
```

Figure 51. Assigner Message Structure

Figure 51 shows the implementation of assigner message structure. It consists of a string for the type of sensor, a float/double for fused sensor value, a string for the units of the

sensor value, and any data type that can be introduced in the *Assigner_Message* for any user-defined details.

```
Vector<Assigner_Message> add_values_to_assigner_message(vector<Fused_Message>&
sensor_message) {
    vector<Assigner_Message> assigner_message;
  for(int I = 0; i<sensor_message.size();i++) {
   if(sensor_message[i].type == "Temperature") {
       Assigner_Message temp("Temperature", sensor_message[i].value, "°C/°F");
       assigner_message.push_back(temp);
      }
      if(sensor_message[i].type == "Humidity") {
      Assigner_Message temp("Humidity", sensor_message[i].value, "RF");
       assigner_message.push_back(temp);
      }
      if(sensor_message[i].type == "CarbonDioxide") {
       Assigner_Message temp("CarbonDioxide", sensor_message[i].value, "PPM");
       assigner_message.push_back(temp);
      }
      if(sensor_message[i].type == "Luminosity") {
       Assigner_Message temp("Luminosity", sensor_message[i].value, "lux");
       assigner_message.push_back(temp);
      }
      if(sensor_message[i].type == "PIR") {
       Assigner_Message temp("PIR", sensor_message[i].value, "state");
       assigner_message.push_back(temp);
      }
  }
  return assigner_message;
}
```

Figure 52. add_values_to_assigner_message() function implementation

The *add_values_to_assigner_message()* function is an important part of the assigner model implementation. The *add_values_to_assigner_message()* function takes the vector of type *Fused_Message* as a parameter and returns a vector of type *Assigner_Message* as shown in Figure 52. The function cycles through the argument vector and adds the units to it according to the sensor type by using the conditional statements and stores it in the vector of type *Assigner_Message*. Once we have the units added, the function returns *assigner_message* to the assigner model.

```
SDFileSystem sd(p5, p6, p7, p8, "sd");
mkdir("/sd/Data", 0777);
void StoreData(Vector_Assigner_Message message) {
          time_t seconds = time(NULL);
          FILE *fp = fopen("/sd/Data/DataStorage.csv", "w");
          if(fp == NULL) {
          error("Could not open file for write\n");
          }

   for(auto i=0;i<message.message.size();++i){
      char buf[40];
```

```
      strftime(buf,40, "%Y-%m-%dT%H:%M:%SZ,", localtime(&seconds));
      fprintf(fp, buf);
              fprintf(fp, message.message[i].type);
              fprintf(fp, ",");
              fprintf(fp, message.message[i].value);
              fprintf(fp, ",");
              fprintf(fp, message.message[i].units);
              fprintf(fp, "\n");
      }
fclose(fp);
}
```

Figure 53. Implementation of data storage using SD Card

We use the function *StoreData()* that is similar to the function implemented in the simulation but instead of the file stream output, we use the standard console output. In this method, we consider the connected computer to the NUCLEO board as the storage device, and hence, we store the data printed by the board to the computer. For the second method, the code snippet in the Figure 53 shows the implementation of the data storage using an SD Card or OTG storage with a file system. Line 1-2 are instantiated in the assigner model which initializes the connection of the SD card to the microcontroller using SPI communication protocol. We use *Mkdir()* function from *SDFileSystem* which creates a directory that holds the file for the SD card storage. We open the file and start adding the data to it by using the *fprintf()* function.

```
MatrixXd sdm_calculator(vector<double> sensorinputs, int size){
    int i,j,k=0;
    double *dmatrix = (double *) malloc(sizeof(double)*(size*size));
    for(i=0; i<size;i++){

        for(j=0;j<size;j++){
            double temp = sensorinputs[i]-sensorinputs[j];

            if(temp<0){
                temp = -temp;
            }

            temp = exp(-(temp));
            dmatrix[k]= temp;
            k++;

            if(k%size==0){
            }
        }
    }
    MatrixXd dmatrix1 = Map<MatrixXd>( dmatrix, size, size );
    return dmatrix1;
}
```

Figure 54. Support Degree Matrix Calculation

Figure 54 shows the implementation of the support degree matrix calculator implementation. The function *sdm_calculator()* takes the sensor readings and the number of sensors as a parameter and returns the support degree matrix. We use the TuxFamily Eigen library and the GNU GSL library to support the implementation. The Eigen and GSL libraries provide a template that makes it easier to implement more complicated operations like matrices, linear algebra, etc. For instance, We use the *MatrixXd* that defines a matrix with the provided size.

```
MatrixXd eigen_value_calculation(MatrixXd dmatrix){
    SelfAdjointEigenSolver<MatrixXd> solver(dmatrix);
    MatrixXd eigenvalues;
    eigenvalues = solver.eigenvalues().real();
    eigenvalues = eigenvalues.colwise().reverse().eval();
    return eigenvalues;
}
MatrixXd eigen_vector_calculation(MatrixXd dmatrix){
    SelfAdjointEigenSolver<MatrixXd> solver(dmatrix);
    MatrixXd eigenvectors;
    eigenvectors = solver.eigenvectors().real();
    eigenvectors = eigenvectors.rowwise().reverse().eval();
    return eigenvectors;
 }
```

Figure 55. Eigen Value and Vectors Calculation

Figure 55 shows the implementation of the eigenvalues and eigenvectors calculation implementation. The functions *eigen_value_calculation()* and *eigen_vector_calculation()* takes the support degree matrix returned by the *sdm_calculator()* function as a parameter and returns the matrix of eigenvalues and the corresponding eigenvectors. We use the Eigen library's *SelfAdjointEigenSolver* to get the eigenvalues and vectors. The imaginary values if produced, are not considered here and only the real values are used.

```
 MatrixXd compute_alpha(MatrixXd eigenvalues) {
    MatrixXd list_of_alphas;
    double sum_of_evals;
    sum_of_evals = eigenvalues.sum();
    list_of_alphas = (eigenvalues / sum_of_evals)*100;
    return list_of_alphas;
 }

MatrixXd compute_phi(MatrixXd list_of_alphas, int size){
    MatrixXd list_of_phi;
    list_of_phi = list_of_alphas;
    for(int i=1; i<size;i++){
```

```
        list_of_phi(i,0) = list_of_phi(i-1,0) + list_of_alphas(i,0);
    }
    return list_of_phi;
}
```

Figure 56. Contribution rate and accumulated contribution rate calculation

Figure 56 shows the implementation of the calculation of contribution rate and accumulated contribution rate. The *compute_alpha()* and *compute_phi()* functions defined calculates the contribution rate of the *k*th principal component and the accumulated contribution rate, respectively. The function *compute_alpha()* takes eigenvalues calculated by *eigen_value_calculation()* as the parameter and returns a matrix consisting of the list of alpha for all the principal components. The function *compute_phi()* calculates the accumulated contribution rate by using the list of alphas that is calculated using the function *compute_alpha()*.

A.2 Hardware Description

This section includes the description and features of the sensor hardware used.

The temperature and humidity sensors DHT11 were used, and the features are given below:

Operating Voltage: 3.3V-5V DC

The temperature sensing accuracy: ±2℃

Measuring range : 0°C ~ 50°C

The humidity sensing accuracy: ±5%RH (0~50°C)

Measuring range: 20%RH ~ 90%RH (25°C)

The CO2 sensors MG811 features are given below:

Operating voltage: 6V DC

Analog output: 0-2V

Measuring Range: 350 – 10000ppm CO2

The Ambient light sensor VEML7700 features are given below:

Operating Voltage: 3.3V-5V DC

Measuring Range: 0-120klx

References

A. M. Astorga, D. Moreno-Salinas, D. C. Garcia and J. A. Almansa. 2011. "Simulation benchmark for autonomous marine vehicles in LabView." *OCEANS 2011 IEEE.* Spain.

Abbas W, Laszka A, Koutsoukos X. 2016. "Resilient wireless sensor networks for cyber-physical systems." *Cyber-Physical System Design With Sensor Networking Technologies.* London.

al., A. Savvides et. 2011. "Cyber-physical systems for next generation intelligent buildings." *Proc. WiP Session ICCPS.*

al., L. Tang et. 2010. "Tru-alarm: Trustworthiness analysis of sensor networks in cyber-physical systems." *Proc. IEEE ICDM.*

Barreras, Jorge Varela, Christian Fleischer, Andreas Elkjaer Christensen, Maciej Swierczynski, Erik Schaltz, Soren Juhl Andreasen, and Dirk Uwe Sauer. 2016. "An Advanced HIL Simulation Battery Model for Battery Management System Testing." *IEEE Transactions on Industry Applications* 52 (6): 5086-5099.

Belloli, Laouen, Damian Vicino, Cristina Ruiz-Martin, and Gabriel Wainer. 2019. "Building DEVS models with the cadmium tool." *Proceedings of the 2019 Winter Simulation Conference.*

Bergero, Federico, and Ernesto Kofman. 2010. "PowerDEVS: a tool for hybrid system modeling and real-time simulation." *Simulation: Transactions of the Society of Modeling and Simulation International* 87 (12): 113-132.

Bin Lu, Xin Wu, H Figueroa, and A Monti. 2007. "A Low-Cost Real-Time Hardware-in-the-Loop Testing Approach of Power Electronics Controls." *IEEE transactions on industrial electronics* 919-931.

Boi-Ukeme, Joseph, and Gabriel Wainer. 2020. "A Framework for the Extension of DEVS With Sensor Fusion Capabilities." *Spring Simulation Conference (SpringSim).* 1-12.

Boi-Ukeme, Joseph, Cristina Ruiz-Martin, and Gabriel Wainer. 2020. "Real-Time Fault Detection and Diagnosis of CPS Faults in DEVS." *6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application.* IEEE. 57–64.

Boukerche, Azzedine, Ahmed Shadid, and Ming Zhang. 2007. "A Formal Approach to RT-RTI Design Using Real Time DEVS." *2007 IEEE International Workshop on Haptic, Audio and Visual Environments and Games, 2007-10.* IEEE. 84-89.

Bryde, David, Martí Broquetas, and Jürgen Marc Volm. 2013. "The project benefits of Building Information Modelling (BIM)." *International Journal of project management* 31 (7): 971-980.

Carrijo, D.Salvio, A.Prieto Oliva, and W.de Castro Leite Filho. 2002. "Hardware-In-Loop Simulation Development." *International Journal of Modelling and Simulation* 22 (3): 167-175.

Castaño, Fernando, Stanislaw Strzełczak, Alberto Villalonga, Rodolfo E Haber, and Joanna Kossakowska. 2019. "Sensor reliability in cyber-physical systems using internet-of-things data: A review and case study." *Remote sensing* 11 (19): 2252.

Cho, Seong Myun, and Tag Gon Kim. 2001. "Analysis of Feasibility for Real-Time Simulation of RT-DEVS Models." *IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace.* IEEE. 3069-3074.

Chong, Heap-Yih, Cen-Ying Lee, and Xiangyu Wang. 2017. "A mixed review of the adoption of Building Information Modelling (BIM) for sustainability." *Journal of cleaner production* 142: 4114-4126.

D.L. Hall, J. Llinas. 1997. "An introduction to multisensor fusion." *Proceedings of IEEE 85.*

Darragi, N., E.M El-Koursi, and S. Collart-Dutilleul. 2014. "From Goal Modeling of Real-time Control System to RT-DEVS Safety Properties Analysis." *IFAC Proceedings Volumes, 2014* 47 (2): 162-169.

Earle, Ben, Kyle Bjornson, Cristina Ruiz-Martin, and Gabriel Wainer. 2020. "Development of a Real-Time DEVS Kernel: RT-Cadmium." *SpringSim.*

Earle, Ben, Kyle Bjornson, Joseph Boi-Ukeme, and Gabriel Wainer. 2019. "Design and Implementation of A Building Control System in Real-Time Devs." *2019 Spring Simulation Conference (SpringSim).*

Ferrari, A, and A Sangiovanni-Vincentelli. 2001. "System design: traditional concepts and new paradigms." *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors.*

Fromel, Bernhard. 2016. "Interface Design in Cyber-Physical Systems-of-Systems." *11th System of Systems Engineering Conference (SoSE).* IEEE. 1-8.

Furfaro, Angelo, and Libero Nigro. 2008. "Embedded Control Systems Design based on RT-DEVS and Temporal Analysis using UPPAAL." *2008 International Multiconference on Computer Science and Information Technology, 2008-10.* IEEE. 601-608.

Gajski, D.D, and F Vahid. 1995. "Specification and design of embedded hardware-software systems." *IEEE Design & Test of Computers* 12 (1): 53-67.

Galetto, Mastrogiacomo, L., Maisano, D., & Franceschini, F. 2015. "Cooperative fusion of distributed multi-sensor LVM (Large Volume Metrology) systems." *CIRP Annals* 64 (1): 483–486.

Gao, Hao, Christian Koch, and Yupeng Wu. 2019. "Building information modelling based building energy modelling: A review." *Applied Energy* 320-343.

Ghaffarianhoseini, Ali, John Tookey, Amirhosein Ghaffarianhoseini, Nicola Naismith, Salman Azhar, Olia Efimova, and Kaamran Raahemifar. 2017. "Building Information Modelling (BIM) uptake: Clear benefits, understanding its implementation, risks and challenges." *Renewable & sustainable energy reviews* 75: 1046-1053.

Giambiasi, Paillet, and Chane. 2003. "From timed automata to DEVS models." *Proceedings of the 2003 Winter Simulation Conference.*

Gourlis, Georgios, and Iva Kovacic. 2017. "Building Information Modelling for analysis of energy efficient industrial buildings – A case study." *Renewable and Sustainable energy reviews* 68: 953-963.

Gunes, Volkan, Steffen Peter, Tony Givargis, and Frank Vahid. 2014. "A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems." *KSII transactions on Internet and information systems* 4242-4268.

Helen, Gill. 2008. "A continuing vision: Cyber-Physical Systems." *Fourth Annual Carnegie Mellon Conference on the Electricity Industry.*

Holloway, C.M. 1997. "Why Engineers Should Consider Formal Methods." *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings.* IEEE. 1:1.3-16.

Hong, Joon Sung, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. 1997. "A Real-Time Discrete Event System Specification Formalism for Seamless Real-time Software Development." *Discrete Event Dynamic Systems: Theory and Applications* 355-375.

Hongyan, Gao. 2009. "A Simple Multi-Sensor Data Fusion Algorithm Based on Principal Component Analysis." *ISECS International Colloquium on Computing, Communication, Control, and Management.* IEEE. 423-426.

Hosking, M, and F Sahin. 2009. "An XML Based System of Systems Agent-in-the-Loop Simulation Framework using Discrete Event Simulation." *2009 IEEE International Conference on Systems, Man and Cybernetics* (10): 3293-3298.

Hu, Xiaolin, and Bernard P. Zeigler. 2005. "Model Continuity in the Design of Dynamic Distributed Real-Time Systems." *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A: SYSTEMS AND HUMANS, VOL. 35, NO. 6.* IEEE. 867-878.

Irwanto, H.Y, and E Artono. 2018. "Correlation of Hardware in the Loop Simulation (HILS) and real control vehicle flight test for reducing flight failures." *Journal of Physics: Conference Series.*

K. Xiao, S. Ren, and K. Kwiat. 2008. "Retrofitting cyber physical systems for survivability through external coordination." *Proc. Hawaii Int. Conf. Syst. Sci.*

Katipamula, Srinivas, and Michael R Brambley. 2005. "Methods for fault detection, diagnostics, and prognostics for building systems - A review, Part I." *HVAC&R research* 11 (1): 3-25.

Kaur, Raj kamal, Babita Pandey, and Lalit Kumar Singh. 2018. "Dependability Analysis of Safety Critical Systems: Issues and Challenges." *Annals of nuclear energy 120* 127-154.

Khaitan, Siddhartha Kumar, and James D McCalley. 2015. "Design Techniques and Applications of Cyberphysical Systems: A Survey." *IEEE systems journal* 9 (2): 350-365.

Khaleghi, Bahador, Alaa Khamis, Fakhreddine O Karray, and Saiedeh N Razavi. 2013. "Multisensor data fusion: A review of the state-of-the-art." *Information fusion* 14 (1): 28-44.

Kim, Kyoung-Dae, and P. R Kumar. 2012. "Cyber-Physical Systems: A Perspective at the Centennial." *Proceedings of the IEEE, 2012-05.*

Köhler, Christian. 2011. *Enhancing Embedded Systems Simulation A Chip-Hardware-in-the-Loop Simulation Framework.* Wiesbaden: Vieweg+Teubner Verlag.

Koval, V. 2001. "The competitive sensor fusion algorithm for multi sensor system." *Proceedings of the International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.*

Lee, Donghwan, Gichun Cha, and Seunghee Park. 2016. "A study on data visualization of embedded sensors for building energy monitoring using BIM." *International Journal of Precision Engineering and Manufacturing* 17 (6): 807-814.

Lee, E. 2010. "CPS foundations." *Proc. DAC.*

Marcosig, Ezequiel Pecker, Juan I Giribet, and Rodrigo Castro. 2018. "DEVS-OVER-ROS (DOVER): A Framework for Simulation-Driven Embedded Control of Robotic Systems based on Model Continuity." *Winter Simulation Conference.*

Marwedel, Peter. 2006. *Embedded System Design.* New York: NY: Springer US.

Moallemi, Mohammad, and Gabriel Wainer. 2020. "Designing real-time systems using imprecise discrete-event system specifications." *Software, practice & experience* 50 (8): 1327-1344.

Moallemi, Mohammad, and Gabriel Wainer. 2013. "Modeling and simulation-driven development of embedded real-time systems." *Simulation modelling practice and theory* 38: 115-131.

Moallemi, Mohammad, and Gabriel Wainer. 2013. "Modeling and simulation-driven development of embedded real-time systems." *Simulation Modelling Practice and Theory* 115-131.

Motawa, Ibrahim, and Abdulkareem Almarshad. 2013. "A knowledge-based BIM system for building maintenance." *Automation in Construction* 29: 173-182.

Nebut, C, F Fleurey, Y Le Traon, and J.-M Jezequel. 2006. "Automatic Test Generation: A Use case driven approach." *IEEE Transactions on Software Engineering* 32 (3): 140-155.

Noordin, Basri, M. A. ., & Mohamed, Z. 2018. "Sensor fusion algorithm by complementary filter for attitude estimation of quadrotor with low-cost IMU." *Telkomnika* 16 (2): 868-875.

Pratt, Annabelle, Mark Ruth, Dheepak Krishnamurthy, Bethany Sparn, Monte Lunacek, Wesley Jones, Saurabh Mittal, Hongyu Wu, and Jesse Marks. 2017. "Hardware-in-the-loop simulation of a distribution system with air conditioners under model predictive control." *2017 IEEE Power & Energy Society General Meeting.*

Rajkumar, Ragunathan, Insup Lee, Lui Sha, and John Stankovic. 2010. "Cyber-physical systems: the next computing revolution." *Proceedings of the 47th Design Automation Conference.*

Raol, J. R. 2010. *Multi-Sensor Data Fusion with MATLAB.* Boca Raton: Taylor & Francis.

Ruiz-Martin, Cristina, Alara Al-Habashna, Gabriel Wainer, and Laouen Belloli. 2019. "Control of a Quadcopter Application with DEVS." *2019 Spring Simulation Conference (SpringSim).*

Sangiovanni-Vincentelli, A, and G Martin. 2001. "Platform-Based Design and Software Design Methodology for Embedded Systems." *IEEE Design & Test of Computers* 18 (6): 23-33.

Sanislav, Teodora, Sherali Zeadally, George Dan Mois, and Hacène Fouchal. 2019. "Reliability, failure detection and prevention in cyber-physical systems (CPSs) with agents." *Concurrency and computation* 31 (24).

Sarjoughian, Hessam S, and Soroosh Gholami. 2015. "Action-level real-time DEVS modeling and simulation." *Simulation: Transaction of the Society for Modeling and Simulation* 91 (10): 869-887.

Smith, Michael, James Miller, and Steve Daeninck. 2009. "A Test-oriented Embedded System Production Methodology." *Journal of Signal Processing Systems* 56 (1): 69-89.

Song, Hae Sang, and Tag Gon Kim. 2005. "Application of Real-Time DEVS to Analysis of Safety Critical Embedded Systems: Railroad Crossing Control Example." *Simulation, 2005-02, Vol.81 (2).* Scopus. 119-136.

Springintveld, Jan, Frits Vaandrager, and Pedro R. D'Argenio. 2001. "Testing timed automata." *Theoretical computer science* 254: 225-257.

Subramanian, Narayanan, and Lawrence Chung. 2000. "Testable embedded system firmware development: the out–in methodology." *Computer standards and interfaces* 22 (5): 337-352.

Volk, Rebekka, Julian Stengel, and Frank Schultmann. 2014. "Building Information Modeling (BIM) for existing buildings — Literature review and future needs." *Automation in construction* 38: 109-127.

Wainer, Gabriel. 2015. "DEVS modelling and simulation for development of embedded systems." *Winter Simulation Conference (WSC).* 73–87.

X. Li, X. Liang, X. Shen, J. Chen, and X. Lin. 2011. "Smart community: An Internet of things application." *IEEE Commun. Mag.* 49 (11): 68-75.

Y. Tan, M. Vuran, and S. Goddard. 2009. "Spatio-temporal event model for cyber-physical systems." *Proc. IEEE ICDCS Workshops.*

Zeigler, B.P, and J Kim. 1993. "Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control." *IEEE Transactions on Robotics and Automation* 9 (3): 351-356.

Zeigler, Bernard P. 1976. *Theory of Modeling and Simulation.* New York: Wiley.

Zeigler, Bernard P. 1984. *Multifacetted Modelling and Discrete Event Simulation.* London: Academic Press.

Zhang, Xiang, Ge Li, Peng Wang, zhonghua Yang, Jiao Lu, and Jie Zhang. 2018. "Design and Implementation of Real-time DEVS Event Scheduling Algorithm." *2018 3rd International Conference on Mechanical, Control and Computer Engineering (ICMCCE).*