Digital twin incubator using ROS2 and model-driven development

by

Benjamin Thompson

B.S., Kansas State University, 2022

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2025

Approved by:

Major Professor
Dr. Robby

Copyright

Abstract

Cyber-physical systems, computer systems that can interact with their environment using actuators and sensors, can be very difficult to develop and ensure proper functionality, especially within a safety critical context. Digital twins are an increasingly popular technique to help ensure the correct operation of these systems. Digital twin systems contain two parts: the physical twin, which is the cyber-physical system itself, and a digital twin, which is a digital representation of the physical twin that is used to perform runtime analysis, simulations of the physical twin using real-time sensor data, and perform self-adaptation on the physical twin to mitigate predicted faults. A digital twin system can be a very complex system with several components each needing to communicate with each other over a network in real time. Setting up these communication pathways by-hand can be very time consuming and potentially error-prone. This report examines how the Sireum High Assurance Modeling and Rapid engineering for embedded systems (HAMR) code generation tool and the Robot Operating System 2 (ROS2) may be used to improve the engineering of twinned systems. The investigation is based on a case study of a twinned system for an incubator for fermented soy-based food. The incubator system was originally designed by Feng et al (2021) at Aarhus University as an open source test bed for digital twin concepts. In addition to a cyber-physical system for controlling the environment in an enclosure for fermenting a soy-based food product, this incubator contains a digital twin service to predict if the incubator lid is open using a simulation and real-time sensor data, and can recalibrate the physical twin in response. Given a model of the twinned system written in the Architecture Analysis and Design Language (AADL), the experimental HAMR code generator for ROS2 can output a ROS2 package with ROS nodes for each system component, and all communication pathways completely configured. Utilizing the HAMR code generator for ROS2, an implementation was made that maintains temperature when the lid is closed, and detects

& adapts to a lid open event and with fewer lines of hand-written code than the original Aarhus University implementation. This report finds that the use of these tools for digital twin development is promising, but more investigation is needed and some improvements to tooling need to be made.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank several people that made this report possible. First, I would like to thank Dr. John Hatcliff, Dr. Robby, and Jason Belt for all of the knowledge and advice they have given me. I would then like to thank Clint McKenzie for his wonderful HAMR code generator for ROS2. Next, I would like to thank Brian Parks for helping me understand and solder the circuitry for the incubator. Finally, I would like to extend my deepest and sincerest of thanks to Ella Tucker, Fig Weins, and Sierra Hawbaker for the tremendous amount of support you have all given me throughout my Master's program. I never would have made it this far without the three of you.

Chapter 1

Introduction & Background

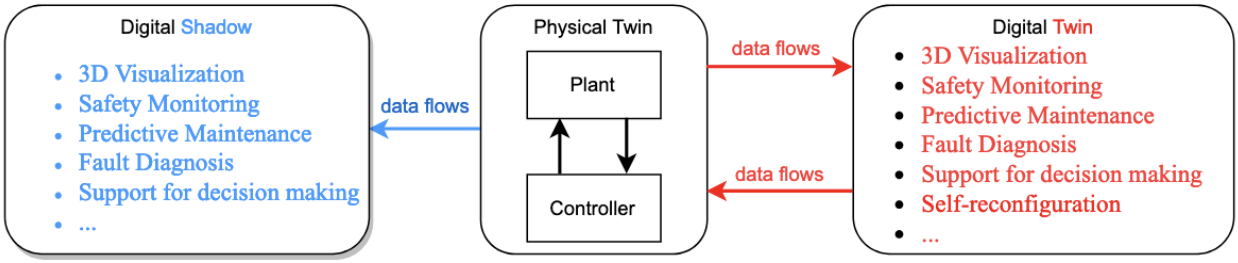

Figure 1.1: *Diagram from Feng et al. showing the communication structure of a Digital Twin System.*

Cyber-physical systems, computer systems that can interact with their environment using actuators and sensors, can be very difficult to develop and ensure proper functionality, especially within a safety critical context. Digital twins are an increasingly popular technique for ensuring high-assurance cyber-physical systems are performing correctly during runtime. A digital twin system contains two parts: the physical twin and the digital twin [1]. The physical twin encapsulates the sensors, actuators, the physical space actuated on, and the controller code in the cyber-physical system. The digital twin is a digital representation of the physical twin, used to perform simulations, runtime analysis, fault detection, and can even be used to recalibrate the physical twin when faults or anomalies are detected. According to Feng et al., a key component of a digital twin is bi-directional, real-time communication between the digital twin and physical twin. The physical twin typically sends

state information, including sensor data and controller states) to the digital twin, and the digital twin typically sends instructions to the physical twin to help it adapt to anomalies [1]. See Figure 1.1 for a high level overview of these systems. These systems can be very complicated, and they have a need for reliable, well-defined communication pathways. This report focuses on investigating the use of model driven development tools in the development of these digital twin systems. Specifically, this report will investigate the following questions:

- Is AADL a good tool for designing digital twin systems?
- Is ROS2 a good framework to facilitate digital twin system communications?
- Is the HAMR ROS2 code generator good for providing scaffolding for digital twin systems?

These questions were investigated by reimplementing an existing Digital Twin system. The system chosen was originally designed and implemented at Aarhus University, and is a simple incubator designed to incubate a soy-based product[2]. To answer the questions listed above, the reimplementation was designed using AADL and implemented using the HAMR ROS2 code generator[3], and reimplements a subset of the digital twin services provided by the original implementation. Doing this provides a completed digital twin implementation using these tools, and comparisons between implementations can be made. This report concludes that ROS2 is a suitable tool for Digital Twin implementation, and that AADL and the HAMR ROS2 code generator are highly promising, but more investigations are needed and improvements to tooling need to be made. The rest of this chapter provides information on the tools being used. Chapter 2 gives a description of the Aarhus University incubator, Chapter 3 gives an in-depth look at the architecture and implementation of the HAMR ROS implementation of the incubator, Chapter 4 compares the two implementations, and Chapter 5 summarizes the findings of this report.

[1] A digital shadow is a system where data only flows out from the physical twin.

1.1 AADL

The Architecture Analysis and Design Language (AADL) is a software modeling tool developed by the Software Engineering Institute at Carnegie Mellon University [4]. Systems modeled in AADL provide an abstraction of a complex software system, viewing the system as discrete components that interact only through the sending and receiving of messages. Components can have a set of input and output ports, and each port may be one of 3 different types: an event port, a data port, or an event data port. Event ports are used to notify a component that there is an event that needs to be handled, and the event contains no data. Data ports are used to transfer data between components, but they do not notify the component of an event. Event data ports are a combination of both types of ports, and both notify the component of an event that needs to be handled, and provide the component data. For the purposes of this report, components can be one of three different types: threads, processes, and systems. Threads are the basic block of an AADL system, and the only component in the abstraction that represents any code. In this report, two types of AADL threads will be considered[2]: sporadic and periodic. Sporadic threads only activate when they receive an event on at least one of their in event ports, and periodic threads activate repeatedly, with gaps between execution based on a specified period. Processes are groups of thread components, and they define the connections between each thread. Systems are groups of processes or other systems, and similar to processes, define how the components communicate. For the purposes of this report, systems are the top level component. Examples of AADL are provided in Section 3.3.

1.2 ROS2

Robot Operating System 2 (ROS2) is a framework for developing robotics applications[5]. Despite the name, ROS2 is not an operating system, but a platform containing libraries, frameworks, and tools that aid in the development and testing of robotic and other cyber-

[2]Other thread types are available, but are not considered in this report.

physical systems. A ROS system consists of one or more discrete software elements called nodes, and these nodes can communicate to each other via a subscribe-publish communications platform. Nodes can subscribe to and publish on various topics in order to share data. These topics are open, and any ROS node on the network can freely read from and write to, or even create any topic without any authentication. ROS systems are entirely distributed and require no central server for communication to occur. If any node goes offline, all topics may still be written to and read from by other nodes in the system. Nodes can be written in several languages, but the two best-supported languages are C++ and Python. A big feature of ROS is its ability to interact with several simulation and visualization tools, including RVIZ2, a tool for visualizing the current state of your robot, and Gazebo, a tool to used simulate and prototype robots completely in software.

1.3 Sirem HAMR

Sireum High Assurance Modeling and Rapid engineering for embedded systems (HAMR) is a tool developed by SAnToS Lab at Kansas State University [6]. It is used to take software models specified in AADL [3] and generate skeleton code based on the model, creating code to represent each thread component and generating the infrastructure to facilitate the communication between components. HAMR can generate code for various targets, including Sireum Slang, C for the SeL4 microkernel, and, more recently, C++ for ROS2. As this project focuses on ROS2, the C++ for ROS2 code generator will be used. In this particular target, threads are represented as ROS nodes, and communication between threads happens over ROS topics.

[3]Support for SysMLv2 has also been recently added

Chapter 2

The Digital Twin Incubator

The main component of this report is a reimplementation of the Digital Twin Incubator designed at Aarhus University in Denmark [7], which is used as a case study in the use of both ROS2 and model driven development tools in the design and implementation of digital twin systems. This system makes an ideal case study for this report because it is relatively simple, well-documented, and a fully functional reference implementation[2] is available and easy to use.

2.1 Hardware Overview

The hardware components of both my implementation and the implementation from Aarhus University are nearly identical, barring a few parts I was unable to source quickly in the United States. The main components include: a Raspberry Pi 4b (used to control the hardware), a large styrofoam box (used as a contained, insulated space), a PC fan (used to increase airflow inside the box), a 3D printer heat bed (used to heat the air inside the box), 3 DS18S20 temperature sensors (two to monitor the internal box temperature, one to monitor the temperature outside the box), and a custom PCB designed at Aarhus University (used to allow the 5 volt Raspberry Pi to control the 18 volt heat bed and 24 volt fan) [7].

I was unable to source the same styrofoam box they used, which had an internal volume of

around $0.03m^3$, so I got a different styrofoam box with an internal volume of around $0.02m^3$. They also used a RepRap PCB Heat bed MK2a heatbed, while I used a MK3 ALU-Heatbed Dual Power, which is very similar and has the same physical area.

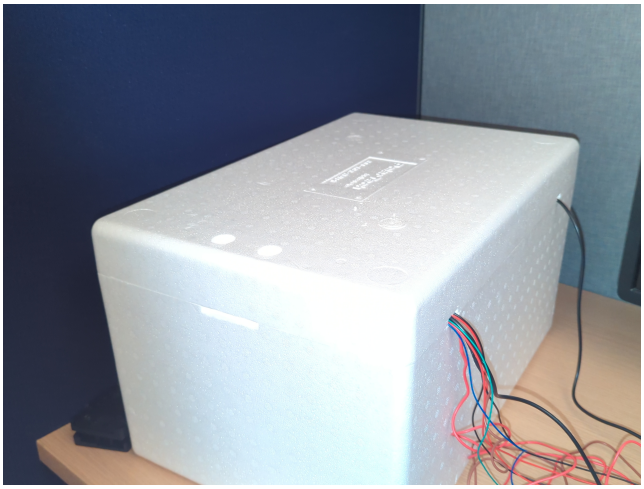

Figure 2.1: *Photograph of the outside of the incubator*

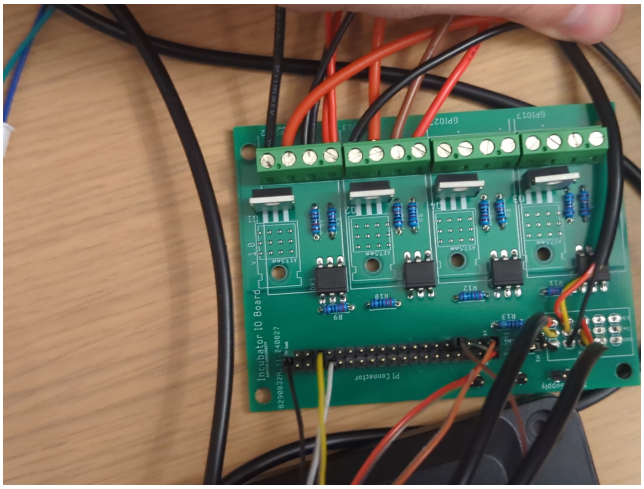

Figure 2.2: *Photograph of the incubator circuitry*

The heatbed power, PC fan power, heat sensors, and Raspberry Pi GPIO pins are all connected to the custom PCB, along with a 12V power supply and a 24V power supply. This PCB (pictured in Figure 2.2) provides the circuitry required to wire up the temperature sensors to the Raspberry Pi, and allows the 5V Raspberry Pi to control the 12V heatbed and 24V PC fan over the GPIO pins. The heatbed is placed in the middle of the inside of

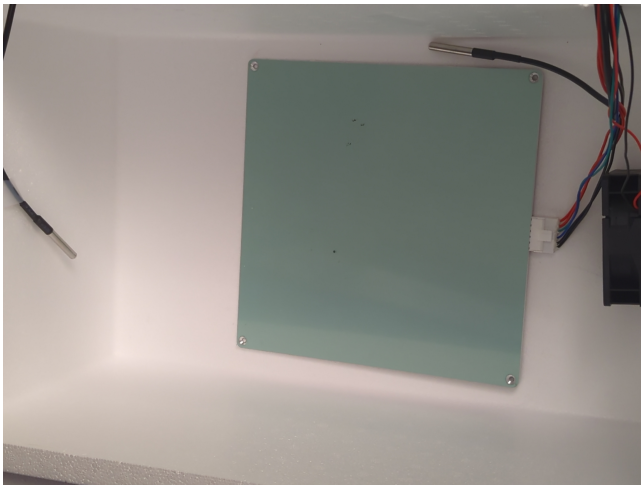

Figure 2.3: *Photograph of the inside of the incubator*

the insulated box, with the fan placed on one end of the heatbed. A temperature sensor is taped inside on a wall opposite the fan, with a second taped on an adjacent wall. The third is placed outside the box, and small holes are cut in the lid to allow the wires to enter the box. The internals of the box can be seen in Figure 2.3. The Raspberry Pi can then control the heatbed by setting pin 12 on or off, and similarly, it can control the fan by setting pin 13 on or off. It can also use the one-wire protocol to read from each of the temperature sensors, which is done via a file (/sys/bus/w1/devices/⟨uid⟩/w1_slave), not through the GPIO API directly, though the thermometers are connected to GPIO pin 4.

2.2 Aarhus University Implementation: Inc-AU

The Aarhus University implementation of the Digital Twin incubator, which will be denoted as Inc-AU, is implemented completely in Python. The architecture for the system can be seen in Figure 2.4. Each individual component from the architecture diagram is implemented as its own python module, and communication is handled over a Python messaging service called RabbitMQ[7]. Communication in RabbitMQ happens through a single server, and if that server is not running, the system cannot be run. Inc-AU has 3 main goals: maintain the temperature of the box within a target range, recalibrate the system in the prescense of anomalies, and log all important aspects of the system.

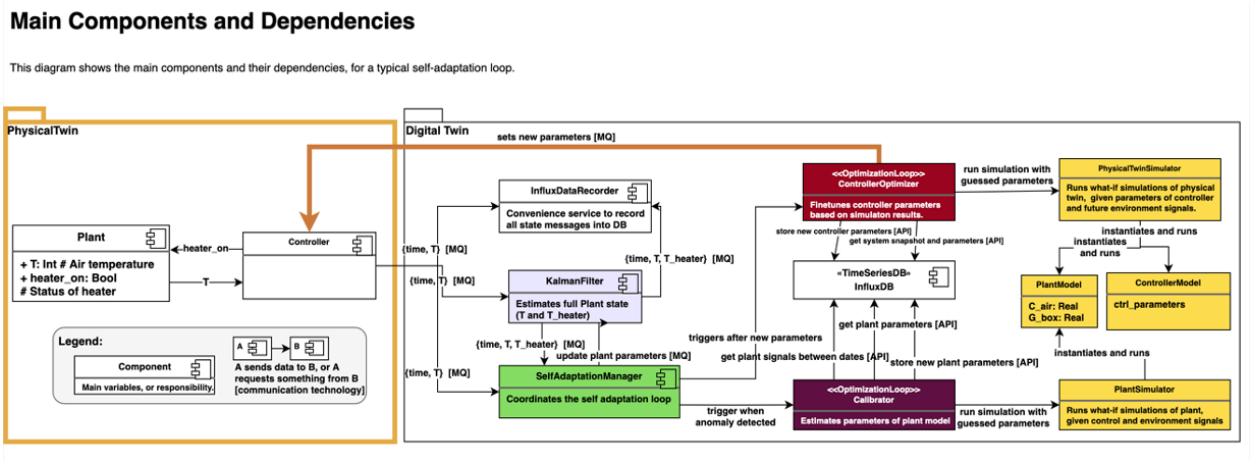

Figure 2.4: *Diagram of the architecture for the Aarhus University Digital Twin Incubator[2]*

Inc-AU has two components that are run as part of the physical twin: the plant and the controller. Both of these components are designed to be run on the Raspberry Pi. The plant is the component that directly manages the hardware. Every 3 seconds, the component will check to see if it received any commands telling it to change the state on one of the actuators, then, regardless of the previous step, send out a message containing the current temperature readings and the state of the actuators. The controller makes all decisions for the physical twin and relays instructions to the plant. Upon receiving a message from the plant detailing the current state of the plant, the controller will step through a state machine to determine what actions it should take next. This state machine is detailed in Figure 2.5, but briefly, it will heat the incubator up to a target temperature through a heating cycle, periodically instructing the plant to turn the heat bed on and off to better control the temperature of the heat bed. When the target temperature is exceeded, the controller will then instruct the plant to turn the heat bed off until it reaches the lower limit temperature, computed as the target temperature minus a lower limit constant. It then sends out messages detailing how the plant should react, and a message with its internal state.

Inc-AU has several digital twin components, but for this report we will be focusing on three: the InfluxDB recorder, the plant kalman filter, and the energy saver. The InfluxDB recorder is the main logging component of this system. It reads every message sent over the RabbitMQ server and logs it into an InfluxDB database, which is a SQL-like time-series

database. When it receives a message from the plan detailing the current device state, the plant kalman filter performs a simulation to predict the current internal temperature of the box and heat bed based on the previous internal box temperature, and sends a out message with the predictions and the error between the actual and predicted box temperature. The last component is the energy saver. This component is not listed in Figure 2.4, but is available in the Inc-AU repository and acts as a simplified version of the much larger self adaptation loop that is present in the diagram. If, when this component receives a prediction from the kalman filter, it sees that the prediction error is greater than a specified threshold, it will make the assumption that the lid of the incubator is open, and instruct the plant controller to change the target temperature to 60% of the current value with the goal of reducing the energy used while the lid is open. When the prediction error gets back into the correct range, the energy saver will then instruct the controller to reset the target temperature back to the original value. The Python code for this component is given in Figure 2.6, and serves as an example of what the code for components of Inc-AU look like. The remaining components are much more complicated simulations, and they are mainly focused on refining the parameters of the controller and the simulation parameters based on historical data from the database. While they are important, they will not be considered in this report.

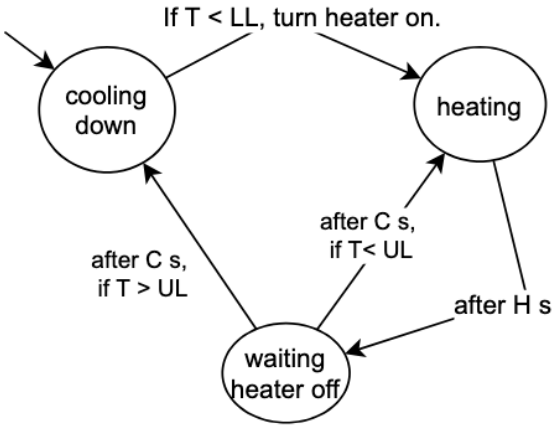

Figure 2.5: *Diagram of the State Machine for the Physical Controller, obtained from Feng et al.*

```python
import logging

from digital_twin.communication.rabbitmq_protocol import
    ROUTING_KEY_KF_PLANT_STATE
from incubator.communication.server.rabbitmq import Rabbitmq
from incubator.communication.shared.protocol import
    ROUTING_KEY_UPDATE_CLOSED_CTRL_PARAMS


class EnergySaverServer:

    def __init__(self, error_threshold, ctrl_config_baseline,
    rabbit_config):
        self._l = logging.getLogger("EnergySaverServer")
        self.filter = None
        self.rabbitmq = Rabbitmq(**rabbit_config)

        self.in_prediction_error = 0.0
        self.error_threshold = error_threshold

        self.ctrl_temperature_desired = ctrl_config_baseline['
    temperature_desired']

    def setup(self):
        self.rabbitmq.connect_to_server()

        self.rabbitmq.subscribe(routing_key=ROUTING_KEY_KF_PLANT_STATE,
                                on_message_callback=self.kalman_step)

    def start_monitoring(self):
        self.rabbitmq.start_consuming()

    def kalman_step(self, ch, method, properties, body_json):
        self.in_prediction_error = body_json["fields"]["prediction_error"]

        new_temp_desired = self.ctrl_temperature_desired*0.6 if abs(self.
    in_prediction_error) > self.error_threshold else self.
    ctrl_temperature_desired

        self._l.debug(f"Updating closed loop controller temperature
    desired to: {new_temp_desired} due to error {abs(self.
    in_prediction_error)}.")
        self.rabbitmq.send_message(ROUTING_KEY_UPDATE_CLOSED_CTRL_PARAMS,
    {
            "temperature_desired": new_temp_desired,
        })

```

Figure 2.6: *Aarhus university implementation of the Energy Saver component[2]*

Chapter 3

Architecture, Design & Implementation

3.1 The HAMR-ROS Incubator: Inc-RH

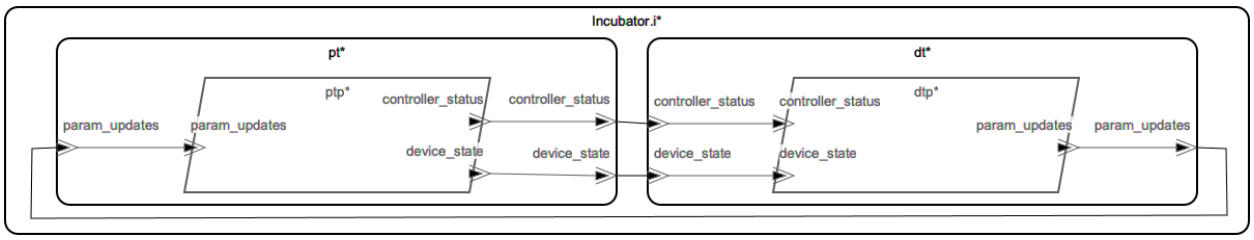

Figure 3.1: *Diagram of the AADL architecture for the top-level Inc-RH system*

The architecture and system model of the HAMR-ROS incubator system, denoted Inc-RH, was designed in the AADL architecture language. It is based on the architecture of Inc-AU, matching it as closely as possible, and making modifications where needed to make it fit in AADL, ROS2, and C++ paradigms. Inc-RH is made up of two subsystems: the Physical Twin system, which focuses on reading the sensor data and controlling the hardware, and the Digital Twin system, which focuses on data recording, system simulation, anomaly detection, and system adaptation. Each subsystem, which are each represented as an AADL system, contains a single AADL process containing several AADL threads. Each thread

represents a single service that the system performs. The inputs and outputs from each system are then connected to form the whole incubator AADL system. See Figure 3.1 for a diagram of the system. The architectures for the Physical Twin and Digital Twin systems will be discussed in Sections 3.2 and 3.3 respectively.

3.2 Inc-RH Physical Twin System Architecture

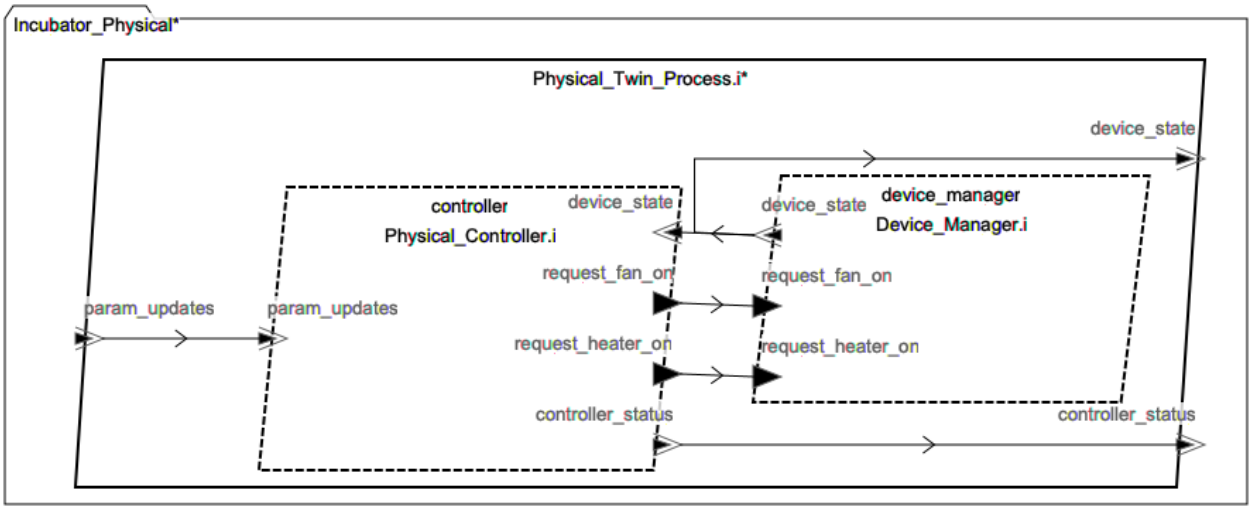

Figure 3.2: *Diagram of the AADL architecture for the Inc-RH Physical Twin*

The Physical Twin system is designed to be run on the Raspberry Pi itself and, despite what is implied by the AADL, can be run entirely independently from the Digital Twin subsystem. The system contains two AADL thread components: the Physical Controller thread, and the Device Manager thread.

The Device Manager thread is a periodic thread with a period of 3 seconds, and is responsible for all hardware interactions. This is the only periodic thread in the entire Incubator system, and as such, acts as a clock: every time this thread executes, it will cause every other thread to execute at least once. Each time the thread executes, it reads the two in data ports, request_fan_on and request_heater_on, which both have boolean values, and then uses their values to determine if the fan and heater should be turned on or off. It then reads from each of the three temperature sensors, puts the current status of each sensor and

actuator into a ROS message, and sends it over the device_state port.

The Physical Controller thread is a sporadic thread, and it has two event data ports it responds to, device_state and param_updates. Internally, this thread has a representation of the state machine found in Figure 2.5. Each time this thread receives a device_state event, it uses the sensor data to make one step through the state machine. It then uses the current state to determine what the state of the heater and fan should be (in Inc-RH, the fan will always be on), and sends the command out over the request_fan_on and request_heater_on data ports. It then outputs a message containing the current status of the state machine on the controller_state port. If the param_updates event is received, the contents of the event are used to change the target temperature the controller tries to attain. This message always comes outside of the Physical Twin system.

3.3 Inc-RH Digital Twin System AADL Architecture

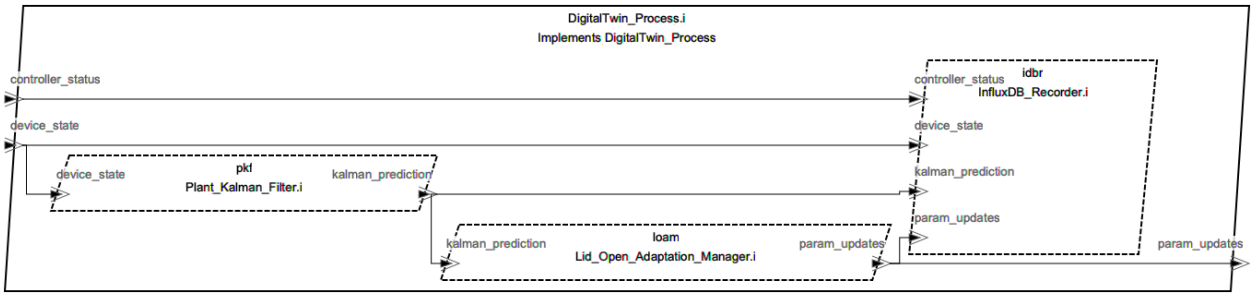

Figure 3.3: *Diagram of the AADL architecture for the Digital Twin*

While it can be run on the Raspberry Pi with the Physical twin, the Digital Twin system is intended to be run a separate computer. The system has 3 AADL thread components: the InfluxDB recorder thread, the Plant Kalman filter thread, and the Lid Open Adaptation Manager thread. Each of these threads represents one of the digital twin services implemented in Inc-RH.

The InfluxDB recorder thread is fairly simple in scope: it listens for the main event sent by each thread in the system (device_state from the Device Manager, controller_status from the Physical Controller, kalman_prediction from the Plant Kalman Filter, and the

```
    thread Lid_Open_Adaptation_Manager
    features
      kalman_prediction: in event data port Incubator_Types::
   Kalman_Prediction.i;
      param_updates: out event data port Incubator_Types::
   Closed_Loop_Param_Updates.i;
    properties
      Thread_Properties::Dispatch_Protocol => Sporadic;
      Timing_Properties::Period => 1 sec;
  end Lid_Open_Adaptation_Manager;

```

Figure 3.4: *Textual AADL representation of the Lid Open Adaptation Manager*

param_updates from the Lid Open Adaptation Manager) and writes the contents of the event message to an InfluxDB database. In Inc-RH, this is mostly for data gathering and analysis, but in Inc-AU, the database is used in running various other digital twin services that are not implemented here.

The Plant Kalman Filter thread listens for a device_state event, and uses that in tandem with a Kalman Filter to perform a one-timestep simulation of the state of the physical incubator. This simulation runs under the assumption that the lid is closed. The thread predicts the current average temperature inside the box, the current temperature of the heat-bed, and computes the error of the box prediction, and sends it out on the kalman_prediction event data port. The simulation uses the previous read temperature as the starting point, so if a large deviation occurred between the actual and predicted values in the previous simulation step, the next simulation step will not be affected by the deviation.

The Lid Open Adaptation Manager thread listens for a kalman_prediction event from the Plant Kalman Filter. When it receives one, it checks the prediction error, and if the magnitude prediction error is greater than 1 degree Celsius, it assumes the lid is open, and instructs the Physical Twin to change the target temperature to 60% of the original. When the prediction error is back within 1 degree, the thread will then tell the Physical Twin to reset the target temperature. To give an example of the textual AADL representation of a component, Fig. 3.4 contains the textual representation of this thread.

3.4 Implementation

A substantial reason for creating AADL architecture diagrams for Inc-RH is so that we may leverage the ability of code generation tools to automatically create the scaffolding for each of the components. In this project, the Sireum HAMR code generator targeting ROS2 was used. This tool takes an AADL architecture model and generates a set of C++ ROS nodes based on the model. Each AADL thread is represented as a ROS node, and each AADL connection (i.e. an ordered pair containing an in-port and an out-port it is connected to) is represented as a ROS topic. The code generator handles all of the node setup, and just leaves a handful of functions for the developer to implement: the initialize function, where a user can set up the initial state of a node, and a set of functions . In order to maintain as much equivalency as possible with Inc-AU, the functionality of each node was mirrored as closely as possible, just implemented in C++ instead of Python. Chapter 4 will analyze how closely Inc-RH performs compared to In-AU in practice. When an AADL model is passed through the HAMR code generator, a series of C++ files making up a ROS node is output for each AADL thread. This includes a runner file, which serves as the entry point for the node executable (example: Figure 3.5), a file containing the base code for the node, which sets up communication and other infrastructure (example: Figure 3.6), and a user code file, containing functions for the user to fill in. These functions include an initialization function, which is run when the node is created, and either a series of event handlers for sporadic threads, or a time triggered functions for periodic threads. Figure 3.7 contains an example of what this looks like after HAMR code generation. To complete the node, a user just has to fill in these functions with the desired behavior code. An example of the completed Lid Open Adaptation Manager can be seen in Figure 3.8.

A number of ROS specific features are implemented by-hand in this project as well. To facilitate configuration of the nodes, several ROS parameters were added to each node, allowing a user to easily use the ROS environment to set various parameters for each of

```cpp
#include "incubator_cpp_pkg/user_headers/
    Incubator_i_Instance_dt_dtp_loam_src.hpp"

//================================================
//  I n i t i a l i z e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::initialize()
{
    PRINT_INFO("Initialize Entry Point invoked");

    // Initialize the node
}

//================================================
//  C o m p u t e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::handle_kalman_prediction(const
    incubator_cpp_pkg_interfaces::msg::KalmanPredictioni::SharedPtr msg)
{
    // Handle kalman_prediction msg
}
```

Figure 3.5: *code for the Lid Open Adaptation Manager Runner [3]*

```cpp
#include "incubator_cpp_pkg/user_headers/
    Incubator_i_Instance_dt_dtp_loam_src.hpp"

//================================================
//  D O   N O T   E D I T   T H I S   F I L E
//================================================

Incubator_i_Instance_dt_dtp_loam::Incubator_i_Instance_dt_dtp_loam() :
    Incubator_i_Instance_dt_dtp_loam_base()
{
    // Invoke initialize entry point
    initialize();

    PRINT_INFO("Incubator_i_Instance_dt_dtp_loam infrastructure set up");
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto executor = rclcpp::executors::MultiThreadedExecutor();
    auto node = std::make_shared<Incubator_i_Instance_dt_dtp_loam>();
    executor.add_node(node);
    executor.spin();
    rclcpp::shutdown();
    return 0;
}
```

Figure 3.6: *Auto generated base code for the Lid Open Adaptation Manager[3]*

```cpp
#include "incubator_cpp_pkg/user_headers/
    Incubator_i_Instance_dt_dtp_loam_src.hpp"

//================================================
//  I n i t i a l i z e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::initialize()
{
    PRINT_INFO("Initialize Entry Point invoked");

    // Initialize the node
}

//================================================
//  C o m p u t e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::handle_kalman_prediction(const
    incubator_cpp_pkg_interfaces::msg::KalmanPredictioni::SharedPtr msg)
{
    // Handle kalman_prediction msg
}
```

Figure 3.7: *Empty user code for the Lid Open Adaptation Manager*

the nodes, such as the target temperature for the Controller node, or the IP address of the InfluxDB server for the InfluxDB recorder node. In this implementation, all of these parameters have to be set at runtime because parameters are only read once by a given node [1]. To give a visualization of the current state of the physical incubator, four publishers were added to the device manager node to send out standard ROS sensor messages, one for each temperature sensor, and one for the heat-bed. These message types are used so that they may be read by preexisting ROS services. This, combined with a URDF model of the incubator, can be viewed in RVIZ, a ROS visualization tool, to view the current state of the incubator in a 3D model. This mirrors the Godot visualization service from Inc-AU, shown in Figure 3.9. Since HAMR cannot import standard ROS message types, and because HAMR has no way of setting up communication with ROS nodes that exist outside of AADL, this had to be added by hand.

A few code generated elements had to be either modified or rewritten by-hand in order to properly run certain aspects of the incubator system. First, the library to interact with

[1] This is not true in ROS as a whole. Some nodes allow dynamic redefinition of ROS parameters.

```cpp
#include "incubator_cpp_pkg/user_headers/
    Incubator_i_Instance_dt_dtp_loam_src.hpp"
#include <cmath>

//================================================
//  I n i t i a l i z e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::initialize()
{
    PRINT_INFO("Initialize Entry Point invoked");
    this->declare_parameter("target-temp", 35.0f);
    this->declare_parameter("error-threshold", 1.0f);

    // Initialize the node
    desired_target_temperature = cur_target_temp = this->get_parameter("
    target-temp").as_double();
    error_threshold = this->get_parameter("error-threshold").as_double();
}

//================================================
//  C o m p u t e    E n t r y    P o i n t
//================================================
void Incubator_i_Instance_dt_dtp_loam::handle_kalman_prediction(const
    incubator_cpp_pkg_interfaces::msg::KalmanPredictioni::SharedPtr msg)
{
    // Handle kalman_prediction msg
    float new_target_temp = std::abs(msg->prediction_error.value.data) >
    error_threshold ? 0.6*desired_target_temperature :
    desired_target_temperature;
    if(new_target_temp != cur_target_temp)
    {
        auto paramUpdateMsg = incubator_cpp_pkg_interfaces::msg::
    ClosedLoopParamUpdatesi();
        paramUpdateMsg.target_temperature.value.data = new_target_temp;
        PRINT_INFO("Prediction error detected, switching target
    temperature to %.02f", new_target_temp);
        put_param_updates(paramUpdateMsg);
        cur_target_temp = new_target_temp;
    }
}
```

Figure 3.8: *Completed user code for the Lid Open Adaptation Manager[3]*

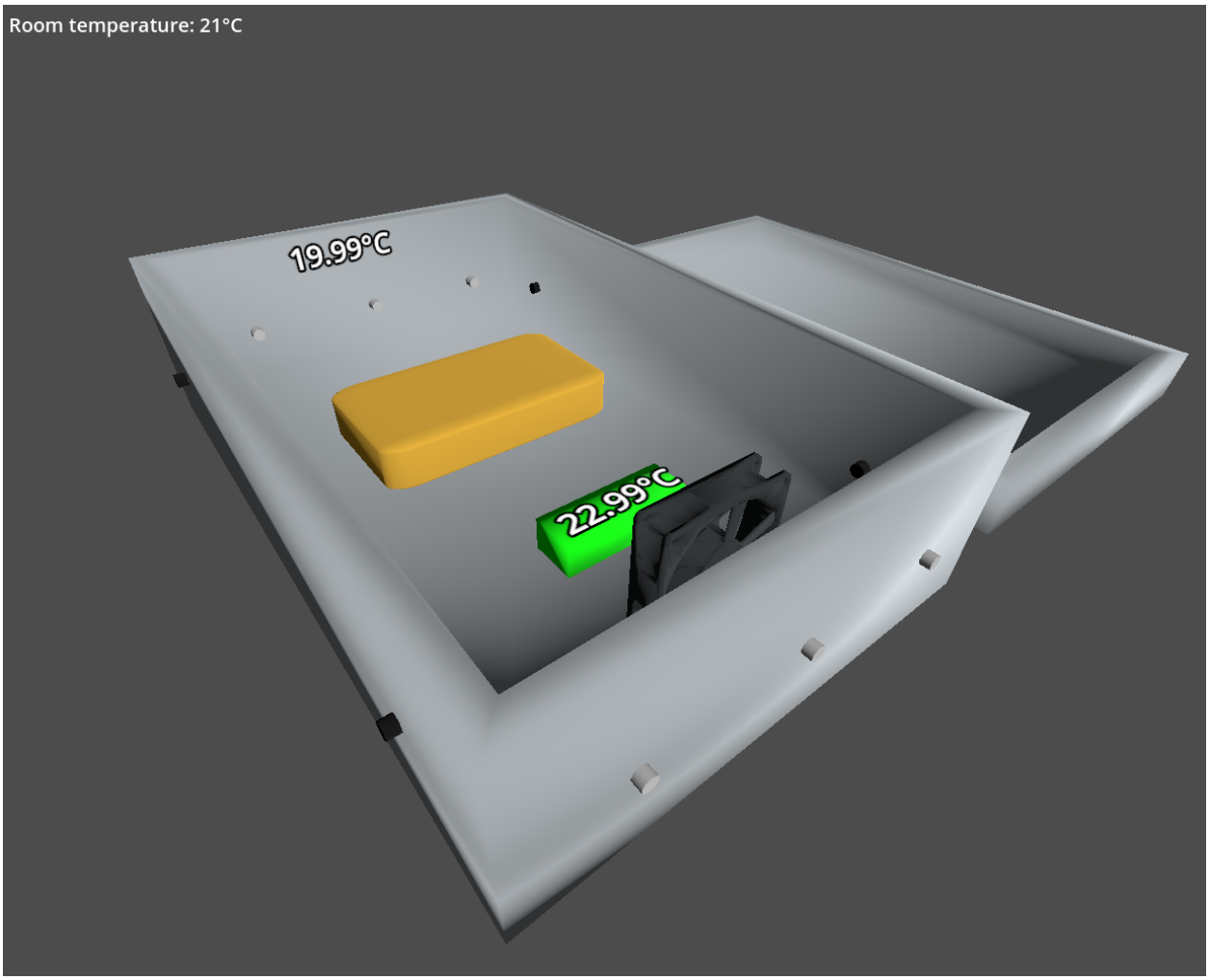

Figure 3.9: *Godot visualizer for Inc-AU*

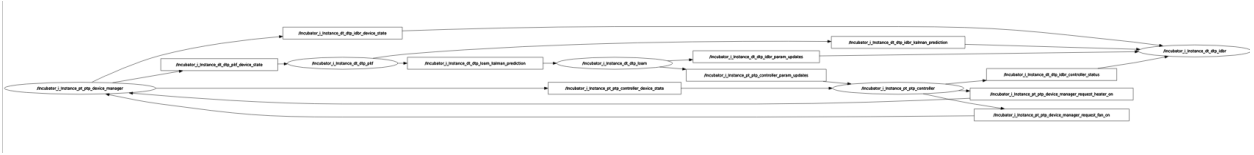

Figure 3.10: *RQT Graph of the incubator digital twin system*

the Raspberry Pi GPIO pins in the device manager has an initialization function that needs to be called early in the life-cycle of the node. I had to modify the entry point function for the node (which isn't supposed to be modified) to call the method. I also needed to add a SIGINT handler in the device manager node to make sure the heating element and fan are turned off when the node exits. Next, due to the complexity of the Kalman filter, I had to implement the Plant Kalman Filter node in Python instead of C++. To do this, I used the C++ skeleton code generated by HAMR and wrote the equivalent Python code as to mimic what a code generated Python node would look like. I then proceeded as I had with the other nodes. Since the Python node is structurally equivalent to a C++ node, and a HAMR Python code generator for ROS2 is in the works, this is a nonissue for my project. The final thing that had to be recreated by hand was the launch file. Right now, the launch file generated by HAMR launches all nodes at once. Since the Physical Twin system and Digital Twin system are designed to be run on separate devices, two launch files are needed, as we don't want to bring all nodes up on each device. A third launch file was also added that just brings up RVIZ with the proper model and model configuration.

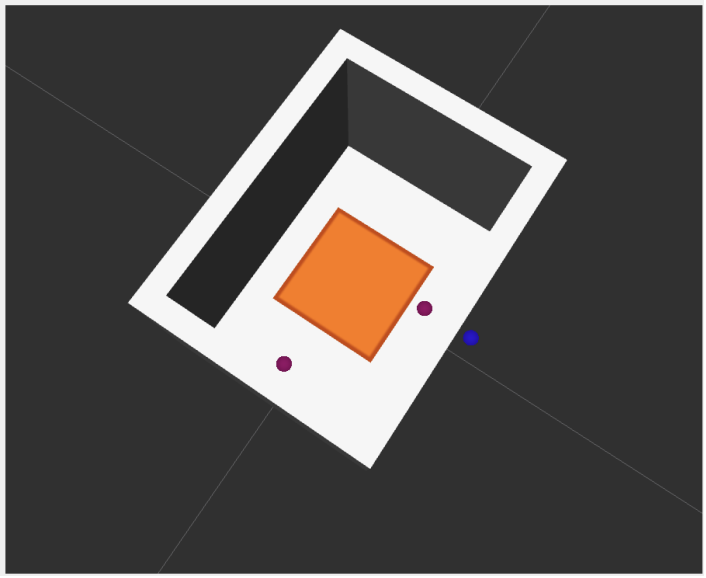

Figure 3.11: *Picture of the RVIZ visualization of the current state of the physical incubator*

Chapter 4

Performance Analysis & Implementation Comparison

4.1 Performance Analysis

In order to ensure that Inc-RH is performing properly, an analysis of how it performs over a long period of time is required. Inc-RH was deployed and run for approximately 12 hours, and the data collected by the InfluxDB recorder node were analyzed. A couple of metrics will be used to evaluate the performance of the node. First, since Inc-RH has a maximum target temperature (35 Celsius) and a minimum target temperature (30 Celsius), we need to ensure that a substantially large number of readings of the box temperature are within that range[1]. Second, since the system assumes the lid is open if the predicted temperature from the Kalman filter has an error magnitude of over 1 degree Celsius, we want to make sure that when the box is closed, the error is between -1 and 1, and similarly, if the box is open, we want the error to quickly get outside of that range.

First we will discuss the performance of the Inc-RH during normal, lid closed operation after the initial warm-up period, as the temperatures would not have reached the target range. A 7 hour and 45 minute time-span during the incubation period was chosen. This

[1]Note that we will be using the average between both internal temperature sensors as our box temperature

period was after the initial warm-up, and did not have any lid-open events. The graph of the temperature can be seen in Figure 4.3. From the graph, it can clearly be seen that the temperature function is periodic, which is expected based on the state machine for the physical controller shown in Figure 2.5, which shows that the incubator should go through the heating cycle until it attains the upper limit temperature, then cool down to the lower limit temperature. In Table 4.1, some statistics calculated from the data are shown. It is worth noting that, since the device manager thread runs once every 3 seconds, there are going to be 3 seconds between each reading. From the table, we can see that approximately 18% of the data points are out of range, and most of the out of range points are above the intended range. We do expect some points to lie out of the target range, as the incubator will not switch states until a temperature outside the target range is read. The fact that most of the points outside the target range are above the upper limit is likely due to the fact that the way cooling down is achieved is by turning the heater off, and letting the box cool back down towards the outside air temperature. Since the box is insulated, and since the heat bed doesn't immediately cool down after it is turned off, it is expected that the box will continue to heat for a small amount of time after the incubator enters the cool-down state. The temperature also generally stays within 0.5 Celsius of the upper and lower bounds, which is within the accuracy range of $\pm 0.5C$ listed for the DS18S20 temperature sensors we are using[8].

Table 4.1: *Statistical analysis of the incubator running Inc-RH over approximately 8 hours*

Median temp	32.56
Average temp	32.58
Max temp	35.5
Min temp	29.78
Number of points above 35	1063
Number of points below 30	645
Total Number of points	9300
Proportion of points out of range	0.1837

We now need to make sure that during our normal operation period, the predicted temperature error stays within our -1 to 1 threshold, since if it does not, that will cause an

unintended recalibration to occur. As stated in Section 3.3, the incubator digital twin performs a 1 time-step simulation to predict the current average temperature of the box using the previously read value as part of the initial conditions. Since this is a simulation, an error between the actual and predicted value is expected, but if the lid is closed, we do not want the error to deviate by more than ± 1 degree Celsius since that will cause the Lid Open Adaptation Manager to predict that the lid is open and send out a message to update the parameters of the incubator. Figure 4.5 is a graph of the error between the actual and predicted values of the temperature over the same time period as Figure 4.3. It is clear from this graph that the error between the temperature was, for the entire 7 hour and 45 minute duration, within our -1 to 1 range of acceptability[2]. The target temperature during this duration was also never changed from 35, meaning no erroneous recalibration occurred.

We finally need to evaluate the performance of Inc-RH during a lid open event. During the approximately 12 hour run of the incubator, I opened the lid one time. A graph of the recorded temperature, predicted temperature, and target temperature during and around the lid open event are shown in Figure 4.1. In this graph, the lid was opened at approximately 10:01:10, and was closed at just before 10:02:00. In the graph, you can see the difference between the actual and predicted temperature widen, and when the distance gets wide enough, the target temperature changes. When the lid is put back on, the difference between the two lessens and the target temperature resets back to the original value. In Figure 4.2, you can see the actual error. It takes about 15 seconds after the lid is open for the system to predict that the lid is open and the target temperature to change, and it takes about 10 seconds for it to predict the lid is closed and for the target temperature to reset. It is likely that if the lid was open for a longer amount of time that the prediction error would have been larger, which would make the amount of time needed for the system to recalibrate to increase.

During the entire 12 hour time the incubator was running, only two recalibration events happened. The first was from the lid open event described previously, and the second was after the very first cycle when the incubator was turned on. This occurred because

[2]The minimum error attained during this period was 0.2, and the maximum error was 0.33

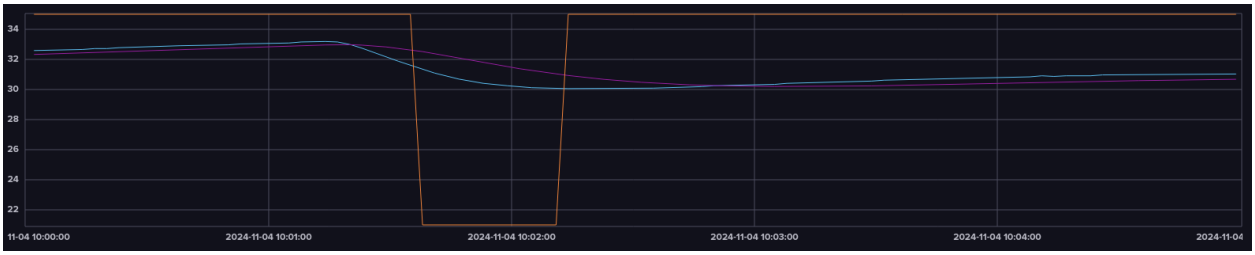

Figure 4.1: *Graph of the recorded temperature (blue), predicted temperature (purple) and target temperature (yellow) of Inc-RH during a lid open event*

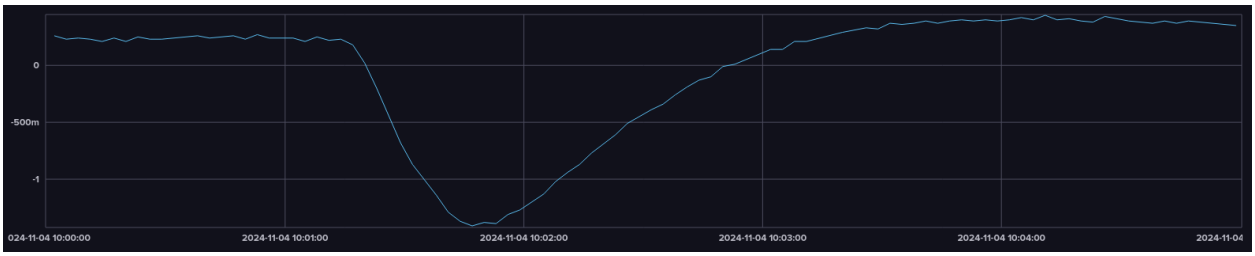

Figure 4.2: *Graph of the prediction error of Inc-RH during a lid open event*

the simulation needed an initial value from the sensor to properly compute the predicted temperature. Since no value existed because this was the first time the node was run, a value of 21 Celsius was used instead. This was more than 1 Celsius above the actual starting temperature, which was around 18.5 Celsius, causing a recalibration. This error was quickly fixed at the start of the second cycle, and caused no issues. The parameters used for the Kalman Filter simulation were also those provided by Aarhus University. Since I have a different insulated box and heat-bed than original Inc-AU build (see Chapter 2), the parameters I am using are likely not correct with respect to my actual hardware [3]. It is likely that running the incubator with proper calibration will reduce the time needed for the system to decide that it needs to update the parameters in the controller.

Figure 4.3: *Graph of the temperature of Inc-RH over a period of several hours*

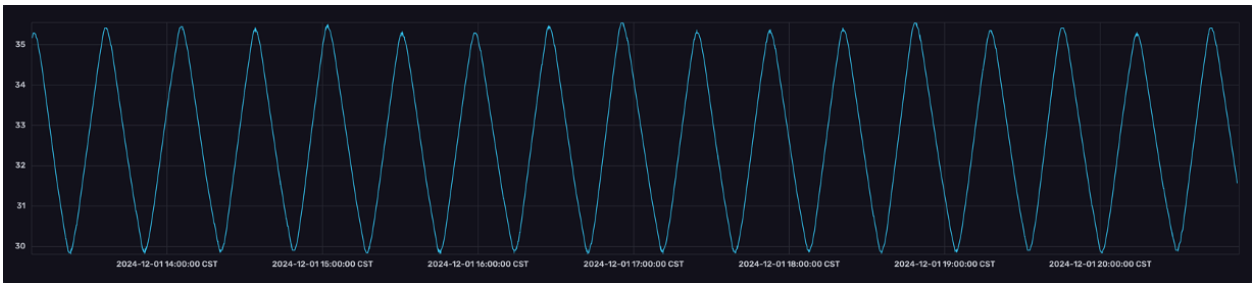

Figure 4.4: *Graph of the temperature of Inc-AU over a period of several hours*

4.2 Implementation Runtime Comparison

To ensure that Inc-RH performs comparably to Inc-AU, I ran a similar experiment using the Inc-AU incubator software. The temperature graph and error graph for the uninterrupted 8 hour period for this experiment can be seen in figures 4.4 and 4.6. Qualitatively, they both look very similar to their counterparts from the Inc-RH, and both have a similar frequency and amplitude. Table 4.2 shows that both implementations perform very similarly, though the Inc-AU spent a marginally higher amount of time out of range, but it is in no way a significant amount. The prediction error also stays bounded in an appropriate range while the lid is closed, meaning no erroneous temperature recalibrations occurred.

4.3 Implementation Code Comparison

A major advantage of using an architecture model in tandem with a code generator is to cut down on development time spent writing and thinking about boilerplate code, as a large

[3]It is worth noting that Inc-AU has a tool to estimate the actual parameters that match the hardware you are running it on. When I tried this tool, it either crashed, or gave parameters that caused the predicted temperature to be much further off than the predictions I was getting with the default settings, so I just used those.

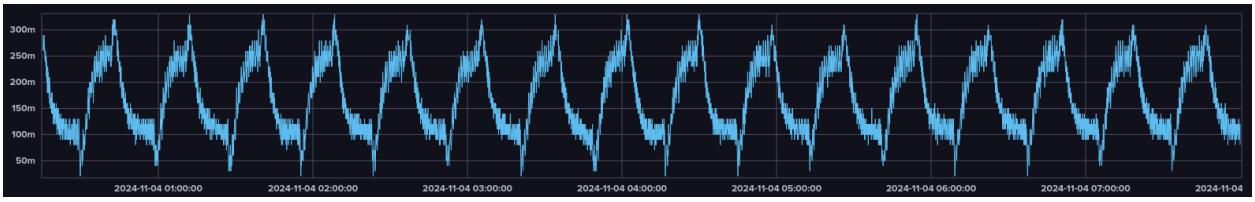

Figure 4.5: *Graph of the error between the predicted and actual temperature of Inc-RH over the same time period as fig. 4.3. Note that 300m = 0.300*

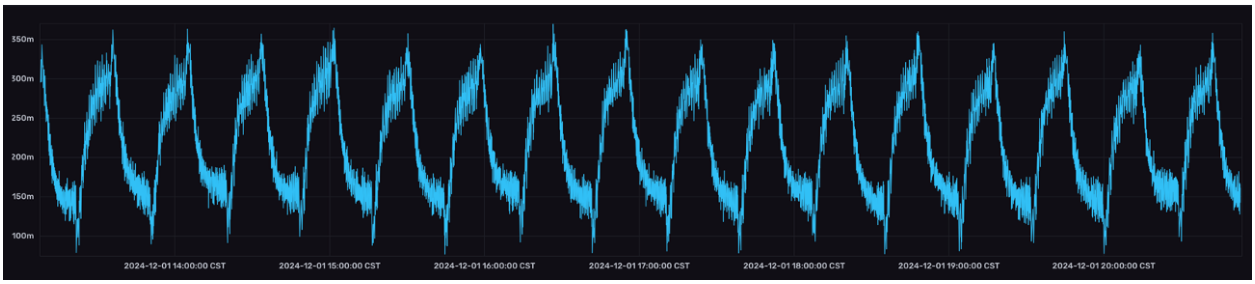

Figure 4.6: *Graph of the error between the predicted and actual temperature in Inc-AU over the same time period as fig. 4.4. Note that 300m = 0.300*

chunk of that will be written for the developer automatically, allowing them to focus on more interesting or challenging aspects of development. Ideally, the ROS2 code generator for HAMR would also give a developer this advantage. To attempt to analyze the efficacy of the ROS2 code generator for HAMR for use in the Digital Twin Incubator in this regard, I will do a comparison on the number of lines that needed to be handwritten in both implementations.

I want to bring up a couple things of note before the comparison. First, since Inc-AU is entirely written in Python, and Inc-RH mostly is written in C++, this comparison is likely not entirely valid, especially given that C++ tends to be more verbose than Python. Second, I would like to clarify that more investigation will need to be performed on the efficacy of HAMR, and model based code generators in general, on the development of Digital Twin Systems. To perform this analysis, I went through the source code of all relevant parts of both implementations and sorted each line into one of three categories: Generated/Boilerplate lines, which are lines that either were generated by a code generator (in the case of Inc-RH) or lines that are focused on setting up the digital twin communication infrastructure (in the case of Inc-AU), Implementation specific lines, which are handwritten lines that implement features that are not present in the other version such as the RVIZ visualization

Table 4.2: *Statistical analysis of the incubator using Inc-AU over approximately 8 hours*

Median temp	32.63
Average temp	32.64
Max temp	35.53
Min temp	29.84
Number of points above 35	1136
Number of points below 30	607
Total Number of points	9300
Proportion of points out of range	0.1874

Table 4.3: *Line counts for Physical Twin Nodes in Inc-AU*

	Device Manager	Physical Controller
Boilerplate Lines	47	29
Implementation Specific Lines	22	9
Node Function Lines	173	116
Total Hand-written Lines	242	154

in Inc-RH or the logging infrastructure set up in Inc-AU, and node function lines, which are the core, algorithmic components in each node that will have strong similarities in both versions. The counts in each category for the physical twin node implementations can be seen in Table 4.3 for Inc-AU, and Table 4.4 for Inc-RH. The counts for the digital twin node implementations can be seen in Table 4.5 for Inc-AU, and Table 4.6 for Inc-RH. In all but one node, Inc-RH has fewer lines of code written by hand. The one node where Inc-RH has fewer lines is the InfluxDB recorder, and that is because the InfluxDB library for Python can write JSON objects directly to the database, while the InfluxDB library for C++ cannot. RabbitMQ can also be set to have one event handler listen on all message channels, while ROS subscriptions can only listen to one. This means that the Inc-AU InfluxDB recorder has one event handler while Inc-RH has 4. The counts listed in each of the tables do not include any code written to facilitate launching the system, as they occur substantially differently in each implementation. Even though Inc-RH has consistently fewer hand-written lines than Inc-AU, this does not immediately imply that using ROS2 for HAMR would speed up development time, or make development substantially easier, especially if the developer is not familiar with C++, as C++ and Python are very different languages. When the Python

Table 4.4: *Line counts for Physical Twin Nodes in Inc-RH*

	Device Manager	Physical Controller
Generated Lines	96	78
Implementation Specific Lines	50	16
Node Function Lines	118	119
Total Hand-written Lines	160	135
Total Lines	264	213

Table 4.5: *Line Counts for Digital Twin Nodes in Inc-AU*

	InfluxDB Recorder	Plant Kalman Filter	Lid Open Manager
Boilerplate Lines	13	16	13
Implementation Specific Lines	3	35	1
Node Function Lines	10	218	10
Total Hand-written Lines	26	269	24

implementation of the ROS code generator for HAMR is complete, a comparison with that and Inc-AU may be warranted.

Table 4.6: *Line Counts for Digital Twin Nodes in Inc-RH*

	InfluxDB Recorder	Plant Kalman Filter	Lid Open Manager
Generated Lines	63	69	64
Implementation Specific Lines	10	12	3
Node Function Lines	50	213	9
Total Hand-written Lines	60	225	12
Total Lines	123	294	76

Chapter 5

Conclusion

In this early examination, the ROS2 code generator for HAMR appears to be a tool that will make the development of digital twin systems more efficient. The tool certainly reduces the number of lines of code that need to be written by a developer when compared to a handwritten implementation, and the AADL model provided a solid abstraction of the different components of both the physical and digital twin systems. ROS2 also proved to be an excellent communications platform, with little auxiliary configuration, and reliable communication between all nodes, even over a network. The tool does have some drawbacks, however. First, the tool can only generate ROS nodes in C++, which limits the language developers can use, meaning it may increase development times for teams unfamiliar with the language. It can also, due to the lack of memory safety, cause issues when trying to ensure that a system is safe and performs properly. ROS2 topics are also insecure, as any ROS node can write to any ROS topic with no authentication, which would allow malicious actors to easily disrupt operation. Due to potential network latency issues, more studies need to be done on how ROS would perform on digital twin systems that need messages between the physical and digital twin to be sent with incredibly fast speed. As outlined in Chapter 4, the code output by the HAMR code generator needed some workarounds to perform correct hardware initialization and interact with wider ROS features.

5.1 Future Work

While the ROS HAMR code generator already works very well, there are a lot of things that can be done to further improve the system. Aside from fixing the issues enumerated above, further integration with existing ROS features could be added to the code generator. This could include support for defining ROS2 parameters in an AADL or SysMLv2 model, allowing them to be included in code generation. Looking into how to give the ability to use the standard, built-in ROS message types as port types in AADL would also be a great addition. Similarly, allowing externally-defined ROS nodes to be represented in the Diagram would be very useful. This would, for instance, allow the Digital Twin Incubator to send properly formatted sensor data to the RVIZ visualizer and have all necessary publishers and subscribers generated along with everything else, instead of doing it manually. ROS also can interact with several simulation and visualization services, such as RVIZ and Gazebo. Since many digital twin services revolve around simulation and visualization, investigation into how they can be used in the development of digital twins may prove useful, as well as investigating if and how these simulation services can be integrated into the HAMR toolchain.

Because we have a formal abstraction of our system system, this gives us the opportunity to investigate how formal methods may be applied to both digital twin systems and ROS2 systems. This could allow us to create stronger guarantees about our systems and ensure stronger system safety and security. use of the Sireum GUMBO contract language and GUMBOX test generation framework for this would naturally be a good fit, as they already integrate with the HAMR slang target.

Bibliography

[1] Hao Feng, Cláudio Gomes, Casper Thule, Kenneth Lausdahl, Alexandros Iosifidis, and Peter Gorm Larsen. Introduction to digital twin engineering. In *2021 Annual Modeling and Simulation Conference (ANNSIM)*, pages 1–12, 2021. doi: 10.23919/ANNSIM52504.2021.9552135.

[2] The INTO-CPS Association. Example digital twin incubator. URL https://github.com/INTO-CPS-Association/example_digital-twin_incubator/tree/master.

[3] Benjamin Thompson. Ros2 digital twin incubator. URL https://github.com/bathompson/MS_Report.

[4] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012. ISBN 978-0321888945.

[5] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[6] John Hatcliff, Jason Belt, Robby, and Todd Carpenter. Hamr: An aadl multi-platform code generation toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, pages 274–295, Cham, 2021. Springer International Publishing. ISBN 978-3-030-89159-6.

[7] Hao Feng, Cláudio Gomes, Casper Thule, Kenneth Lausdahl, Michael Sandberg, and Peter Gorm Larsen. The incubator case study for digital twin engineering, 2021. URL https://arxiv.org/abs/2102.10390.

[8] *DS18S20 High-Precision 1-Wire Digital Thermometer*. Maxim Integrated, 2015. URL https://www.analog.com/media/en/technical-documentation/data-sheets/DS18S20.pdf. Rev. 3.