RespiraWorks Progress Status Report

Introduction

Our mission is to radically democratize the ventilator. We aim to deliver a full-feature ventilator at a dramatically lower price that will be affordable for hospitals anywhere in the world. We publish and maintain our design in accordance with open-source principles.

The Covent Challenge enables us to showcase our second design iteration (v0.2). Notable additions to this version include: pressure assist control mode, touchscreen user interface, improved flow sensing, inspiratory effort detection, an alarm subsystem.

Our next steps are to develop a continuous integration test platform, optimize the hardware for regional manufacturing, and enable further modes of operation - namely pressure support ventilation and high flow oxygen therapy. We aim to refine our oxygen control design, user interface workflow, and increase our sensing capabilities.

Contents:

RespiraWorks Open Source

RespraWorks is an Apache 2-licensed open-source project (both hardware and software), developed in the open.

A number of open-source veterans work on the project, and we take openness seriously, both in our software license and in the way we work as a project.

Choice of License (Permissive vs Copyleft)

For us, the choice of license came down to permissive vs copyleft. We chose a permissive license, and narrowed that down to Apache 2.

A copyleft license like GPL requires that if you change then distribute the software, you have to make the changed source code available. This sounds good, and in many ways it is! But at the risk of taking a position in a religious war, we believe that the interests specifically of COVID-19 are better served by software developed under a more permissive license.

We have found that scrupulous organizations are hesitant to use copyleft (specifically GPL'ed) code. It's easy to accidentally link proprietary code with GPL'ed code, and if that ever escapes into the wild, all of your proprietary code is now covered under the GPL and must be released as open-source. In our experience, many organizations find this to be an unacceptable risk and simply won't work on GPL'ed projects, even if they have every intention of upstreaming all of their changes.

A permissive license doesn't have this problem, because you can take the code and do pretty much whatever you want with it. You don't have to share your changes under the same license.

The perceived risk of this is that an organization might take the open-source code, make many changes to it, and not share those changes with the community -- thus harming the overall project.

In our experience, the risk of this is low. The value of an open-source project is not just in its source code, but also in the community: the people who understand the code and can help. If you make a closed-source fork of the code, the community necessarily *can't* help, even if they wanted. Moreover, maintaining a closed-source fork of a project is *hard*, because as the underlying project changes, you have to carefully merge those changes back into your fork. These changes can be disruptive if they're made without any consideration of the fork.

Ultimately, we decided to use a permissive license because we *want* people to use our work in service of humanity. We hope that if they make substantial changes they will donate their work back to the upstream community, and we think it's in their commercial interests to do so, because we expect they will want to interact meaningfully with the community and because we expect maintaining a fork will be challenging. But the bottom line for us is, **if a third party doesn't want to contribute their changes back to the main project, we would rather that they build ventilators than they don't build ventilators.**

Having chosen to use a permissive license, Apache 2 was an easy choice. It's similar to MIT and BSD in that it doesn't place major restrictions on what you can do with the code, but it adds some protection against malicious use of patents. Specifically, if you own a patent and contribute code to the project, you can't then sue the project for infringing on that patent.

An Open Community

It's possible to run an open-source project where all development happens behind the scenes. Occasionally, a new version appears online, with huge and unexplained changes compared to the previous version.

We strive for the opposite of this. In our project, we strive for all development to happen in a forum accessible to all. Anyone who upholds our community standards is welcome to participate, and indeed we have participants from around the world.

In many ways we think this kind of openness is more important than the choice of open-source license, or even *whether* we have an open-source license. A closed community may be able to distribute nominally open-source software, but fundamentally it can't grow beyond its bounds.

Status of Volume Sensing

We track volume by integrating flow over time. Getting an accurate flow measurement is challenging with our current hardware, especially at low flow rates, where the venturi's SNR is worst.

We currently use a "big hammer" to address this, essentially forcing volume to 0 at each breath boundary. This may hide leaks and other clinically-relevant data from users, and so this algorithm likely needs to be improved in the future, potentially in combination with different or additional sensors.

See additional details in document 01-1 Design Overview Document: flow sensing section.

PS, SIMV and PRVC Ventilator Mode

Currently, the ventilator has a CMV mode, where the device provides commanded breaths at a set rate and with a controlled pressure profile, and a pressure assist mode, where the device triggers breaths by detecting inspiratory effort.

The next modes planned to be added are pressure support (PS), Synchronized intermittent mandatory ventilation (SIMV), and pressure regulated volume control (PRVC).

Pressure support is the most developed mode. Currently, the system is implemented in our Modelica simulation environment, and our pressure assist mode gives us experience detecting patient effort. SIMV and PRVC were originally higher priority modes, but after extensive consultation we determined that most doctors would be most comfortable for most patients using CMV and AC/P, and so we focused on improving those modes first as much as possible.

Status of Oxygen Flow Control

Our ventilator can currently deliver 100% oxygen (supplied by an external source of pressurized oxygen) or ambient air (supplied via our blower fan), but cannot currently mix these two gasses.

Variable FiO2 requires a more complicated control scheme, controlling both the air proportional valve and the oxygen proportional solenoid together. We are currently exploring two controller schemes for achieving this. We've tried both of these in the modelica model, though neither has been tried on hardware yet.

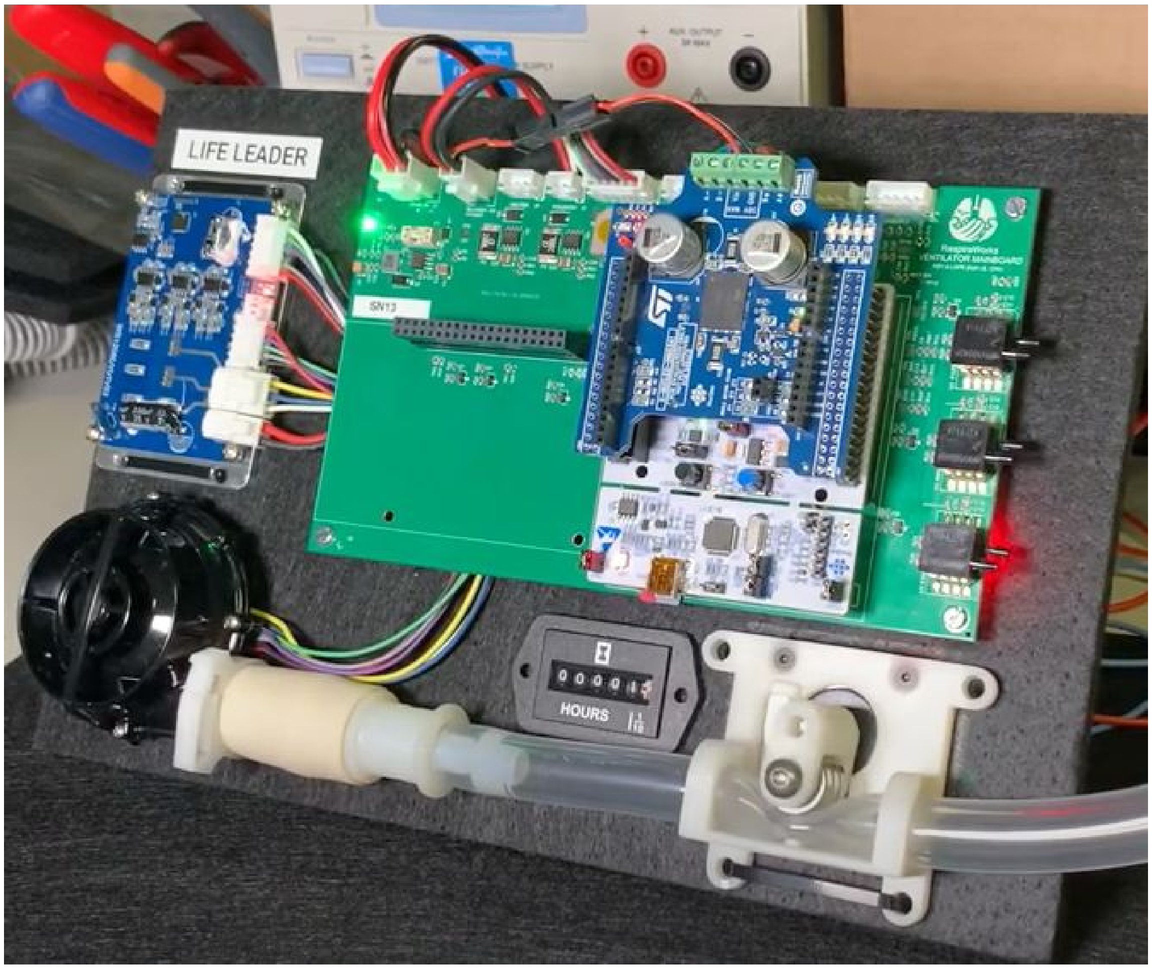
The first controller scheme is to ratio-control the two valves. In this scheme, a single PID commands both the air valve and the oxygen valve simultaneously to achieve the desired pressure targets in the different breathe stages. The mapping from the PID command to the actual valve command is determined by a ratio that is adjusted across breathe cycles by a separate PID controller designed to achieve the desired FiO2.

The challenge with this scheme is that the current valves (the oxygen proportional solenoid and the air proportional pinch valve) have extremely different response times, which may make it difficult to achieve stable control.

An alternative scheme is to use one of the valves to control pressure and the other valve to control FiO2. The valve that is controlling pressure would be controlled by a PID during the inhale and exhale phases. The valve that is controlling FiO2 would have a constant value across inhale and a constant value across exhale - these values would then be adjusted by the FiO2 PID across breaths. For this scheme, if the FiO2 were below about 70-80%, then the air valve would be used to control pressure and the oxygen valve would be used to manage FiO2; for higher FiO2, this would be reversed.

The benefit of this approach is that it avoids trying to control both valves simultaneously. The downside of this approach is that it can lead to greater FiO2 variability over a breath, has a greater risk of pressure over/undershoot, and, for stability reasons, will likely necessitate a slower rate of FiO2 change. Based on discussions with doctors, achieving the necessary FiO2 within 30-60s was deemed acceptable, meaning that a safer control scheme that minimizes over/undershoot risk by adjusting FiO2 more slowly may be acceptable.

Durability and Life-Leader Testing



To validate that the ventilator components are suitable for extended periods of operation, we are setting up automated durability and life-leader tests. These tests will cycle both individual components and assembled ventilators over time, measuring any change in performance. The goal is that these test units will have more cycles on them than any operational ventilators, giving us confidence that the design can withstand continued operation.

Our initial life-leader testing has focused on the custom proportional pinch valve design. As discussed in more detail elsewhere, the pinch valve design results in cyclical stress both on the valve and on the tubing - a key question is how well the design will hold up over time. We have had one pinch valve running continuously for a few weeks and are doing performance testing to see how it compares to fresh valves.

User Manual and Documentation

A user manual is currently being developed with the help of a team of technical writers from Technically Write IT. The manual will be written for the viewpoint of the end user (the medical professional using the device) such that they will be able to use the device from when it's taken out of its packaging through to the end of treatment with a patient. This manual will include descriptions of how to do the following:

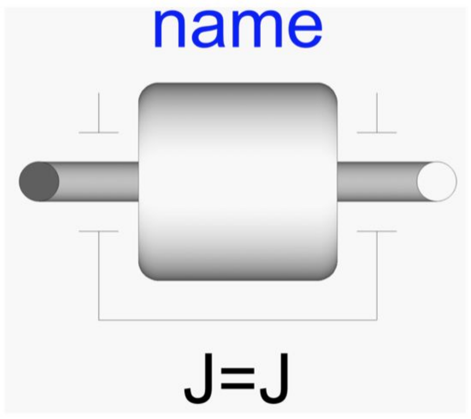
- General
  - Indications for Use
  - Contraindications
  - Target environment and user
  - Warnings
- Device operation, installation, and setup
  - Connecting to power and calibrating the system through the self-test
  - Connecting to patient
- Setting and understanding the system and user interface including:
  - Setting and switching between modes of ventilation (to include Pressure Assist, Pressure Control, and High Flow Nasal Cannula)
  - Setting and changing parameters including
    - PIP
    - PEEP
    - I-time
    - RR
  - Reading and responding to displayed waveforms and alarms
    - Understanding the priority of the alarms
    - Deactivating alarms
    - Addressing the probable causes for these alarms (either the ventilator settings or the mechanical components) that could be responsible for said alarms
- Maintenance recommendations and requirements
  - Instructions for cleaning
  - Disposal of filters and other single-use items
  - Assembly diagrams
  - Instructions for parts replacement
- Shutting the system down

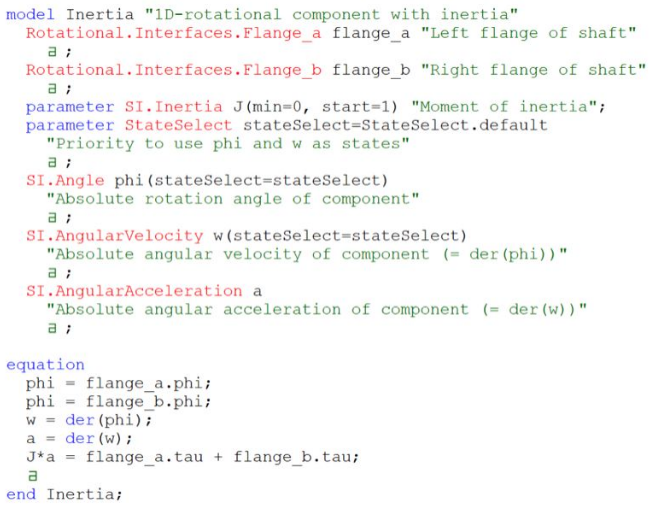
Modelica Simulation Model

Modelica Overview

Modelica [www.modelica.org] is a non-proprietary, object-oriented, equation-based language used to model complex multi-domain physical systems. Developed by the non-profit Modelica Association, the Modelica language is widely used in industry for modeling physical systems with mechanical, electrical, pneumatic, hydraulic, thermal, control, electric power, and process-oriented components. With Modelica, models can be built from the open source Modelica Standard Library (the largest free library for multi-domain models), other open source libraries, commercial libraries, and custom in-house libraries to leverage existing modeling technology while providing the openness and flexibility to incorporate custom modeling IP.

Modelica is used for modeling the dynamic behavior of technical systems. Models are described by differential, algebraic, and discrete equations. Modelica physical models are acausal and allow the creation of reusable primitive components, subsystems, and complete systems. To illustrate a simple Modelica model, the figure below shows the graphical representation and Modelica code for the 1D rotational inertia from the Modelica Standard Library. Modelica models are written as equations, typically in standard text book form, that express relationships between variables. Since Modelica models are descriptive but not necessarily programmatic, a Modelica tool is needed to generate executable code. The entire process starting from Modelica code to a simulation result is shown in the figure below.   A Modelica compiler assembles the Modelica code, performs symbolic manipulation on the equations, generates C code, and compiles the C code into a simulation object that is linked with a numerical solver that executes to generate the simulation result. Both open source and commercial compilers are available for Modelica. With the powerful symbolic processing from a Modelica compiler, modelers are free to focus on developing high quality models for systems engineering rather than numerics and equation solving.

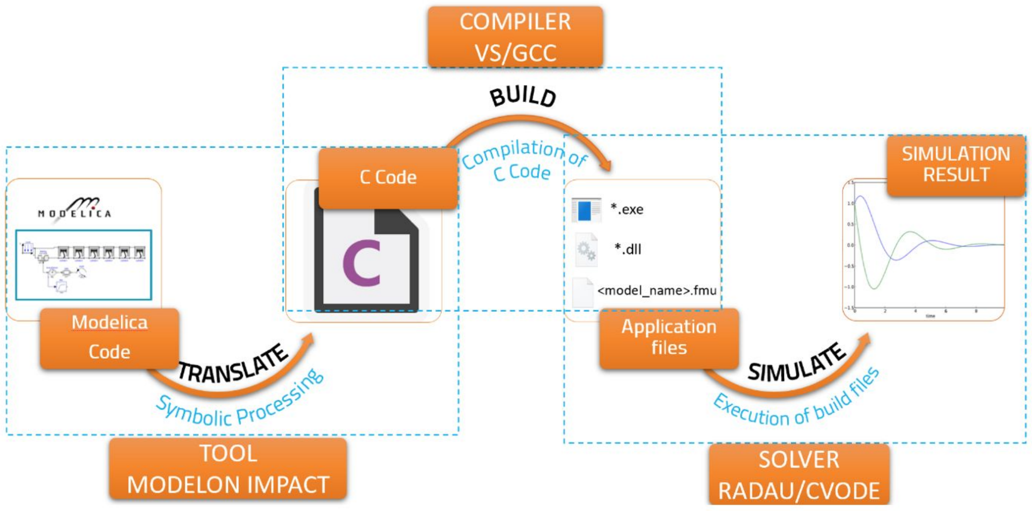




```
model Inertia "1D-rotational component with inertia"
  Rotational.Interfaces.Flange_a flange_a "Left flange of shaft"
    a ;
  Rotational.Interfaces.Flange_b flange_b "Right flange of shaft"
    a ;
  parameter SI.Inertia J(min=0, start=1) "Moment of inertia";
  parameter StateSelect stateSelect=StateSelect.default
    "Priority to use phi and w as states"
    a ;
  SI.Angle phi(stateSelect=stateSelect)
    "Absolute rotation angle of component"
    a ;
  SI.AngularVelocity w(stateSelect=stateSelect)
    "Absolute angular velocity of component (= der(phi))"
    a ;
  SI.AngularAcceleration a
    "Absolute angular acceleration of component (= der(w))"
    a ;

equation
  phi = flange_a.phi;
  phi = flange_b.phi;
  w = der(phi);
  a = der(w);
  J*a = flange_a.tau + flange_b.tau;
  a
end Inertia;
```

Modelica model for rotational inertia



Model process for Modelica

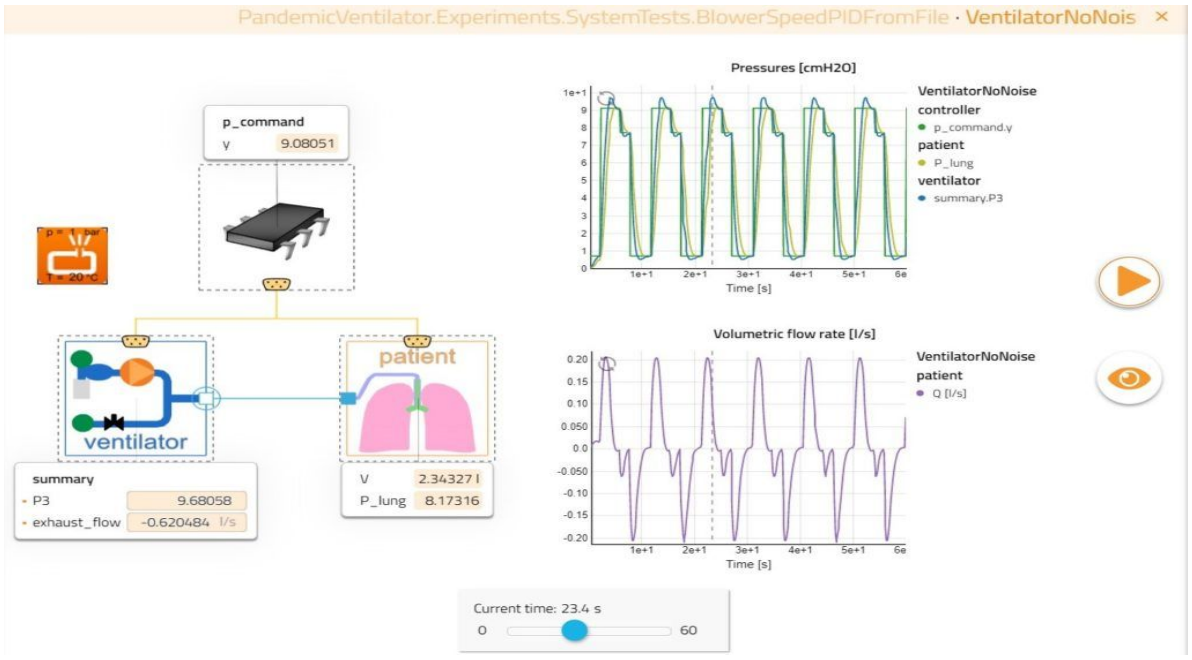
Model Overview

For the ventilator work, we partnered with Modelon, who provided licenses for a beta version of their commercial Modelica tool Modelon Impact [https://www.modelon.com/modelon-impact/]. Modelon Impact is a fully featured Modelica development and simulation environment integrated into a web browser. Modelon Impact is designed to enable system simulation beyond just CAE experts and facilitate collaboration and model-based engineering.

Modelon helped us to develop a package of models to support the ventilator physical system and controls development. These models are based on Modelon's Pneumatics Library, a commercial library for modeling pneumatic systems. Modelon continues to develop the ventilator modeling package in partnership with RespiraWorks.

As shown in the figure below, there are three main parts of the model: the controller, the ventilator, and the patient. The beauty of modelon impact is that it lets you easily swap in different instances of each of the components and see how that affects your results. Below we'll walk through how each of these were created.

Screenshot of Modelon Impact Showing The Model GUI



p_command
y 9.08051

summary
P3 9.68058
exhaust_flow -0.620484 l/s

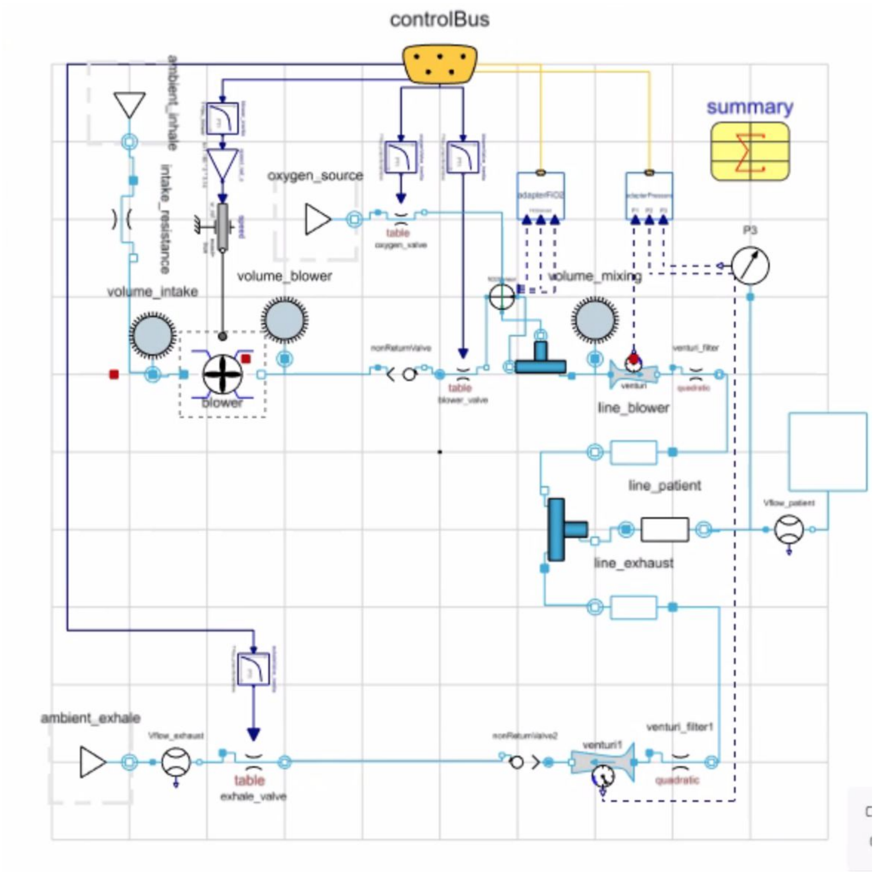
V 2.343271
P_lung 8.17316

Current time: 23.4 s
0 60

Patient

There are currently three main different patient models. The first is a basic model that allows the user to specify a resistance and compliance and then calculates the internal volume and pressure based on pressure and flow rate from the ventilator. The second is a breathing model that allows a respiratory rate, IE ratio, and a musculatory pressure to be specified, controlling both how quickly and how forcefully the simulated patient breathes. There is also a more advanced lung model that incorporates more parameters to enable more complex simulations, though we have not used this model extensively yet.

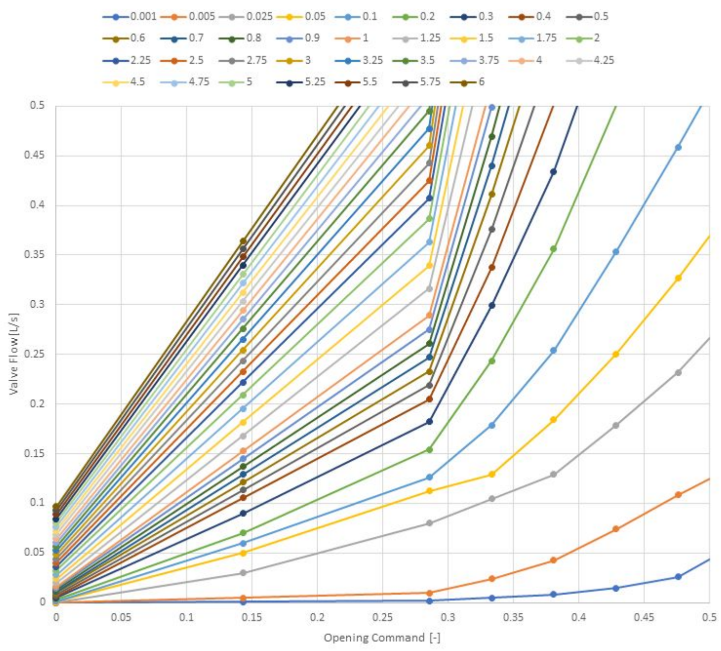
Ventilator



Current Ventilator Model in Modelica

As the ventilator design has changed, there have been many iterations of the ventilator model in modelica. The most recent one is shown in the figure below, where you can see the oxygen leg with oxygen source and ox valve, the air inhale branch with blower and pinch valve, the patient circuit with venturis before and after, and the exhale leg with the exhale pinch valve. The control signals from the controller enter through the control bus at the top and are then routed to the actuators (blower and 3x valves), and the pressure measurements are passed back to the controller through the same control interface.

Each component is built up using the modelica language. The components in this ventilator model are a combination of default modelica components and components created specifically for our ventilator. The blower component, the proportional pinch valve components, and the oxygen valve components are based on test data and data sheets for the actual hardware we are using. An example of the flow curves inside the modelica model for the custom proportional pinch valves are shown below. Similar curves were created for the blower and the oxygen valve.



Flow Curves for Custom Proportional Pinch Valves Used Inside Modelica, Based on Flow Testing. Valve flow vs opening command with different curves for DP across the valve.

Controller

We explored two different ways of implementing a controller in Modelica. The first was to build the controller in modelica, using components based in the modelica language. The second is to write a controller outside of modelica (e.g. in C++) and to then link it through the modelica interface. The first approach allows rapid interaction for those who are less familiar with how to implement controllers in software; this allowed us to prototype different controller configurations before we actually had the hardware set up to test them with the real system. The second approach could be useful for testing the controller software without being constrained by hardware, something that is discussed in the next section.

Executing The Model

When the model is run, the various components begin in their initial state. The tool then steps forward in time, calculating both the commands from the controller and the corresponding response of the system at each time step, solving the various underlying equations to calculate pressures, flow rates, etc. The results can then be visualized to see how the system performed over time.

Comparing Model to Hardware

Currently, we are working to verify that the model matches what we see with hardware. This has been challenging for a few key reasons:

- Rapidly changing controller code that hasn't been integrated into the modelica model. This makes it difficult to tell if differences between results from the model and the hardware are because of issues in how the hardware is modeled, or differences in the commands coming from the controller, which affects the system response. Unfortunately, the structure of the controller code has also been changing, which means that this isn't completely solved by using the external controller.
- Rapidly changing hardware that has not been completely characterized. As we iterate on the hardware design, we have been changing components faster than we've been able to collect characterization data with which to anchor our modelica models.

Moving forward, we are evaluating ways to anchor the modelica model to give us more confidence in its predictive capabilities.

Status of SiL (Software-in-the-Loop) Testing

Software-in-the-loop testing is a way of testing the controller software with the modelica model such that software development is not held up by access to hardware (testing on actual hardware would be HIL (hardware-in-the-loop)). For a distributed team that may not have access to hardware, this can be a key way to allow the team to move quickly. Additionally, modelica models of new components can be developed based on their data sheets before they are acquired, allowing us to explore designs both cheaper (can evaluate components without buying them) and faster (can start updating the software before the new hardware arrives).

We have built out the infrastructure for interfacing our controller code with modelica and tested early prototypes. We are currently evaluating how best to integrate our existing controller code. Unfortunately, modelica relies on an earlier version of Visual Studio to compile the external controller for it to interact with, which means it does not support all of the C++ features that we are using. How best to work around this limitation without adding so much overhead that SIL testing becomes more work than it is worth is still to be determined.

Status of Continuous Integration

We have the following automatic or automated testing and continuous integration infrastructure currently set up:

- Cycle Controller and **(New in v0.2)** UI Controller unit tests being run on each commit. Only commits that pass tests can be merged. We use CircleCI, a popular continuous integration service.
- **(New in v0.2)** Hardware-in-the-loop unit tests that developers can trigger manually using a shared server connected to a development build of the device or can run against their personal device.
- **(New in v0.2)** The Cycle Controller debugger tool allows automating hardware-in-the-loop tests and was used to collect data for CoVent testing scenarios presented in 02-1 Performance Evaluation Report.
- **(New in v0.2)** Continuous building of UI Controller Docker images, which also uses CircleCI.

Our near term plans also include the following:

- Reporting of code coverage of unit tests
- Optimizing GUI deployment with a loose coupling strategy to minimize update and development overhead:
  - Splitting the GUI dependencies from the GUI code in the Docker container*
  - Switching to Debian package management system
- Unit tests for communications between the GUI and controller
- Automate generation of customized Raspbian OS images to run on Raspberry Pi's with the most up-to-date GUI Docker image preloaded*
- Automate data analysis from unit tests to provide insight from test data

* See *Building UI Controller Docker images*

Hardware-in-the-loop testing

We are beginning to build infrastructure for automated hardware-in-the-loop tests.

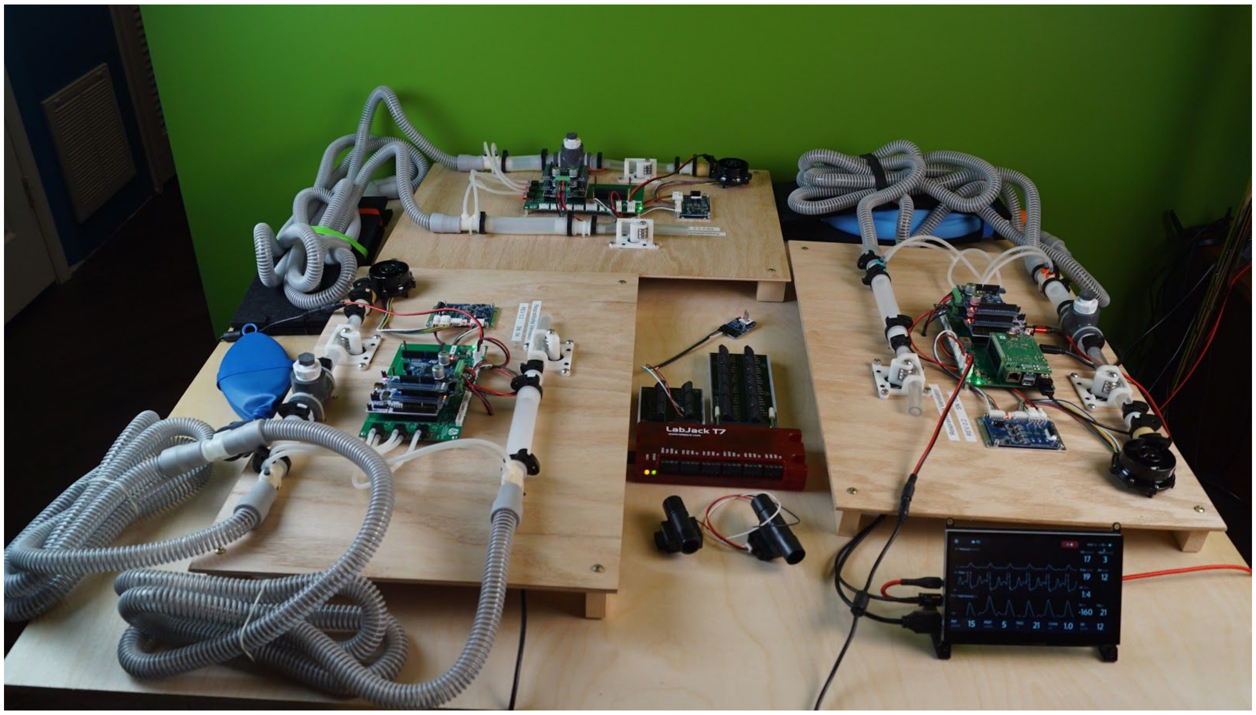
Hardware-in-the-loop unit tests

The goal of these tests is to test some subsets of our system with hardware on a regular basis and in a systematic way, to increase the likelihood of catching at least some problems early. These tests consider some cross-section of the hardware-software stack. To avoid the "fox watching the hen-house" situation, we have aimed at using an alternative hardware-software stack to confirm the performance of our system.

We are in the process of procuring all necessary components for this testing infrastructure, as well as building the automation software, however some of it is already available for developers. In particular:

- We have set up a server connected to a build of the ventilator, which developers can access over ssh
- We have written several tests that can be run against the device to confirm the functionality of the hardware abstraction layer of our software.
- Multiple independent sensors have been procured to help confirm our code for pressure and flow sensing.
- Modified pinch valves have been designed with additional sensors to confirm proper calibration and homing routines.
- A Labjack data acquisition system has been obtained that will allow us to acquire validation data in an automated fashion.
- The test units are powered via USB-controlled relays, allowing developers to remotely power cycle the test units.
- A Jenkins CI server has been set up that will allow us to run all of these tests on a regular basis, same as software unit tests done in CircleCI.

This testing lab will remain functional for the duration of further ventilator design through final product deployment and beyond, while any software development is taking place.



Some of the developed test infrastructure will likely inform the development of validation and testing suites for the manufacturers of the final product.

More information about this infrastructure is available in the repository.

System tests using the scriptable debug interface

Additionally, we have built a debug interface into the device, and a Python-based command-line tool that can communicate with the device over that interface. The tool can be attached to the device even when the GUI is running.

Normally the debug tool is used simply for debugging (e.g. retrieving or overwriting values of certain system state variables or parameters, or collecting data traces), however it is scriptable and is also used for automating testing scenarios. As of writing, we have used the tool for automating the collection of test data for CoVent testing scenarios. We plan to expand its use to more automated tests and apply it as part of the shared hardware-in-the-loop testing infrastructure described above.

More information about the debug interface is available here.

Building UI Controller OS and Docker images

The UI runs on the Raspberry Pi. The Pi is deployed with a custom Raspbian OS image, and the UI runs inside a Docker container on this OS. The UI controller runs inside a Docker container on the Raspbian OS on a Raspberry Pi in order to simplify dependency management.

Creation of the image and the Docker container are both automated.

Docker image creation

One of our CircleCI workflows continuously builds and publishes an up-to-date Docker container version from the latest UI controller code on GitHub to Docker Hub. This is fully automatic.

Updated UI Docker images are not *automatically* picked up by the device in order to minimize risk of updating from a working version to a buggy version in a field setting. However, updating them manually is easy, making it convenient for UI Controller developers to test the latest version of the code on device.

Developers who wish to test local changes to UI code can run the Dockerfile scripts that the CircleCI workflow builds with and transfer the resulting UI Docker image manually to the Raspberry Pi.

OS image creation

The Raspbian OS image that's flashed on the Raspberry Pi is generated with a combination of QEMU and Ansible, which together perform configuration steps such as hardware configuration (screen resolution and serial port), installation of dependencies (Docker) and setting the GUI to run on startup.

QEMU runs the raw Raspbian image in a virtual machine while the Ansible playbook script connects to it and runs configuration commands. The Ansible script also supports connecting to a Raspberry Pi connected over the network so developers can install necessary dependencies remotely.

This is currently only partially automated, i.e. creating an OS image is easy for a developer to do manually via these scripts, but we have not yet set up continuous creation of OS images.