Utilizing Code Generation from Models for Electric Aircraft Motor Controller Flight Software

John M. Maroli*, Brian A. Morris†, and Julie A. Blystone‡
NASA Glenn Research Center, Cleveland, Ohio, 44135

Andrew M. Oconnor§
HX5 LLC, Brook Park, Ohio, 44142

The processes followed to develop and test safety-critical flight software significantly increase development effort and overhead in comparison to both non-safety-critical and non-flight software. Generating code from models can be used to reduce development effort; however, the processes for using code generation to develop flight software are relatively immature. This work presents an approach for developing safety-critical flight software utilizing code generation from models to reduce development time and effort. The presented use-case is of electric aircraft motor control software developed for NASA's X-57 Maxwell aircraft.

I. Introduction

The process requirements necessary for developing safety-critical flight software lead to increased development time and effort when compared to what is considered functionally equivalent software that does not meet safety-critical flight software standards. There is cost benefit to reducing software development effort; however, standards cannot be relaxed for safety-critical flight software. Code generation can be used to reduce development effort for certain software components; however, the processes for developing safety-critical flight software from generated code are relatively immature. In this work, electric motor control software is developed using code generation from a Simulink model, and processes are established for developing the resulting software to meet NASA safety-critical flight software standards. The process used by the HLMC software team is presented so that it can be utilized and expanded upon in the future.

The use-case presented in this work is electric motor control software developed for the High Lift Motor Controller (HLMC) on the Mod IV design of NASA's X-57 Maxwell aircraft. The Mod IV design contains 12 High Lift Motors (HLM) distributed across the aircraft wing to demonstrate distributed electric propulsion. Each HLM is paired with an HLMC to control it, and each HLMC is a power inverter and controller for driving the HLM. The motor control algorithm is complex, involving nested control loops and many states. Requirements on high power density also demanded tight cooperation between a multidisciplinary team of electrical, control, software, aerospace, thermal, and mechanical engineers. Generating code from a visual Simulink model not only reduced the software effort needed to develop the complex algorithm, but it also increased cooperation between the software team and other disciplines.

The first step required for the use of code generation is the input model, which should be developed with a simulation for functional validation prior to generating code. The code generation model and corresponding simulation development are discussed in Section II. The process requirements on the final safety-critical flight software are presented in Section III, and the tools used on the model and generated code to verify that those requirements are met are presented in Section V. The work is concluded in Section VI.

II. Creating a Model for Code Generation

The code for the motor controller software is generated from a Simulink model, referred to as the code generation model in this work. The code generation model has a corresponding simulation, referred to as the simulation model. The purpose of the code generation model is simply to generate code for the target processor hardware, while the purpose of the simulation model is to validate the function of the software within the system prior to generating code. The primary inputs and outputs of the code generation model are interfaces to hardware. The inputs are Analog-to-Digital

*Computer Engineer, Flight Software Branch, john.m.maroli@nasa.gov
†Aerospace Engineer, Quality Engineering and Assurance Branch, brian.a.morris@nasa.gov
‡Computer Engineer, Flight Software Branch, julie.a.blystone@nasa.gov
§Software Engineer, HX5 LLC Software Branch, andrew.m.oconnor@nasa.gov

converter (ADC) measurements, digital inputs from aircraft cockpit switches, and commands from an Ethernet network. The outputs are Pulse Width Modulation signals to control power switches, digital outputs to the aircraft cockpit, and telemetry over Ethernet. The motor controller algorithm itself is packaged into a single Simulink subsystem reference block within the code generation model, and the governing state machine is packaged into another subsystem reference block. The motor control algorithm is known as field-oriented control (FOC), so the block is simply referred to FOC. The governing state machine block is called the State Controller. These blocks require inputs from the hardware to operate, making the code generation model a container for the controller blocks that interface them with the hardware. The code generation model is shown in Figure 1. Note that some code is still manually written; the goal of code generation is to produce code for the most complex parts of the software.

The motor controller algorithm can be referenced outside of the code generation model since the subsystem blocks exist in standalone files. A full system simulation of the HLMC connected to the HLM is created containing simulated power electronics, a simulated motor, and subsystem reference blocks pointing to the motor controller algorithm and governing state machine. The subsystem reference blocks in the simulation model link to the exact subsystems that are generated into code in the code generation model. In this way, the behavior of the controller can be validated in the simulation model before creating code through the code generation model. Execution paths of the motor controller block can also be exercised in simulation, which is relevant later in this work. Changes to the motor controller algorithm in simulation propagate to the generated code since the code generation model references the same subsystem files. The simulation model is shown in Figure 2.

The control algorithm for HLMC was developed entirely in simulation prior to generating code. Only controller models that successfully operated in simulation within safety margins were used to generate code. The team found that this significantly reduced hardware failures during software testing in comparison to both manual software development and development using code generation without full hardware simulation. Once code was generated and tested on the hardware, results were used to improve the simulation fidelity, which in turn was used in further development of the controller. This iterative process resulted in a gradual optimization of both the controller software and simulation.

The controller operated at a control frequency of 11.5kHz, meaning it executed the full control algorithm approximately every 87us. This was true in both hardware and simulation. The hardware reported telemetry from every control loop, so a direct comparison between the hardware state and simulation state could be observed after each control cycle. The simulation operated at much higher fidelity to simulate the power electronics and motor, however the controller still updated at the designated control frequency. The simulation could therefore capture aliasing effects present in the real system. Figure 3 and Figure 4 show the lab and simulation results respectively for full-power operation of the HLMC with the HLM and High Lift Propeller (HLP). The scenario is startup of the HLMC with a command to 5000 RPM. After 10 seconds of steady state, the HLMC spins the motor down again. The high current periods at the start and end of spinning are present from the sensorless startup algorithm. After many iterative rounds of optimization, the real controller met performance requirements and the simulation controller matched the observed results. Though noise is notably absent in the simulated system, the average signal levels match well.

III. Requirements of Flight Software

NASA requirements for flight software are contained in the NASA Software Engineering Requirements (NPR 7150.2 [1]) and NASA Software Assurance and Software Safety Standard (NASA-STD-8739.8 [2]). Specific NASA Center software considerations for aeronautics are included in Center-level requirements and specifications. The flight software requirements of focus in this work are Modified Condition and Decision Coverage (MC/DC) and cyclomatic complexity. Code analysis requirements for detecting defects and coding errors are also addressed. Effort to meet these requirements is distributed across both the model and generated code levels.

The software team also examined regulations external to NASA requirements and standards to inform the software development process. A primary goal for the team was to substantiate that the software is safe to fly with confidence. This included applying 14 CFR 35.23 Propeller Control System requirements for software that controlled, limited, or monitored HLMC subsystem functions. It also included applying electronic propeller control system embedded software requirements for design and implementation for critical functions that reduce software errors or defects. Implementing 14 CFR 35.15 Safety Analysis requirements was suggested at a subsystem level, including analyzing the propeller control system and subsystem effects on the vehicle such as uncontrollable torque or speed fluctuation which could have been further augmented from a hazard analysis. Applying 14 CFR Part 35 Subpart C – Tests and Inspections was relevant for examining potential flight scenarios; these include 35.36 bird impact, 35.41 overspeed and overtorque, 35.35 centrifugal load tests, 35.39 endurance test, and 35.40 functional test among others. The 14 CFR 35.5 Propeller Ratings

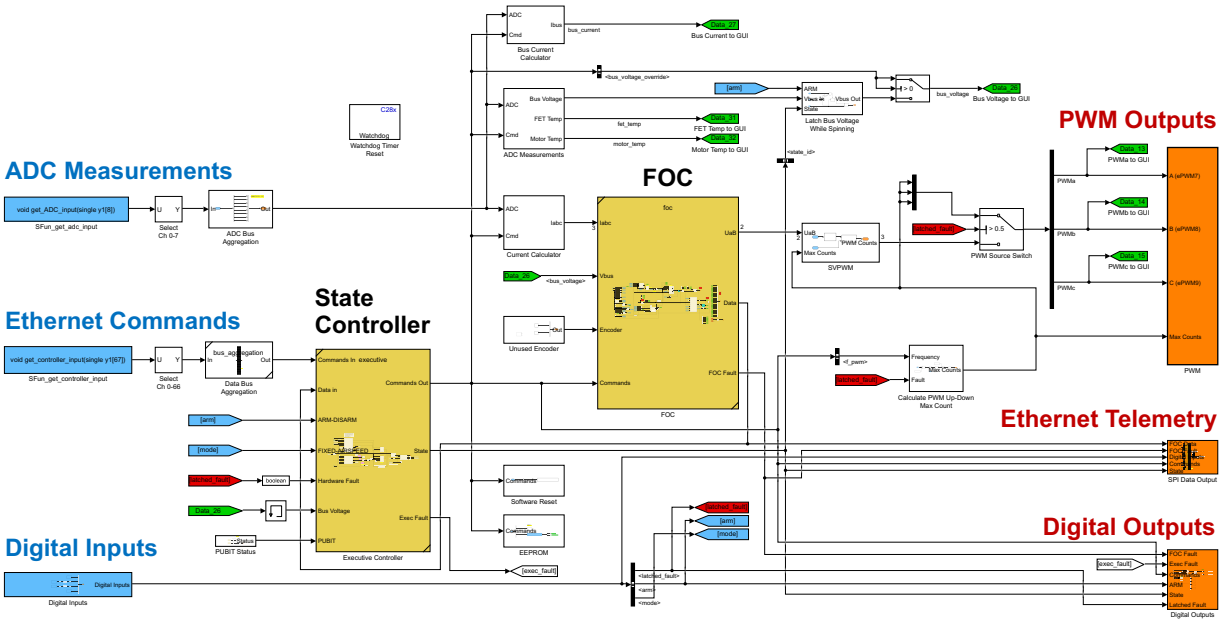

Fig. 1 Simulink code generation model. Blue components are input sources from hardware, orange components are hardware outputs, and yellow components make up the motor control algorithm. The controller components are shared between the code generation and simulation models.

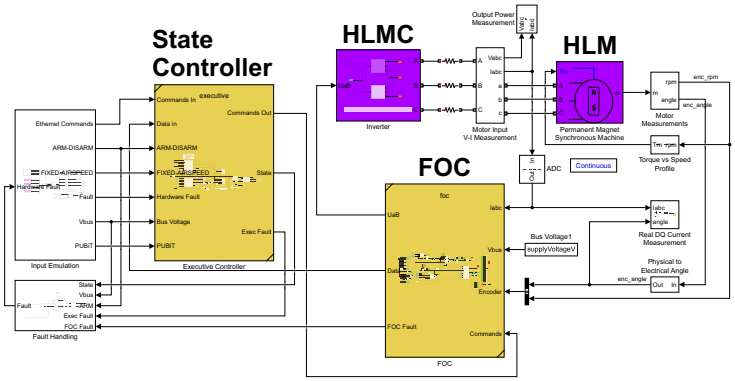

Fig. 2 Simulink simulation model. Purple components simulate hardware and yellow components make up the motor control algorithm. The controller components are shared between the code generation and simulation models.

and Operating Limits were considered for application such as power and rotational speed for takeoff and maximum continuous [3]. These 14 CFR Part 35 requirements informed the HLMC control and performance requirements and led to the inclusion of software enforced speed and torque limits.

The software development was ultimately governed by NASA requirements over external requirements. The team addressed and implemented the NASA Software Engineering Requirements, NASA Software Assurance and Software Safety requirements, and Center-level requirements. The MC/DC requirement is 100% coverage for safety-critical software components at the time of this writing per NPR7150.2D and NASA-STD-8739.8B; however, it was not explicitly required for the HLMC software since it was developed under the prior versions of these standards (NPR7150.2C and NASA-STD-8739.8A). Full MC/DC coverage is important for safety-critical flight software as evidenced by its inclusion in the updated NASA standards. It is also present in the Radio Technical Commission for Aeronautics DO-178C standard [4]. The NASA requirement for cyclomatic complexity is <= 15 for safety-critical software components per NPR7150.2D and NASA-STD-8739.8B; however, the definition of software component is left to interpretation. This

leaves ambiguity when drawing component borders for cyclomatic complexity analysis, particularly in systems with many layers of nested components. Future clarification on NASA's software component definitions is encouraged.

IV. Analysis Process and Associated Tools

Software plans were created to identify useful tools that could help the team meet NASA software requirements for safety-critical flight software at the model, generated code, and manually written code levels. Simulink Design Verifier, Simulink Coverage, Simulink Check, Polyspace Bug Finder, and Polyspace Code Prover are the primary tools that the team identified to use to comply with the safety-critical flight software requirements. Simulink Design Verifier, Simulink Coverage, and Simulink Check are used at the model level, while Polyspace Bug Finder and Polyspace Code Prover are used at the code level. Software design analysis was performed periodically throughout development as opposed to after completion of development to parallelize the analysis effort.

Simulink Design Verifier™ (SDV) identifies model design errors like dead logic, division by zero, out-of-bounds array access, integer overflow, non-finite and Not a Number floating-point values, subnormal floating-point values, specified minimum and maximum value violations, and data store access violations [5]. A simulation case can be created for each design error. SDV is capable of generating test cases for model coverage that can drive the model to satisfy condition, decision, MC/DC, and custom coverage objectives. The team primarily used SDV for design error detection at the model level prior to each release, analyzing the simulation model at the top-level and individually analyzing important lower level components. Both the State Controller and FOC blocks were analyzed, and some of their components were analyzed individually in more targeted simulations. Since the State Controller and FOC blocks are subsystem reference files, resolving errors found in them in the simulation environment also resolves those in the code generation model.

Simulink Coverage™ can perform model coverage analysis to measure testing completeness in models and generated code[6]. The team primarily used the tool for model coverage metrics at the MC/DC structural coverage level and to capture cyclomatic complexity metrics for the different model components (particularly the State Controller, FOC, and their various components). The cyclomatic complexity metric was monitored throughout software builds to indicate whether any safety-critical software components or models had a cyclomatic or model complexity value greater than 15. The cyclomatic complexity at the model level, or model complexity, typically is different than the cyclomatic complexity for the generated code produced from the models. A different tool was used to assess cyclomatic complexity in the generated code. The model complexity was used to further look at areas that may have higher aggregated or local complexities and explore opportunities for improvement related to model structure or model components. The MC/DC was used to identify areas of the model that need further test coverage and to modify the simulation appropriately to increase coverage. Some component blocks required separate simulations to increase coverage. These often overlapped with the more targeted simulations used for analysis of certain components by Design Verifier.

Simulink Check™ provides a way for checking compliance with various modelling standards[7]. The HLMC team used Simulink Check throughout various builds to check warnings associated with modelling standards such as those for DO-178/331 [4]/[8]. Simulink Check expands the capabilities of the Simulink Model Advisor and includes the addition of Model Metrics such as model cyclomatic complexity. Notable for this work, the default model complexity threshold is set to 30 for identification of high complexity components, which is higher than the NASA requirement of <= 15. The stated reasoning for this is that the graphical modelling in Simulink allows managing complex algorithms better than manually written code [9].

Polyspace Bug Finder™ detects defects and runtime issues in C and C++ software [10]. Polyspace Bug Finder is capable of checking and capturing coding standard rule violations and coding metrics like cyclomatic complexity. The HLMC team used Polyspace Bug Finder primarily to identify, track, assess, and resolve defects in code periodically throughout the development process. Results were evaluated to determine whether a detected design defect, error, or violation was a true defect or not. A categorizing and summarizing process was established with the development team to collaboratively specify defect or error severity and categorize each defect or error by impact. The high and medium impact results from the tool were discussed with the software team for disposition and course of action. This helped the team identify any high impact defects early in both generated and source code files created by the team. No true defects were identified in the generated code throughout development, so the results were primarily used for improvement of the developer code. This tool was also used to assess cyclomatic complexity for source files created by the team and for the generated code. The generated code cyclomatic complexity was compared to cyclomatic complexity at the model level, though the two were found to be identical or generally similar.

Polyspace Code Prover™ is a static analysis tool that proves the absence of divide-by-zero, overflow, and other

runtime errors in C and C++ source code [11]. The tool can be used for both autogenerated code and handwritten code. The team primarily used the tool for handwritten code since no errors were found in generated code throughout development. Polyspace Code Prover was used after all Polyspace Bug Finder defects were addressed. It was necessary to first specify all files of the code to be analyzed, which is a non-trivial effort with a large number of header files, libraries, functions, etc. (which is also a requirement for Bug Finder). After running the analysis, all gray checks associated with a source file are resolved with the team, followed by any red or orange checks. As with the Polyspace Bug Finder results, the results are either justified or a plan is established to correct the code.

V. Analysis Results

The purpose of this section is to provide examples of how the analysis tools helped improve the software throughout development. Two software versions referred to as Build 2 and Build 3 are referenced in this section. Build 2 represents partially complete software that is functional but does not implement all requirements, and Build 3 represents software that fully implements all requirements. The software analysis tools were not applied until after completion of Build 2 software. Build 3 software did not undergo targeted optimization for cyclomatic complexity and MC/DC, however it was developed from Build 2 with periodic complexity and MC/DC analysis.

Simulink Design Verifier was the tool found to be most immediately useful by the development team since it operated at the model level where active development occurred. For example, Design Verifier identified that a divide by zero error could occur in a division block and demonstrated the error with values given by a test case. The error was precisely traced and highlighted red in the model. The design error was corrected, and the tool proved that the divide by zero error could no longer occur.

Some of the larger blocks such as the State Controller are only partially compatible for design error detection, with unsupported components stubbed during the analysis. This results in an incomplete analysis, but still yields useful results. Critical portions of partially compatible blocks can be proven to be free of design errors, as was the case for the sensorless motor start algorithm (referred to as Autostart) contained in the State Controller block. This algorithm was a critical component for spinning up and spinning down the motor. Additionally, it had high cyclomatic complexity, so proving it was free of design errors was very important.

While Simulink Design Verifier is helpful for finding design errors at the model level, Polyspace Bug Finder is used for identifying errors at the code level. Both the generated code and manual code were analyzed throughout development, and the manual code was found to consistently have more defects even though the majority of code was generated. There were 23 defects (5 high, 13 medium, 5 low impact) found in the manual code but only 7 defects (1 high, 6 low impact) in the generated code for Build 3 despite 82.75% of the source lines of code being generated. There is no fair comparison between Build 2 and 3 results in this scenario since the analysis scope increased between the versions.

Simulink Coverage assisted the team in identifying and tracking high cyclomatic complexity model components. Autostart was the highest complexity block in both Builds 2 and 3. The block is divided into a component handling motor start and another handling motor stop. The motor start component complexity was reduced from 113 to 14 between Builds 2 and 3, while the motor stop component complexity was reduced from 43 to 15.

Simulink Coverage also assisted the team in identifying if certain high priority model components within the State Controller and FOC met MC/DC objectives in simulation. An early version of the Autostart model component within the State Controller had only 20% of the MC/DC objectives met. The MC/DC increased to 73% in the Build 3 release. MC/DC for the FOC block improved from 48% to 59% between Builds 2 and 3. Each time MC/DC analysis was performed by software assurance, efforts were made by the development team to increase coverage. It should be emphasized that the goal during this time was requirements implementation, not increasing coverage. Periodic analysis of software components during development led to increased coverage before coverage became a team focus. Build 4 is the planned software version containing cyclomatic complexity and MC/DC optimization of Build 3.

VI. Conclusion

Utilizing code generation can reduce software development effort, though the use of code generation in safety-critical flight software is relatively new. This work has presented the process for leveraging code generation for flight software adhering to NASA standards and requirements, and has presented some external requirements as well for reference. The use of presented analysis tools is critical to substantiate the resultant software. Results of the tools that indicate design errors, coding errors, run-time errors, dynamic code issues, model errors, cyclomatic complexity, and test coverage are useful in identifying and improving model and code complexity, test coverage, and issues throughout development. The

tool outputs substantiate the progress and improvements of maturing software and provide a path towards developing safety-critical flight software.

Acknowledgments

This work was funded under the Flight Demonstrations and Capabilities project as part of the development of the X-57 Maxwell aircraft.

References

[1] "NASA Software Engineering Requirements, NPR 7150.2D," , 2022. URL https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=main, last accessed 26 October 2022.

[2] "Software Assurance and Software Safety Standard, NASA-STD-8739.8B," , 2022. URL https://standards.nasa.gov/standard/NASA/NASA-STD-87398, last accessed 26 October 2022.

[3] "14 Code of Federal Regulations (CFR) Part 35 - Airworthiness Standards: Propellers," , 2022. URL https://www.ecfr.gov/current/title-14/chapter-I/subchapter-C/part-35, last accessed 25 October 2022.

[4] RTCA, *DO-178C - Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[5] "Simulink Design Verifier," , 2023. URL https://www.mathworks.com/products/simulink-design-verifier.html, last accessed 17 April 2023.

[6] "Simulink Coverage," , 2023. URL https://www.mathworks.com/products/simulink-coverage.html, last accessed 17 April 2023.

[7] "Simulink Check," , 2023. URL https://www.mathworks.com/products/simulink-check.html, last accessed 17 April 2023.

[8] RTCA, *DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A*, 2011.

[9] "Compare Model Complexity and Code Complexity Metrics," , 2023. URL https://www.mathworks.com/help/slcheck/ug/compare-model-complexity-and-code-complexity.html, last accessed 17 April 2023.

[10] "Polyspace Bug Finder," , 2023. URL https://www.mathworks.com/products/polyspace-bug-finder.html, last accessed 17 April 2023.

[11] "Polyspace Code Prover," , 2023. URL https://www.mathworks.com/products/polyspace-code-prover.html, last accessed 17 April 2023.

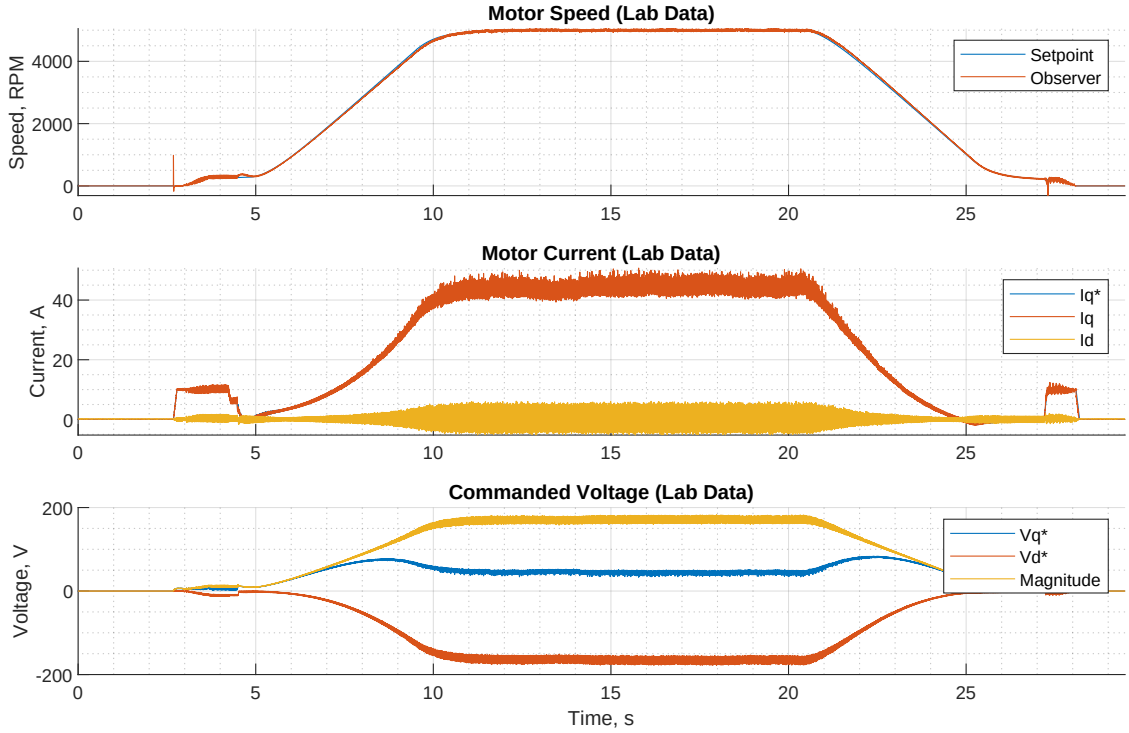

Fig. 3 Lab data from the HLMC with HLM and HLP.

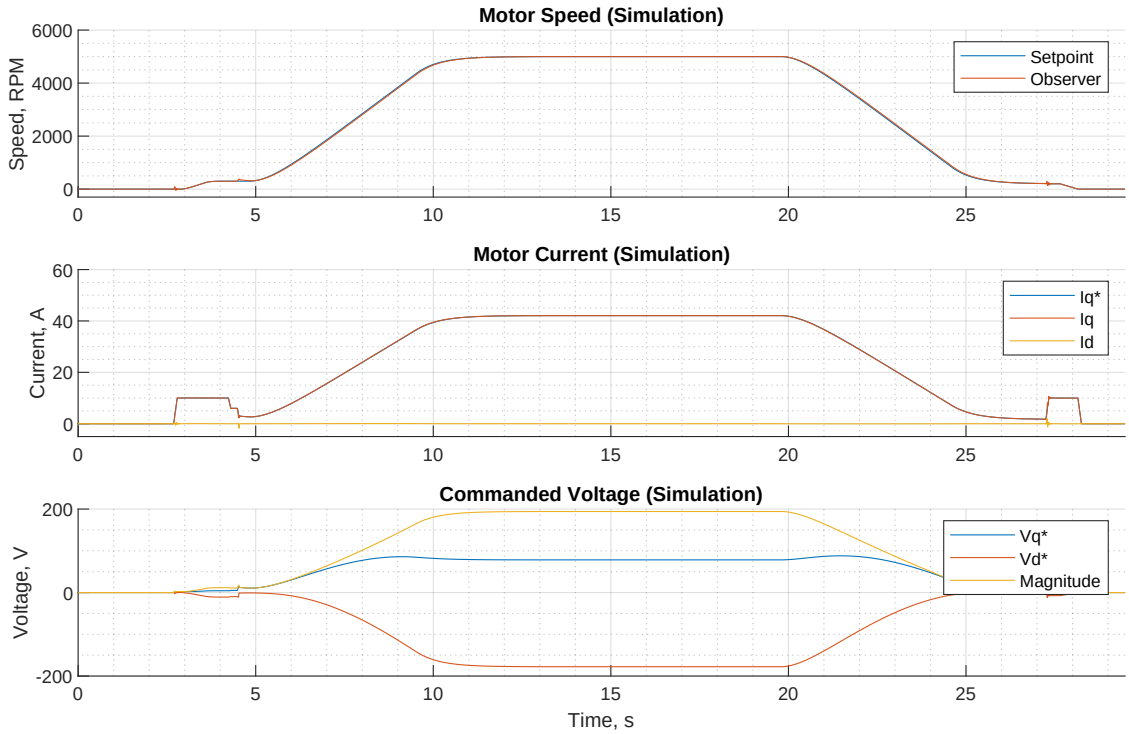

Fig. 4 Simulation data of the HLMC with HLM and HLP.